MySQL NDB Cluster API Developer Guide

MySQL NDB Cluster API Developer Guide

This is the *MySQL NDB Cluster API Developer Guide*, which provides information about developing applications using NDB Cluster as a data store. Application interfaces covered in this Guide include the low-level C++-language *NDB API* for the MySQL `NDB` storage engine, the C-language *MGM API* for communicating with and controlling NDB Cluster management servers, and the *MySQL Cluster Connector for Java*, which is a collection of Java APIs for writing applications against NDB Cluster, including JDBC, JPA, and ClusterJ.

NDB Cluster also provides support for the Memcache API; for more information, see Chapter 6, *ndbmemcache—Memcache API for NDB Cluster*.

NDB Cluster 7.3 and later also provides support for applications written using `Node.js`. See Chapter 5, *MySQL NoSQL Connector for JavaScript*, for more information.

This Guide includes concepts, terminology, class and function references, practical examples, common problems, and tips for using these APIs in applications.

For information about NDB internals that may be of interest to developers working with `NDB`, see NDB Cluster Internals Manual.

The information presented in this guide is current for recent releases of NDB Cluster up to and including NDB Cluster 7.5.5. Due to significant functional and other changes in NDB Cluster and its underlying APIs, you should not expect this information to apply to previous releases of the NDB Cluster software prior to NDB Cluster 7.2. Users of older NDB Cluster releases should upgrade to the latest available GA release of NDB Cluster 7.5.

For legal information, see the Legal Notices.

For help with using MySQL, please visit either the MySQL Forums or MySQL Mailing Lists, where you can discuss your issues with other MySQL users.

For additional documentation on MySQL products, including translations of the documentation into other languages, and downloadable versions in variety of formats, including HTML and PDF formats, see the MySQL Documentation Library.

Third-party licensing information. This product may include third-party software, used under license. If you are using a *Commercial* release of NDB Cluster, see this document for licensing information relating to third-party software that may be included in this Commercial release. If you are using a *Community* release of NDB Cluster, see this document for licensing information relating to third-party software that may be included in this Community release.

Document generated on: 2016-12-16 (revision: 50193)

Table of Contents

Preface and Legal Notices vii
1 NDB Cluster APIs: Overview and Concepts 1
1.1 NDB Cluster API Overview: Introduction 1
1.1.1 NDB Cluster API Overview: The NDB API 1
1.1.2 NDB Cluster API Overview: The MGM API 2
1.2 NDB Cluster API Overview: Terminology 2
1.3 The NDB Transaction and Scanning API 4
1.3.1 Core NDB API Classes 4
1.3.2 Application Program Basics 4
1.3.3 Review of NDB Cluster Concepts 12
1.3.4 The Adaptive Send Algorithm 13
2 The NDB API 15
2.1 Getting Started with the NDB API 16
2.1.1 Compiling and Linking NDB API Programs 16
2.1.2 Connecting to the Cluster 19
2.1.3 Mapping MySQL Database Object Names and Types to NDB 20
2.2 The NDB API Class Hierarachy 22
2.3 NDB API Classes, Interfaces, and Structures 23
2.3.1 The AutoGrowSpecification Structure 23
2.3.2 The Column Class 24
2.3.3 The Datafile Class 42
2.3.4 The Dictionary Class 48
2.3.5 The Element Structure 66
2.3.6 The Event Class 66
2.3.7 The EventBufferMemoryUsage Structure 77
2.3.8 The ForeignKey Class 78
2.3.9 The GetValueSpec Structure 85
2.3.10 The HashMap Class 86
2.3.11 The Index Class 90
2.3.12 The IndexBound Structure 97
2.3.13 The LogfileGroup Class 97
2.3.14 The List Class 102
2.3.15 The Key_part_ptr Structure 103
2.3.16 The Ndb Class 103
2.3.17 The Ndb_cluster_connection Class 121
2.3.18 The NdbBlob Class 131
2.3.19 The NdbDictionary Class 141
2.3.20 The NdbError Structure 146
2.3.21 The NdbEventOperation Class 150
2.3.22 The NdbIndexOperation Class 160
2.3.23 The NdbIndexScanOperation Class 162
2.3.24 The NdbInterpretedCode Class 167
2.3.25 The NdbOperation Class 192
2.3.26 The NdbRecAttr Class 205
2.3.27 The NdbRecord Interface 213
2.3.28 The NdbScanFilter Class 214
2.3.29 The NdbScanOperation Class 223
2.3.30 The NdbTransaction Class 232
2.3.31 The Object Class 249
2.3.32 The OperationOptions Structure 253
2.3.33 The PartitionSpec Structure 256
2.3.34 The RecordSpecification Structure 257
2.3.35 The ScanOptions Structure 259
2.3.36 The SetValueSpec Structure 261
2.3.37 The Table Class 262

2.3.38 The Tablespace Class 286
2.3.39 The Undofile Class 291
2.4 NDB API Errors and Error Handling 296
2.4.1 Handling NDB API Errors 297
2.4.2 NDB Error Codes: by Type 300
2.4.3 NDB Error Codes: Single Listing 327
2.4.4 NDB Error Classifications 359
2.5 NDB API Examples 359
2.5.1 NDB API Example Using Synchronous Transactions 360
2.5.2 NDB API Example Using Synchronous Transactions and Multiple Clusters 364
2.5.3 NDB API Example: Handling Errors and Retrying Transactions 369
2.5.4 NDB API Basic Scanning Example 373
2.5.5 NDB API Example: Using Secondary Indexes in Scans 386
2.5.6 NDB API Example: Using NdbRecord with Hash Indexes 390
2.5.7 NDB API Example Comparing RecAttr and NdbRecord 395
2.5.8 NDB API Event Handling Example 440
2.5.9 NDB API Example: Basic BLOB Handling 444
2.5.10 NDB API Example: Handling BLOB Columns and Values Using NdbRecord 452
2.5.11 NDB API Simple Array Example 460
2.5.12 NDB API Simple Array Example Using Adapter 466
2.5.13 Common Files for NDB API Array Examples 475
3 The MGM API 483
3.1 MGM API Concepts 483
3.1.1 Working with Log Events 484
3.1.2 Structured Log Events 484
3.2 MGM API Function Listing 485
3.2.1 Log Event Functions 485
3.2.2 MGM API Error Handling Functions 488
3.2.3 Management Server Handle Functions 490
3.2.4 Management Server Connection Functions 491
3.2.5 Cluster Status Functions 496
3.2.6 Functions for Starting & Stopping Nodes 498
3.2.7 Cluster Log Functions 503
3.2.8 Backup Functions 506
3.2.9 Single-User Mode Functions 507
3.3 MGM API Data Types 507
3.3.1 The ndb_mgm_node_type Type 508
3.3.2 The ndb_mgm_node_status Type 508
3.3.3 The ndb_mgm_error Type 508
3.3.4 The Ndb_logevent_type Type 508
3.3.5 The ndb_mgm_event_severity Type 513
3.3.6 The ndb_logevent_handle_error Type 513
3.3.7 The ndb_mgm_event_category Type 514
3.4 MGM API Structures 514
3.4.1 The ndb_logevent Structure 514
3.4.2 The ndb_mgm_node_state Structure 520
3.4.3 The ndb_mgm_cluster_state Structure 521
3.4.4 The ndb_mgm_reply Structure 521
3.5 MGM API Errors 521
3.5.1 Request Errors 522
3.5.2 Node ID Allocation Errors 522
3.5.3 Service Errors 522
3.5.4 Backup Errors 522
3.5.5 Single User Mode Errors 523
3.5.6 General Usage Errors 523
3.6 MGM API Examples 523
3.6.1 Basic MGM API Event Logging Example 523
3.6.2 MGM API Event Handling with Multiple Clusters 525

4 MySQL Cluster Connector for Java 529
4.1 MySQL Cluster Connector for Java: Overview 529
4.1.1 MySQL Cluster Connector for Java Architecture 529
4.1.2 Java and NDB Cluster 529
4.1.3 The ClusterJ API and Data Object Model 531
4.2 Using MySQL Cluster Connector for Java 533
4.2.1 Getting, Installing, and Setting Up MySQL Cluster Connector for Java 533
4.2.2 Using ClusterJ 535
4.2.3 Using JPA with NDB Cluster 543
4.2.4 Using Connector/J with NDB Cluster 544
4.3 ClusterJ API Reference 544
4.3.1 com.mysql.clusterj 544
4.3.2 com.mysql.clusterj.annotation 598
4.3.3 com.mysql.clusterj.query 609
4.3.4 Constant field values 616
4.4 MySQL Cluster Connector for Java: Limitations and Known Issues 618
5 MySQL NoSQL Connector for JavaScript 621
5.1 MySQL NoSQL Connector for JavaScript Overview 621
5.2 Installing the JavaScript Connector 621
5.3 Connector for JavaScript API Documentation 622
5.3.1 Batch 623
5.3.2 Context 623
5.3.3 Converter 625
5.3.4 Errors 626
5.3.5 Mynode 626
5.3.6 Session 629
5.3.7 SessionFactory 630
5.3.8 TableMapping and FieldMapping 630
5.3.9 TableMetadata 631
5.3.10 Transaction 632
5.4 Using the MySQL JavaScript Connector: Examples 633
5.4.1 Requirements for the Examples 633
5.4.2 Example: Finding Rows 637
5.4.3 Inserting Rows 639
5.4.4 Deleting Rows 640
6 ndbmemcache—Memcache API for NDB Cluster 643
6.1 Overview 643
6.2 Compiling NDB Cluster with Memcache Support 643
6.3 `memcached` command line options 643
6.4 NDB Engine Configuration 644
6.5 Memcache protocol commands 650
6.6 The memcached log file 652
6.7 Known Issues and Limitations of ndbmemcache 654
Index 655

Preface and Legal Notices

This is the *MySQL NDB Cluster API Developer Guide*, which provides information about developing applications using NDB Cluster as a data store. Application interfaces covered in this Guide include the low-level C++-language *NDB API* for the MySQL `NDB` storage engine, the C-language *MGM API* for communicating with and controlling NDB Cluster management servers, and the *MySQL Cluster Connector for Java*, which is a collection of Java APIs for writing applications against NDB Cluster, including JDBC, JPA, and ClusterJ.

NDB Cluster also provides support for the Memcache API; for more information, see Chapter 6, *ndbmemcache—Memcache API for NDB Cluster*.

NDB Cluster 7.3 and later also provides support for applications written using `Node.js`. See Chapter 5, *MySQL NoSQL Connector for JavaScript*, for more information.

This Guide includes concepts, terminology, class and function references, practical examples, common problems, and tips for using these APIs in applications. It also contains information about NDB internals that may be of interest to developers working with `NDB`, such as communication protocols employed between nodes, file systems used by management nodes and data nodes, error messages, and debugging (`DUMP`) commands in the management client.

The information presented in this guide is current for recent releases of NDB Cluster up to and including NDB Cluster 7.5.5. Due to significant functional and other changes in NDB Cluster and its underlying APIs, you should not expect this information to apply to previous releases of the NDB Cluster software prior to NDB Cluster 7.2. Users of older NDB Cluster releases should upgrade to the latest available GA release of NDB Cluster 7.5.

For legal information, see the Legal Notices.

Licensing information. This product may include third-party software, used under license. If you are using a *Commercial* release of NDB Cluster, see this document for licensing information, including licensing information relating to third-party software that may be included in this Commercial release. If you are using a *Community* release of NDB Cluster, see this document for licensing information, including licensing information relating to third-party software that may be included in this Community release.

Legal Notices

Copyright © 1997, 2016, Oracle and/or its affiliates. All rights reserved.

This software and related documentation are provided under a license agreement containing restrictions on use and disclosure and are protected by intellectual property laws. Except as expressly permitted in your license agreement or allowed by law, you may not use, copy, reproduce, translate, broadcast, modify, license, transmit, distribute, exhibit, perform, publish, or display any part, in any form, or by any means. Reverse engineering, disassembly, or decompilation of this software, unless required by law for interoperability, is prohibited.

The information contained herein is subject to change without notice and is not warranted to be error-free. If you find any errors, please report them to us in writing.

If this is software or related documentation that is delivered to the U.S. Government or anyone licensing it on behalf of the U.S. Government, then the following notice is applicable:

U.S. GOVERNMENT END USERS: Oracle programs, including any operating system, integrated software, any programs installed on the hardware, and/or documentation, delivered to U.S. Government end users are "commercial computer software" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, use, duplication, disclosure, modification, and adaptation of the programs, including any operating system, integrated software, any programs installed on the hardware, and/or documentation, shall be subject to license

terms and license restrictions applicable to the programs. No other rights are granted to the U.S. Government.

This software or hardware is developed for general use in a variety of information management applications. It is not developed or intended for use in any inherently dangerous applications, including applications that may create a risk of personal injury. If you use this software or hardware in dangerous applications, then you shall be responsible to take all appropriate fail-safe, backup, redundancy, and other measures to ensure its safe use. Oracle Corporation and its affiliates disclaim any liability for any damages caused by use of this software or hardware in dangerous applications.

Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

Intel and Intel Xeon are trademarks or registered trademarks of Intel Corporation. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. AMD, Opteron, the AMD logo, and the AMD Opteron logo are trademarks or registered trademarks of Advanced Micro Devices. UNIX is a registered trademark of The Open Group.

This software or hardware and documentation may provide access to or information about content, products, and services from third parties. Oracle Corporation and its affiliates are not responsible for and expressly disclaim all warranties of any kind with respect to third-party content, products, and services unless otherwise set forth in an applicable agreement between you and Oracle. Oracle Corporation and its affiliates will not be responsible for any loss, costs, or damages incurred due to your access to or use of third-party content, products, or services, except as set forth in an applicable agreement between you and Oracle.

Documentation Accessibility

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at
http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc.

Access to Oracle Support

Oracle customers that have purchased support have access to electronic support through My Oracle Support. For information, visit
http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info or visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs if you are hearing impaired.

This documentation is NOT distributed under a GPL license. Use of this documentation is subject to the following terms:

You may create a printed copy of this documentation solely for your own personal use. Conversion to other formats is allowed as long as the actual content is not altered or edited in any way. You shall not publish or distribute this documentation in any form or on any media, except if you distribute the documentation in a manner similar to how Oracle disseminates it (that is, electronically for download on a Web site with the software) or on a CD-ROM or similar medium, provided however that the documentation is disseminated together with the software on the same medium. Any other use, such as any dissemination of printed copies or use of this documentation, in whole or in part, in another publication, requires the prior written consent from an authorized representative of Oracle. Oracle and/or its affiliates reserve any and all rights to this documentation not expressly granted above.

Chapter 1 NDB Cluster APIs: Overview and Concepts

Table of Contents

1.1 NDB Cluster API Overview: Introduction 1
1.1.1 NDB Cluster API Overview: The NDB API 1
1.1.2 NDB Cluster API Overview: The MGM API 2
1.2 NDB Cluster API Overview: Terminology 2
1.3 The NDB Transaction and Scanning API 4
1.3.1 Core NDB API Classes 4
1.3.2 Application Program Basics 4
1.3.3 Review of NDB Cluster Concepts 12
1.3.4 The Adaptive Send Algorithm 13

This chapter provides a general overview of essential NDB Cluster, NDB API, and MGM API concepts, terminology, and programming constructs.

For an overview of Java APIs that can be used with NDB Cluster, see Section 4.1, “MySQL Cluster Connector for Java: Overview”.

For information about using Memcache with NDB Cluster, see Chapter 6, *ndbmemcache—Memcache API for NDB Cluster*.

For information about writing JavaScript applications using Node.js with MySQL, see Chapter 5, *MySQL NoSQL Connector for JavaScript*.

1.1 NDB Cluster API Overview: Introduction

This section introduces the NDB Transaction and Scanning APIs as well as the NDB Management (MGM) API for use in building applications to run on NDB Cluster. It also discusses the general theory and principles involved in developing such applications.

1.1.1 NDB Cluster API Overview: The NDB API

The *NDB API* is an object-oriented application programming interface for NDB Cluster that implements indexes, scans, transactions, and event handling. NDB transactions are ACID-compliant in that they provide a means to group operations in such a way that they succeed (commit) or fail as a unit (rollback). It is also possible to perform operations in a “no-commit” or deferred mode, to be committed at a later time.

NDB scans are conceptually rather similar to the SQL cursors implemented in MySQL 5.0 and other common enterprise-level database management systems. These provide high-speed row processing for record retrieval purposes. (NDB Cluster naturally supports set processing just as does MySQL in its non-Cluster distributions. This can be accomplished through the usual MySQL APIs discussed in the MySQL Manual and elsewhere.) The NDB API supports both table scans and row scans; the latter can be performed using either unique or ordered indexes. Event detection and handling is discussed in Section 2.3.21, “The NdbEventOperation Class”, as well as Section 2.5.8, “NDB API Event Handling Example”.

In addition, the NDB API provides object-oriented error-handling facilities in order to provide a means of recovering gracefully from failed operations and other problems. (See Section 2.5.3, “NDB API Example: Handling Errors and Retrying Transactions”, for a detailed example.)

The NDB API provides a number of classes implementing the functionality described above. The most important of these include the `Ndb`, `Ndb_cluster_connection`, `NdbTransaction`, and `NdbOperation` classes. These model (respectively) database connections, cluster connections,

transactions, and operations. These classes and their subclasses are listed in Section 2.3, “NDB API Classes, Interfaces, and Structures”. Error conditions in the NDB API are handled using `NdbError`.

Note

NDB API applications access the NDB Cluster's data store directly, without requiring a MySQL Server as an intermediary. This means that such applications are not bound by the MySQL privilege system; any NDB API application has read and write access to any `NDB` table stored in the same NDB Cluster at any time without restriction.

It is possible to distribute the MySQL grant tables, converting them from the default storage engine to `NDB`. Once this has been done, NDB API applications can access any of the MySQL grant tables. This means that such applications can read or write user names, passwords, and any other data stored in these tables.

1.1.2 NDB Cluster API Overview: The MGM API

The *NDB Cluster Management API*, also known as the *MGM API*, is a C-language programming interface intended to provide administrative services for the cluster. These include starting and stopping NDB Cluster nodes, handling NDB Cluster logging, backups, and restoration from backups, as well as various other management tasks. A conceptual overview of the MGM API and its uses can be found in Chapter 3, *The MGM API*.

The MGM API's principal structures model the states of individual modes (`ndb_mgm_node_state`), the state of the NDB Cluster as a whole (`ndb_mgm_cluster_state`), and management server response messages (`ndb_mgm_reply`). See Section 3.4, “MGM API Structures”, for detailed descriptions of these.

1.2 NDB Cluster API Overview: Terminology

This section provides a glossary of terms which are unique to the NDB and MGM APIs, or that have a specialized meaning when applied in the context of either or both of these APIs.

The terms in the following list are useful to an understanding of NDB Cluster, the NDB API, or have a specialized meaning when used in one of these:

Backup. A complete copy of all NDB Cluster data, transactions and logs, saved to disk.

Restore. Return the cluster to a previous state, as stored in a backup.

Checkpoint. Generally speaking, when data is saved to disk, it is said that a checkpoint has been reached. When working with the `NDB` storage engine, there are two sorts of checkpoints which work together in order to ensure that a consistent view of the cluster's data is maintained. These two types, *local checkpoints* and *global checkpoints*, are described in the next few paragraphs:

Local checkpoint (LCP). This is a checkpoint that is specific to a single node; however, LCPs take place for all nodes in the cluster more or less concurrently. An LCP involves saving all of a node's data to disk, and so usually occurs every few minutes, depending upon the amount of data stored by the node.

More detailed information about LCPs and their behavior can be found in the MySQL Manual; see in particular Defining NDB Cluster Data Nodes.

Global checkpoint (GCP). A GCP occurs every few seconds, when transactions for all nodes are synchronized and the REDO log is flushed to disk.

A related term is *GCI*, which stands for “Global Checkpoint ID”. This marks the point in the REDO log where a GCP took place.

Node. A component of NDB Cluster. 3 node types are supported:

- A *management (MGM) node* is an instance of `ndb_mgmd`, the NDB Cluster management server daemon.
- A *data node* an instance of `ndbd`, the NDB Cluster data storage daemon, and stores NDB Cluster data. This may also be an instance of `ndbmtd`, a multithreaded version of `ndbd`.
- An *API node*is an application that accesses NDB Cluster data. *SQL node* refers to a `mysqld` (MySQL Server) process that is connected to the NDB Cluster as an API node.

For more information about these node types, please refer to Section 1.3.3, “Review of NDB Cluster Concepts”, or to NDB Cluster Programs, in the *MySQL Manual*.

Node failure. An NDB Cluster is not solely dependent upon the functioning of any single node making up the cluster, which can continue to run even when one node fails.

Node restart. The process of restarting an NDB Cluster node which has stopped on its own or been stopped deliberately. This can be done for several different reasons, listed here:

- Restarting a node which has shut down on its own. (This is known as *forced shutdown* or *node failure*; the other cases discussed here involve manually shutting down the node and restarting it).
- To update the node's configuration.
- As part of a software or hardware upgrade.
- In order to defragment the node's `DataMemory`.

Initial node restart. The process of starting an NDB Cluster node with its file system having been removed. This is sometimes used in the course of software upgrades and in other special circumstances.

System crash (system failure). This can occur when so many data nodes have failed that the NDB Cluster's state can no longer be guaranteed.

System restart. The process of restarting an NDB Cluster and reinitializing its state from disk logs and checkpoints. This is required after any shutdown of the cluster, planned or unplanned.

Fragment. Contains a portion of a database table. In the `NDB` storage engine, a table is broken up into and stored as a number of subsets, usually referred to as fragments. A fragment is sometimes also called a *partition*.

Replica. Under the `NDB` storage engine, each table fragment has number of replicas in order to provide redundancy.

Transporter. A protocol providing data transfer across a network. The NDB API supports 4 different types of transporter connections: TCP/IP (local), TCP/IP (remote), SCI, and SHM. TCP/IP is, of course, the familiar network protocol that underlies HTTP, FTP, and so forth, on the Internet. SCI (Scalable Coherent Interface) is a high-speed protocol used in building multiprocessor systems and parallel-processing applications. SHM stands for Unix-style shared memory segments. For an informal introduction to SCI, see this essay at www.dolphinics.com.

NDB. This originally stood for “Network DataBase”. It now refers to the MySQL storage engine (named `NDB` or `NDBCLUSTER`) used to enable the NDB Cluster distributed database system.

ACC (Access Manager). An NDB kernel block that handles hash indexes of primary keys providing speedy access to the records. For more information, see The DBACC Block.

TUP (Tuple Manager). This NDB kernel block handles storage of tuples (records) and contains the filtering engine used to filter out records and attributes when performing reads or updates. See The DBTUP Block, for more information.

TC (Transaction Coordinator). Handles coordination of transactions and timeouts in the NDB kernel (see The DBTC Block). Provides interfaces to the NDB API for performing indexes and scan operations.

For more information, see NDB Kernel Blocks, elsewhere in this *Guide*..

See also NDB Cluster Overview, in the *MySQL Manual*.

1.3 The NDB Transaction and Scanning API

This section discusses the high-level architecture of the NDB API, and introduces the NDB classes which are of greatest use and interest to the developer. It also covers most important NDB API concepts, including a review of NDB Cluster Concepts.

1.3.1 Core NDB API Classes

The NDB API is an NDB Cluster application interface that implements transactions. It consists of the following fundamental classes:

- `Ndb_cluster_connection` represents a connection to a cluster.
- `Ndb` is the main class, and represents a connection to a database.
- `NdbDictionary` provides meta-information about tables and attributes.
- `NdbTransaction` represents a transaction.
- `NdbOperation` represents an operation using a primary key.
- `NdbScanOperation` represents an operation performing a full table scan.
- `NdbIndexOperation` represents an operation using a unique hash index.
- `NdbIndexScanOperation` represents an operation performing a scan using an ordered index.
- `NdbRecAttr` represents an attribute value.

In addition, the NDB API defines an `NdbError` structure, which contains the specification for an error.

It is also possible to receive events triggered when data in the database is changed. This is accomplished through the `NdbEventOperation` class.

The `NDB` event notification API is not supported prior to MySQL 5.1.

For more information about these classes as well as some additional auxiliary classes not listed here, see Section 2.3, “NDB API Classes, Interfaces, and Structures”.

1.3.2 Application Program Basics

The main structure of an application program is as follows:

1. Connect to a cluster using the `Ndb_cluster_connection` object.
2. Initiate a database connection by constructing and initialising one or more `Ndb` objects.
3. Identify the tables, columns, and indexes on which you wish to operate, using `NdbDictionary` and one or more of its subclasses.
4. Define and execute transactions using the `NdbTransaction` class.
5. Delete `Ndb` objects.

6. Terminate the connection to the cluster (terminate an instance of `Ndb_cluster_connection`).

1.3.2.1 Using Transactions

The procedure for using transactions is as follows:

1. Start a transaction (instantiate an `NdbTransaction` object).
2. Add and define operations associated with the transaction using instances of one or more of the `NdbOperation`, `NdbScanOperation`, `NdbIndexOperation`, and `NdbIndexScanOperation` classes.
3. Execute the transaction (call `NdbTransaction::execute()`).
4. The operation can be of two different types—`Commit` or `NoCommit`:
   - If the operation is of type `NoCommit`, then the application program requests that the operation portion of a transaction be executed, but without actually committing the transaction. Following the execution of a `NoCommit` operation, the program can continue to define additional transaction operations for later execution.

     `NoCommit` operations can also be rolled back by the application.
   - If the operation is of type `Commit`, then the transaction is immediately committed. The transaction must be closed after it has been committed (even if the commit fails), and no further operations can be added to or defined for this transaction.

   See Section 2.3.30.5, “NdbTransaction::ExecType”.

1.3.2.2 Synchronous Transactions

Synchronous transactions are defined and executed as follows:

1. Begin (create) the transaction, which is referenced by an `NdbTransaction` object typically created using `Ndb::startTransaction()`. At this point, the transaction is merely being defined; it is not yet sent to the NDB kernel.
2. Define operations and add them to the transaction, using one or more of the following, along with the appropriate methods of the respective`NdbOperation` class (or possibly one or more of its subclasses):
   - `NdbTransaction::getNdbOperation()`
   - `NdbTransaction::getNdbScanOperation()`
   - `NdbTransaction::getNdbIndexOperation()`
   - `NdbTransaction::getNdbIndexScanOperation()`

   At this point, the transaction has still not yet been sent to the NDB kernel.
3. Execute the transaction, using the `NdbTransaction::execute()` method.
4. Close the transaction by calling `Ndb::closeTransaction()`.

For an example of this process, see Section 2.5.1, “NDB API Example Using Synchronous Transactions”.

To execute several synchronous transactions in parallel, you can either use multiple `Ndb` objects in several threads, or start multiple application programs.

1.3.2.3 Operations

An `NdbTransaction` consists of a list of operations, each of which is represented by an instance of `NdbOperation`, `NdbScanOperation`, `NdbIndexOperation`, or `NdbIndexScanOperation` (that is, of `NdbOperation` or one of its child classes).

See NDB Access Types, for general information about NDB Cluster access operation types.

NDB Access Types

The data node process has a number of simple constructs which are used to access the data in an NDB Cluster. We have created a very simple benchmark to check the performance of each of these.

There are four access methods:

- **Primary key access.** This is access of a record through its primary key. In the simplest case, only one record is accessed at a time, which means that the full cost of setting up a number of TCP/IP messages and a number of costs for context switching are borne by this single request. In the case where multiple primary key accesses are sent in one batch, those accesses share the cost of setting up the necessary TCP/IP messages and context switches. If the TCP/IP messages are for different destinations, additional TCP/IP messages need to be set up.
- **Unique key access.** Unique key accesses are similar to primary key accesses, except that a unique key access is executed as a read on an index table followed by a primary key access on the table. However, only one request is sent from the MySQL Server, and the read of the index table is handled by the data node. Such requests also benefit from batching.
- **Full table scan.** When no indexes exist for a lookup on a table, a full table scan is performed. This is sent as a single request to the `ndbd` process, which then divides the table scan into a set of parallel scans on all `NDB` data node processes.
- **Range scan using ordered index.** When an ordered index is used, it performs a scan in the same manner as the full table scan, except that it scans only those records which are in the range used by the query transmitted by the MySQL server (SQL node). All partitions are scanned in parallel when all bound index attributes include all attributes in the partitioning key.

Single-row operations

After the operation is created using NdbTransaction::getNdbOperation() or NdbTransaction::getNdbIndexOperation(), it is defined in the following three steps:

1. Specify the standard operation type using `NdbOperation::readTuple()`.
2. Specify search conditions using `NdbOperation::equal()`.
3. Specify attribute actions using `NdbOperation::getValue()`.

Here are two brief examples illustrating this process. For the sake of brevity, we omit error handling.

This first example uses an `NdbOperation`:

```
// 1. Retrieve table object
myTable= myDict->getTable("MYTABLENAME");

// 2. Create an NdbOperation on this table
myOperation= myTransaction->getNdbOperation(myTable);

// 3. Define the operation's type and lock mode
myOperation->readTuple(NdbOperation::LM_Read);

// 4. Specify search conditions
myOperation->equal("ATTR1", i);

// 5. Perform attribute retrieval
myRecAttr= myOperation->getValue("ATTR2", NULL);
```

For additional examples of this sort, see Section 2.5.1, “NDB API Example Using Synchronous Transactions”.

The second example uses an `NdbIndexOperation`:

```
// 1. Retrieve index object
myIndex= myDict->getIndex("MYINDEX", "MYTABLENAME");

// 2. Create
myOperation= myTransaction->getNdbIndexOperation(myIndex);

// 3. Define type of operation and lock mode
myOperation->readTuple(NdbOperation::LM_Read);

// 4. Specify Search Conditions
myOperation->equal("ATTR1", i);

// 5. Attribute Actions
myRecAttr = myOperation->getValue("ATTR2", NULL);
```

Another example of this second type can be found in Section 2.5.5, “NDB API Example: Using Secondary Indexes in Scans”.

We now discuss in somewhat greater detail each step involved in the creation and use of synchronous transactions.

1. **Define single row operation type.** The following operation types are supported:

   - `NdbOperation::insertTuple()`: Inserts a nonexisting tuple.
   - `NdbOperation::writeTuple()`: Updates a tuple if one exists, otherwise inserts a new tuple.
   - `NdbOperation::updateTuple()`: Updates an existing tuple.
   - `NdbOperation::deleteTuple()`: Deletes an existing tuple.
   - `NdbOperation::readTuple()`: Reads an existing tuple using the specified lock mode.

   All of these operations operate on the unique tuple key. When `NdbIndexOperation` is used, then each of these operations operates on a defined unique hash index.

   Note

   If you want to define multiple operations within the same transaction, then you need to call `NdbTransaction::getNdbOperation()` or `NdbTransaction::getNdbIndexOperation()` for each operation.

2. **Specify Search Conditions.** The search condition is used to select tuples. Search conditions are set using `NdbOperation::equal()`.

3. **Specify Attribute Actions.** Next, it is necessary to determine which attributes should be read or updated. It is important to remember that:

   - Deletes can neither read nor set values, but only delete them.
   - Reads can only read values.
   - Updates can only set values. Normally the attribute is identified by name, but it is also possible to use the attribute's identity to determine the attribute.

   `NdbOperation::getValue()` returns an `NdbRecAttr` object containing the value as read. To obtain the actual value, one of two methods can be used; the application can either

   - Use its own memory (passed through a pointer *`aValue`*) to `NdbOperation::getValue()`, or

- receive the attribute value in an `NdbRecAttr` object allocated by the NDB API.

The `NdbRecAttr` object is released when `Ndb::closeTransaction()` is called. For this reason, the application cannot reference this object following any subsequent call to `Ndb::closeTransaction()`. Attempting to read data from an `NdbRecAttr` object before calling `NdbTransaction::execute()` yields an undefined result.

Scan Operations

Scans are roughly the equivalent of SQL cursors, providing a means to perform high-speed row processing. A scan can be performed on either a table (using an `NdbScanOperation`) or an ordered index (by means of an `NdbIndexScanOperation`).

Scan operations have the following characteristics:

- They can perform read operations which may be shared, exclusive, or dirty.
- They can potentially work with multiple rows.
- They can be used to update or delete multiple rows.
- They can operate on several nodes in parallel.

After the operation is created using `NdbTransaction::getNdbScanOperation()` or `NdbTransaction::getNdbIndexScanOperation()`, it is carried out as follows:

1. Define the standard operation type, using `NdbScanOperation::readTuples()`.

   Note

   See Section 2.3.29.7, “NdbScanOperation::readTuples()”, for additional information about deadlocks which may occur when performing simultaneous, identical scans with exclusive locks.

2. Specify search conditions, using `NdbScanFilter`, `NdbIndexScanOperation::setBound()`, or both.
3. Specify attribute actions using `NdbOperation::getValue()`.
4. Execute the transaction using `NdbTransaction::execute()`.
5. Traverse the result set by means of successive calls to `NdbScanOperation::nextResult()`.

Here are two brief examples illustrating this process. Once again, in order to keep things relatively short and simple, we forego any error handling.

This first example performs a table scan using an `NdbScanOperation`:

```
// 1. Retrieve a table object
myTable= myDict->getTable("MYTABLENAME");

// 2. Create a scan operation (NdbScanOperation) on this table
myOperation= myTransaction->getNdbScanOperation(myTable);

// 3. Define the operation's type and lock mode
myOperation->readTuples(NdbOperation::LM_Read);

// 4. Specify search conditions
NdbScanFilter sf(myOperation);
sf.begin(NdbScanFilter::OR);
sf.eq(0, i);   // Return rows with column 0 equal to i or
sf.eq(1, i+1); // column 1 equal to (i+1)
sf.end();
```

```
// 5. Retrieve attributes
myRecAttr= myOperation->getValue("ATTR2", NULL);
```

The second example uses an `NdbIndexScanOperation` to perform an index scan:

```
// 1. Retrieve index object
myIndex= myDict->getIndex("MYORDEREDINDEX", "MYTABLENAME");

// 2. Create an operation (NdbIndexScanOperation object)
myOperation= myTransaction->getNdbIndexScanOperation(myIndex);

// 3. Define type of operation and lock mode
myOperation->readTuples(NdbOperation::LM_Read);

// 4. Specify search conditions
// All rows with ATTR1 between i and (i+1)
myOperation->setBound("ATTR1", NdbIndexScanOperation::BoundGE, i);
myOperation->setBound("ATTR1", NdbIndexScanOperation::BoundLE, i+1);

// 5. Retrieve attributes
myRecAttr = MyOperation->getValue("ATTR2", NULL);
```

Some additional discussion of each step required to perform a scan follows:

1. **Define Scan Operation Type.** It is important to remember that only a single operation is supported for each scan operation (`NdbScanOperation::readTuples()` or `NdbIndexScanOperation::readTuples()`).

   Note

   If you want to define multiple scan operations within the same transaction, then you need to call `NdbTransaction::getNdbScanOperation()` or `NdbTransaction::getNdbIndexScanOperation()` separately for *each* operation.

2. **Specify Search Conditions.** The search condition is used to select tuples. If no search condition is specified, the scan will return all rows in the table. The search condition can be an `NdbScanFilter` (which can be used on both `NdbScanOperation` and `NdbIndexScanOperation`) or bounds (which can be used only on index scans - see `NdbIndexScanOperation::setBound()`). An index scan can use both `NdbScanFilter` and bounds.

   Note

   When NdbScanFilter is used, each row is examined, whether or not it is actually returned. However, when using bounds, only rows within the bounds will be examined.

3. **Specify Attribute Actions.** Next, it is necessary to define which attributes should be read. As with transaction attributes, scan attributes are defined by name, but it is also possible to use the attributes' identities to define attributes as well. As discussed elsewhere in this document (see Section 1.3.2.2, “Synchronous Transactions”), the value read is returned by the `NdbOperation::getValue()` method as an `NdbRecAttr` object.

Using Scans to Update or Delete Rows

Scanning can also be used to update or delete rows. This is performed as follows:

1. Scanning with exclusive locks using `NdbOperation::LM_Exclusive`.
2. (*When iterating through the result set*:) For each row, optionally calling either `NdbScanOperation::updateCurrentTuple()` or `NdbScanOperation::deleteCurrentTuple()`.

3. (*If performing `NdbScanOperation::updateCurrentTuple()`*:) Setting new values for records simply by using `NdbOperation::setValue()`. `NdbOperation::equal()` should not be called in such cases, as the primary key is retrieved from the scan.

Important

The update or delete is not actually performed until the next call to `NdbTransaction::execute()` is made, just as with single row operations. `NdbTransaction::execute()` also must be called before any locks are released; for more information, see Lock Handling with Scans.

Features Specific to Index Scans. When performing an index scan, it is possible to scan only a subset of a table using `NdbIndexScanOperation::setBound()`. In addition, result sets can be sorted in either ascending or descending order, using `NdbIndexScanOperation::readTuples()`. Note that rows are returned unordered by default unless *`sorted`* is set to `true`.

It is also important to note that, when using `NdbIndexScanOperation::BoundEQ` (see Section 2.3.23.1, "NdbIndexScanOperation::BoundType") with a partition key, only fragments containing rows will actually be scanned. Finally, when performing a sorted scan, any value passed as the `NdbIndexScanOperation::readTuples()` method's *`parallel`* argument will be ignored and maximum parallelism will be used instead. In other words, all fragments which it is possible to scan are scanned simultaneously and in parallel in such cases.

Lock Handling with Scans

Performing scans on either a table or an index has the potential to return a great many records; however, Ndb locks only a predetermined number of rows per fragment at a time. The number of rows locked per fragment is controlled by the batch parameter passed to `NdbScanOperation::readTuples()`.

In order to enable the application to handle how locks are released, `NdbScanOperation::nextResult()` has a Boolean parameter *`fetchAllowed`*. If `NdbScanOperation::nextResult()` is called with *`fetchAllowed`* equal to `false`, then no locks may be released as result of the function call. Otherwise the locks for the current batch may be released.

This next example shows a scan delete that handles locks in an efficient manner. For the sake of brevity, we omit error-handling.

```
int check;

// Outer loop for each batch of rows
while((check = MyScanOperation->nextResult(true)) == 0)
{
  do
  {
    // Inner loop for each row within the batch
    MyScanOperation->deleteCurrentTuple();
  }
  while((check = MyScanOperation->nextResult(false)) == 0);

  // When there are no more rows in the batch, execute all defined deletes
  MyTransaction->execute(NoCommit);
}
```

For a more complete example of a scan, see Section 2.5.4, "NDB API Basic Scanning Example".

Error Handling

Errors can occur either when operations making up a transaction are being defined, or when the transaction is actually being executed. Catching and handling either sort of error requires testing the value returned by `NdbTransaction::execute()`, and then, if an error is indicated (that is, if this value is equal to `-1`), using the following two methods in order to identify the error's type and location:

- `NdbTransaction::getNdbErrorOperation()` returns a reference to the operation causing the most recent error.
- `NdbTransaction::getNdbErrorLine()` yields the method number of the erroneous method in the operation, starting with `1`.

This short example illustrates how to detect an error and to use these two methods to identify it:

```
theTransaction = theNdb->startTransaction();
theOperation = theTransaction->getNdbOperation("TEST_TABLE");
if(theOperation == NULL)
  goto error;

theOperation->readTuple(NdbOperation::LM_Read);
theOperation->setValue("ATTR_1", at1);
theOperation->setValue("ATTR_2", at1);  //  Error occurs here
theOperation->setValue("ATTR_3", at1);
theOperation->setValue("ATTR_4", at1);

if(theTransaction->execute(Commit) == -1)
{
  errorLine = theTransaction->getNdbErrorLine();
  errorOperation = theTransaction->getNdbErrorOperation();
}
```

Here, `errorLine` is `3`, as the error occurred in the third method called on the `NdbOperation` object (in this case, `theOperation`). If the result of `NdbTransaction::getNdbErrorLine()` is `0`, then the error occurred when the operations were executed. In this example, `errorOperation` is a pointer to the object `theOperation`. The `NdbTransaction::getNdbError()` method returns an `NdbError` object providing information about the error.

Note

Transactions are *not* automatically closed when an error occurs. You must call `Ndb::closeTransaction()` or `NdbTransaction::close()` to close the transaction.

See Section 2.3.16.2, “Ndb::closeTransaction()”, and Section 2.3.30.1, “NdbTransaction::close()”.

One recommended way to handle a transaction failure (that is, when an error is reported) is as shown here:

1. Roll back the transaction by calling `NdbTransaction::execute()` with a special `ExecType` value for the *`type`* parameter.

   See Section 2.3.30.6, “NdbTransaction::execute()” and Section 2.3.30.5, “NdbTransaction::ExecType”, for more information about how this is done.

2. Close the transaction by calling `NdbTransaction::close()`.

3. If the error was temporary, attempt to restart the transaction.

Several errors can occur when a transaction contains multiple operations which are simultaneously executed. In this case the application must go through all operations and query each of their `NdbError` objects to find out what really happened.

Important

Errors can occur even when a commit is reported as successful. In order to handle such situations, the NDB API provides an additional `NdbTransaction::commitStatus()` method to check the transaction's commit status.

See Section 2.3.30.2, “NdbTransaction::commitStatus()”.

1.3.3 Review of NDB Cluster Concepts

This section covers the NDB Kernel, and discusses NDB Cluster transaction handling and transaction coordinators. It also describes NDB record structures and concurrency issues.

The *NDB Kernel* is the collection of data nodes belonging to an NDB Cluster. The application programmer can for most purposes view the set of all storage nodes as a single entity. Each data node is made up of three main components:

- **TC**: The transaction coordinator.
- **ACC**: The index storage component.
- **TUP**: The data storage component.

When an application executes a transaction, it connects to one transaction coordinator on one data node. Usually, the programmer does not need to specify which TC should be used, but in some cases where performance is important, the programmer can provide “hints” to use a certain TC. (If the node with the desired transaction coordinator is down, then another TC will automatically take its place.)

Each data node has an ACC and a TUP which store the indexes and data portions of the database table fragment. Even though a single TC is responsible for the transaction, several ACCs and TUPs on other data nodes might be involved in that transaction's execution.

1.3.3.1 Selecting a Transaction Coordinator

The default method is to select the transaction coordinator (TC) determined to be the "nearest" data node, using a heuristic for proximity based on the type of transporter connection. In order of nearest to most distant, these are:

1. SCI
2. SHM
3. TCP/IP (localhost)
4. TCP/IP (remote host)

If there are several connections available with the same proximity, one is selected for each transaction in a round-robin fashion. Optionally, you may set the method for TC selection to round-robin mode, where each new set of transactions is placed on the next data node. The pool of connections from which this selection is made consists of all available connections.

As noted in Section 1.3.3, “Review of NDB Cluster Concepts”, the application programmer can provide hints to the NDB API as to which transaction coordinator should be uses. This is done by providing a table and a partition key (usually the primary key). If the primary key is the partition key, then the transaction is placed on the node where the primary replica of that record resides. Note that this is only a hint; the system can be reconfigured at any time, in which case the NDB API chooses a transaction coordinator without using the hint. For more information, see Column::getPartitionKey(), and Section 2.3.16.34, “Ndb::startTransaction()”.

The application programmer can specify the partition key from SQL by using the following construct:

```
CREATE TABLE ... ENGINE=NDB PARTITION BY KEY (attribute_list);
```

For additional information, see Partitioning, and in particular KEY Partitioning, in the MySQL Manual.

1.3.3.2 NDB Record Structure

The NDB storage engine used by NDB Cluster is a relational database engine storing records in tables as with other relational database systems. Table rows represent records as tuples of relational data. When a new table is created, its attribute schema is specified for the table as a whole, and thus each table row has the same structure. Again, this is typical of relational databases, and NDB is no different in this regard.

Primary Keys. Each record has from 1 up to 32 attributes which belong to the primary key of the table.

Transactions. Transactions are committed first to main memory, and then to disk, after a global checkpoint (GCP) is issued. Since all data are (in most NDB Cluster configurations) synchronously replicated and stored on multiple data nodes, the system can handle processor failures without loss of data. However, in the case of a system-wide failure, all transactions (committed or not) occurring since the most recent GCP are lost.

Concurrency Control.
NDB uses *pessimistic concurrency control* based on locking. If a requested lock (implicit and depending on database operation) cannot be attained within a specified time, then a timeout error results.

Concurrent transactions as requested by parallel application programs and thread-based applications can sometimes deadlock when they try to access the same information simultaneously. Thus, applications need to be written in a manner such that timeout errors occurring due to such deadlocks are handled gracefully. This generally means that the transaction encountering a timeout should be rolled back and restarted.

Hints and Performance.
Placing the transaction coordinator in close proximity to the actual data used in the transaction can in many cases improve performance significantly. This is particularly true for systems using TCP/IP. For example, a Solaris system using a single 500 MHz processor has a cost model for TCP/IP communication which can be represented by the formula

```
[30 microseconds] + ([100 nanoseconds] * [number of bytes])
```

This means that if we can ensure that we use “popular” links we increase buffering and thus drastically reduce the costs of communication. The same system using SCI has a different cost model:

```
[5 microseconds] + ([10 nanoseconds] * [number of bytes])
```

This means that the efficiency of an SCI system is much less dependent on selection of transaction coordinators. Typically, TCP/IP systems spend 30 to 60% of their working time on communication, whereas for SCI systems this figure is in the range of 5 to 10%. Thus, employing SCI for data transport means that less effort from the NDB API programmer is required and greater scalability can be achieved, even for applications using data from many different parts of the database.

A simple example would be an application that uses many simple updates where a transaction needs to update one record. This record has a 32-bit primary key which also serves as the partitioning key. Then the `keyData` is used as the address of the integer of the primary key and `keyLen` is `4`.

1.3.4 The Adaptive Send Algorithm

Discusses the mechanics of transaction handling and transmission in NDB Cluster and the NDB API, and the objects used to implement these.

When transactions are sent using `NdbTransaction::execute()`, they are not immediately transferred to the NDB Kernel. Instead, transactions are kept in a special send list (buffer) in the `Ndb` object to which they belong. The adaptive send algorithm decides when transactions should actually be transferred to the NDB kernel.

The NDB API is designed as a multi-threaded interface, and so it is often desirable to transfer database operations from more than one thread at a time. The NDB API keeps track of which `Ndb` objects are active in transferring information to the NDB kernel and the expected number of threads to interact with the NDB kernel. Note that a given instance of `Ndb` should be used in at most one thread; different threads should *not* share the same `Ndb` object.

There are four conditions leading to the transfer of database operations from `Ndb` object buffers to the NDB kernel:

1. The NDB Transporter (TCP/IP, SCI, or shared memory) decides that a buffer is full and sends it off. The buffer size is implementation-dependent and may change between NDB Cluster releases. When TCP/IP is the transporter, the buffer size is usually around 64 KB. Since each `Ndb` object provides a single buffer per data node, the notion of a “full” buffer is local to each data node.

2. The accumulation of statistical data on transferred information may force sending of buffers to all storage nodes (that is, when all the buffers become full).

3. Every 10 milliseconds, a special transmission thread checks whether or not any send activity has occurred. If not, then the thread will force transmission to all nodes. This means that 20 ms is the maximum amount of time that database operations are kept waiting before being dispatched. A 10-millisecond limit is likely in future releases of NDB Cluster; checks more frequent than this require additional support from the operating system.

4. For methods that are affected by the adaptive send algorithm (such as `NdbTransaction::execute()`), there is a *`force`* parameter that overrides its default behavior in this regard and forces immediate transmission to all nodes. See the individual NDB API class listings for more information.

The conditions listed above are subject to change in future releases of NDB Cluster.

Chapter 2 The NDB API

Table of Contents

2.1 Getting Started with the NDB API 16
2.1.1 Compiling and Linking NDB API Programs 16
2.1.2 Connecting to the Cluster 19
2.1.3 Mapping MySQL Database Object Names and Types to NDB 20
2.2 The NDB API Class Hierarachy 22
2.3 NDB API Classes, Interfaces, and Structures 23
2.3.1 The AutoGrowSpecification Structure 23
2.3.2 The Column Class 24
2.3.3 The Datafile Class 42
2.3.4 The Dictionary Class 48
2.3.5 The Element Structure 66
2.3.6 The Event Class 66
2.3.7 The EventBufferMemoryUsage Structure 77
2.3.8 The ForeignKey Class 78
2.3.9 The GetValueSpec Structure 85
2.3.10 The HashMap Class 86
2.3.11 The Index Class 90
2.3.12 The IndexBound Structure 97
2.3.13 The LogfileGroup Class 97
2.3.14 The List Class 102
2.3.15 The Key_part_ptr Structure 103
2.3.16 The Ndb Class 103
2.3.17 The Ndb_cluster_connection Class 121
2.3.18 The NdbBlob Class 131
2.3.19 The NdbDictionary Class 141
2.3.20 The NdbError Structure 146
2.3.21 The NdbEventOperation Class 150
2.3.22 The NdbIndexOperation Class 160
2.3.23 The NdbIndexScanOperation Class 162
2.3.24 The NdbInterpretedCode Class 167
2.3.25 The NdbOperation Class 192
2.3.26 The NdbRecAttr Class 205
2.3.27 The NdbRecord Interface 213
2.3.28 The NdbScanFilter Class 214
2.3.29 The NdbScanOperation Class 223
2.3.30 The NdbTransaction Class 232
2.3.31 The Object Class 249
2.3.32 The OperationOptions Structure 253
2.3.33 The PartitionSpec Structure 256
2.3.34 The RecordSpecification Structure 257
2.3.35 The ScanOptions Structure 259
2.3.36 The SetValueSpec Structure 261
2.3.37 The Table Class 262
2.3.38 The Tablespace Class 286
2.3.39 The Undofile Class 291
2.4 NDB API Errors and Error Handling 296
2.4.1 Handling NDB API Errors 297
2.4.2 NDB Error Codes: by Type 300
2.4.3 NDB Error Codes: Single Listing 327
2.4.4 NDB Error Classifications 359
2.5 NDB API Examples 359
2.5.1 NDB API Example Using Synchronous Transactions 360

2.5.2 NDB API Example Using Synchronous Transactions and Multiple Clusters 364
2.5.3 NDB API Example: Handling Errors and Retrying Transactions 369
2.5.4 NDB API Basic Scanning Example 373
2.5.5 NDB API Example: Using Secondary Indexes in Scans 386
2.5.6 NDB API Example: Using NdbRecord with Hash Indexes 390
2.5.7 NDB API Example Comparing RecAttr and NdbRecord 395
2.5.8 NDB API Event Handling Example 440
2.5.9 NDB API Example: Basic BLOB Handling 444
2.5.10 NDB API Example: Handling BLOB Columns and Values Using NdbRecord 452
2.5.11 NDB API Simple Array Example 460
2.5.12 NDB API Simple Array Example Using Adapter 466
2.5.13 Common Files for NDB API Array Examples 475

This chapter contains information about the NDB API, which is used to write applications that access data in the `NDB` storage engine.

2.1 Getting Started with the NDB API

This section discusses preparations necessary for writing and compiling an NDB API application.

2.1.1 Compiling and Linking NDB API Programs

This section provides information on compiling and linking NDB API applications, including requirements and compiler and linker options.

2.1.1.1 General Requirements

To use the NDB API with MySQL, you must have the `libndbclient` client library and its associated header files installed alongside the regular MySQL client libraries and headers. These are automatically installed when you build MySQL using `-DWITH_NDBCLUSTER=ON` or use a MySQL binary package that supports the `NDB` storage engine.

This *Guide* is targeted for use with MySQL NDB Cluster 7.2 and later.

2.1.1.2 Compiler Options

Header Files. In order to compile source files that use the NDB API, you must ensure that the necessary header files can be found. Header files specific to the NDB and MGM APIs are installed in the following subdirectories of the MySQL `include` directory, respectively:

- `include/mysql/storage/ndb/ndbapi`
- `include/mysql/storage/ndb/mgmapi`

Compiler Flags. The MySQL-specific compiler flags needed can be determined using the `mysql_config` utility that is part of the MySQL installation:

```
$ mysql_config --cflags
-I/usr/local/mysql/include/mysql -Wreturn-type -Wtrigraphs -W -Wformat
-Wsign-compare -Wunused  -mcpu=pentium4 -march=pentium4
```

This sets the include path for the MySQL header files but not for those specific to the NDB API. The `--include` option to `mysql_config` returns the generic include path switch:

```
shell> mysql_config --include
-I/usr/local/mysql/include/mysql
```

It is necessary to add the subdirectory paths explicitly, so that adding all the needed compile flags to the `CXXFLAGS` shell variable should look something like this:

```
CFLAGS="$CFLAGS "`mysql_config --cflags`
CFLAGS="$CFLAGS "`mysql_config --include`/storage/ndb
CFLAGS="$CFLAGS "`mysql_config --include`/storage/ndb/ndbapi
CFLAGS="$CFLAGS "`mysql_config --include`/storage/ndb/mgmapi
```

Tip

If you do not intend to use the NDB Cluster management functions, the last line in the previous example can be omitted. However, if you are interested in the management functions only, and do not want or need to access NDB Cluster data except from MySQL, then you can omit the line referencing the `ndbapi` directory.

2.1.1.3 Linker Options

NDB API applications must be linked against both the MySQL and `NDB` client libraries. The `NDB` client library also requires some functions from the `mystrings` library, so this must be linked in as well.

The necessary linker flags for the MySQL client library are returned by `mysql_config --libs`. For multithreaded applications you should use the `--libs_r` instead:

```
$ mysql_config --libs_r
-L/usr/local/mysql-5.1/lib/mysql -lmysqlclient_r -lz -lpthread -lcrypt
-lnsl -lm -lpthread -L/usr/lib -lssl -lcrypto
```

It is now necessary only to add `-lndbclient` to `LD_FLAGS`, as shown here:

```
LDFLAGS="$LDFLAGS "`mysql_config --libs_r`
LDFLAGS="$LDFLAGS -lndbclient"
```

2.1.1.4 Using Autotools

It is often faster and simpler to use GNU autotools than to write your own makefiles. In this section, we provide an `autoconf` macro `WITH_MYSQL` that can be used to add a `--with-mysql` option to a configure file, and that automatically sets the correct compiler and linker flags for given MySQL installation.

All of the examples in this chapter include a common `mysql.m4` file defining `WITH_MYSQL`. A typical complete example consists of the actual source file and the following helper files:

- `acinclude`
- `configure.in`
- `Makefile.m4`

`automake` also requires that you provide `README`, `NEWS`, `AUTHORS`, and `ChangeLog` files; however, these can be left empty.

To create all necessary build files, run the following:

```
aclocal
autoconf
automake -a -c
configure --with-mysql=/mysql/prefix/path
```

Normally, this needs to be done only once, after which `make` will accommodate any file changes.

Example 1-1: acinclude.m4.

```
m4_include([../mysql.m4])
```

Example 1-2: configure.in.

```
AC_INIT(example, 1.0)
AM_INIT_AUTOMAKE(example, 1.0)
WITH_MYSQL()
AC_OUTPUT(Makefile)
```

Example 1-3: Makefile.am.

```
bin_PROGRAMS = example
example_SOURCES = example.cc
```

Example 1-4: WITH_MYSQL source for inclusion in acinclude.m4.

```
dnl
dnl configure.in helper macros
dnl

AC_DEFUN([WITH_MYSQL], [
  AC_MSG_CHECKING(for mysql_config executable)

  AC_ARG_WITH(mysql, [  --with-mysql=PATH path to mysql_config binary or mysql prefix dir], [
  if test -x $withval -a -f $withval
    then
      MYSQL_CONFIG=$withval
    elif test -x $withval/bin/mysql_config -a -f $withval/bin/mysql_config
    then
     MYSQL_CONFIG=$withval/bin/mysql_config
    fi
  ], [
  if test -x /usr/local/mysql/bin/mysql_config -a -f /usr/local/mysql/bin/mysql_config
    then
      MYSQL_CONFIG=/usr/local/mysql/bin/mysql_config
    elif test -x /usr/bin/mysql_config -a -f /usr/bin/mysql_config
    then
      MYSQL_CONFIG=/usr/bin/mysql_config
    fi
  ])

  if test "x$MYSQL_CONFIG" = "x"
  then
    AC_MSG_RESULT(not found)
    exit 3
  else
    AC_PROG_CC
    AC_PROG_CXX

    # add regular MySQL C flags
    ADDFLAGS=`$MYSQL_CONFIG --cflags`

    # add NDB API specific C flags
    IBASE=`$MYSQL_CONFIG --include`
    ADDFLAGS="$ADDFLAGS $IBASE/storage/ndb"
    ADDFLAGS="$ADDFLAGS $IBASE/storage/ndb/ndbapi"
    ADDFLAGS="$ADDFLAGS $IBASE/storage/ndb/mgmapi"

    CFLAGS="$CFLAGS $ADDFLAGS"
    CXXFLAGS="$CXXFLAGS $ADDFLAGS"

    LDFLAGS="$LDFLAGS "`$MYSQL_CONFIG --libs_r`" -lndbclient -lmystrings -lmysys"
    LDFLAGS="$LDFLAGS "`$MYSQL_CONFIG --libs_r`" -lndbclient -lmystrings"

    AC_MSG_RESULT($MYSQL_CONFIG)
  fi
])
```

2.1.2 Connecting to the Cluster

This section covers connecting an NDB API application to an NDB Cluster.

2.1.2.1 Include Files

NDB API applications require one or more of the following include files:

- Applications accessing NDB Cluster data using the NDB API must include the file `NdbApi.hpp`.
- Applications making use of the regular MySQL client API as well as the NDB API must also include `mysql.h` (in addition to `NdbApi.hpp`).
- Applications that use NDB Cluster management functions from the MGM API need the include file `mgmapi.h`.

2.1.2.2 API Initialization and Cleanup

Before using the NDB API, it must first be initialized by calling the `ndb_init()` function. Once an NDB API application is complete, call `ndb_end(0)` to perform any necessary cleanup. Both of these functions are defined in `storage/ndb/include/ndb_init.h`.

Note

It should be possible to use `fork()` in NDB API applications, but you must do so prior to calling `ndb_init()` or `my_init()` to avoid sharing of resources such as files and connections between processes.

2.1.2.3 Establishing the Connection

To establish a connection to the server, you must create an instance of `Ndb_cluster_connection`, whose constructor takes as its argument a cluster connection string. If no connection string is given, `localhost` is assumed.

The cluster connection is not actually initiated until the `Ndb_cluster_connection::connect()` method is called. When invoked without any arguments, the connection attempt is retried indefinitely, once per second, until successful. No reporting is done until the connection has been made.

By default an API node connects to the “nearest” data node. This is usually a data node running on the same machine as the nearest, due to the fact that shared memory transport can be used instead of the slower TCP/IP. This may lead to poor load distribution in some cases, so it is possible to enforce a round-robin node connection scheme by calling the `set_optimized_node_selection()` method with `0` as its argument prior to calling `connect()`.

The `connect()` method initiates a connection to an NDB Cluster management node only, without waiting for any connections to be made to data nodes. You ca override this behavior by using `wait_until_ready()` after calling `connect()`. The `wait_until_ready()` method waits up to a given number of seconds for a connection to a data node to be established.

In the following example, initialization and connection are handled in the two functions `example_init()` and `example_end()`, which are included in subsequent examples by means of including the file `example_connection.h`.

Example 2-1: Connection example.

```
#include <stdio.h>
#include <stdlib.h>
#include <NdbApi.hpp>
#include <mysql.h>
```

```
#include <mgmapi.h>

Ndb_cluster_connection* connect_to_cluster();
void disconnect_from_cluster(Ndb_cluster_connection *c);

Ndb_cluster_connection* connect_to_cluster()
{
  Ndb_cluster_connection* c;

  if(ndb_init())
    exit(EXIT_FAILURE);

  c= new Ndb_cluster_connection();

  if(c->connect(4, 5, 1))
  {
    fprintf(stderr, "Unable to connect to cluster within 30 seconds.\n\n");
    exit(EXIT_FAILURE);
  }

  if(c->wait_until_ready(30, 0) < 0)
  {
    fprintf(stderr, "Cluster was not ready within 30 seconds.\n\n");
    exit(EXIT_FAILURE);
  }

  return c;
}

void disconnect_from_cluster(Ndb_cluster_connection *c)
{
  delete c;

  ndb_end(2);
}

int main(int argc, char* argv[])
{
  Ndb_cluster_connection *ndb_connection= connect_to_cluster();

  printf("Connection Established.\n\n");

  disconnect_from_cluster(ndb_connection);

  return EXIT_SUCCESS;
}
```

2.1.3 Mapping MySQL Database Object Names and Types to NDB

This section discusses NDB naming and other conventions with regard to database objects.

Databases and Schemas. Databases and schemas are not represented by objects as such in the NDB API. Instead, they are modelled as attributes of `Table` and `Index` objects. The value of the `database` attribute of one of these objects is always the same as the name of the MySQL database to which the table or index belongs. The value of the `schema` attribute of a `Table` or `Index` object is always '`def`' (for “default”).

Tables. MySQL table names are directly mapped to `NDB` table names without modification. Table names starting with '`NDB$`' are reserved for internal use>, as is the `SYSTAB_0` table in the `sys` database.

Indexes. There are two different type of NDB indexes:

- *Hash indexes* are unique, but not ordered.
- *B-tree indexes* are ordered, but permit duplicate values.

Names of unique indexes and primary keys are handled as follows:

- For a MySQL `UNIQUE` index, both a B-tree and a hash index are created. The B-tree index uses the MySQL name for the index; the name for the hash index is generated by appending '`$unique`' to the index name.
- For a MySQL primary key only a B-tree index is created. This index is given the name `PRIMARY`. There is no extra hash; however, the uniqueness of the primary key is guaranteed by making the MySQL key the internal primary key of the `NDB` table.

Column Names and Values. `NDB` column names are the same as their MySQL names.

Data Types. MySQL data types are stored in `NDB` columns as follows:

- The MySQL `TINYINT`, `SMALLINT`, `INT`, and `BIGINT` data types map to `NDB` types having the same names and storage requirements as their MySQL counterparts.
- The MySQL `FLOAT` and `DOUBLE` data types are mapped to `NDB` types having the same names and storage requirements.
- The storage space required for a MySQL `CHAR` column is determined by the maximum number of characters and the column's character set. For most (but not all) character sets, each character takes one byte of storage. When using `utf8`, each character requires three bytes; `utfmb4` uses up to four bytes per character. You can find the maximum number of bytes needed per character in a given character set by checking the `Maxlen` column in the output of `SHOW CHARACTER SET`.
- An `NDB VARCHAR` column value maps to a MySQL `VARCHAR`, except that the first two bytes of the `NDB VARCHAR` are reserved for the length of the string. A utility function like that shown here can make a `VARCHAR` value ready for use in an NDB API application:

```
void make_ndb_varchar(char *buffer, char *str)
{
  int len = strlen(str);
  int hlen = (len > 255) ? 2 : 1;
  buffer[0] = len & 0xff;
  if( len > 255 )
    buffer[1] = (len / 256);
  strcpy(buffer+hlen, str);
}
```

  You can use this function as shown here:

```
char myVal[128+1]; // Size of myVal (+1 for length)
...
make_ndb_varchar(myVal, "NDB is way cool!!");
myOperation->setValue("myVal", myVal);
```

  See Section 2.5.11, “NDB API Simple Array Example”, for a complete example program that writes and reads `VARCHAR` and `VARBINARY` values to and from a table using the NDB API.

- MySQL storage requirements for a `VARCHAR` or `VARBINARY` column depend on whether the column is stored in memory or on disk:
  - For in-memory columns, the `NDB` storage engine supports variable-width columns with 4-byte alignment. This means that (for example) a the string `'abcde'` stored in a `VARCHAR(50)` column using the `latin1` character set requires 12 bytes—in this case, 2 bytes times 5 characters is 10, rounded up to the next even multiple of 4 yields 12.
  - For Disk Data columns, `VARCHAR` and `VARBINARY` are stored as fixed-width columns. This means that each of these types requires the same amount of storage as a `CHAR` of the same size.
- Each row in an NDB Cluster `BLOB` or `TEXT` column is made up of two separate parts. One of these is of fixed size (256 bytes), and is actually stored in the original table. The other consists of any data in excess of 256 bytes, which stored in a hidden table. The rows in this second table are always 2000 bytes long. This means that record of *`size`* bytes in a `TEXT` or `BLOB` column requires

- 256 bytes, if *`size`* `<= 256`
- `256 + 2000 * ((`*`size`*` – 256) \ 2000) + 1)` bytes otherwise

2.2 The NDB API Class Hierarachy

This section provides a hierarchical listing of all classes, interfaces, and structures exposed by the NDB API.

- `Ndb`
  - `Key_part_ptr`
  - `PartitionSpec`
- `NdbBlob`
- `Ndb_cluster_connection`
- `NdbDictionary`
  - `AutoGrowSpecification`
  - `Dictionary`
    - `List`
      - `Element`
  - `Column`
  - `Object`
    - `Datafile`
    - `Event`
    - `ForeignKey`
    - `HashMap`
    - `Index`
    - `LogfileGroup`
    - `Table`
    - `Tablespace`
    - `Undofile`
  - `RecordSpecification`
- `NdbError`
- `NdbEventOperation`
- `NdbInterpretedCode`
- `NdbOperation`
  - `NdbIndexOperation`

- NdbScanOperation
  - NdbIndexScanOperation
    - IndexBound
  - ScanOptions
- GetValueSpec
- SetValueSpec
- OperationOptions
- NdbRecAttr
- NdbRecord
- NdbScanFilter
- NdbTransaction

2.3 NDB API Classes, Interfaces, and Structures

This section provides a detailed listing of all classes, interfaces, and stuctures defined in the NDB API.

Each listing includes the following information:

- Description and purpose of the class, interface, or structure.
- Pointers, where applicable, to parent and child classes.
- A diagram of the class and its members.

Note

The sections covering the NdbDictionary and NdbOperation classes also include entity-relationship diagrams showing the hierarchy of inner classes, subclasses, and public type descending from them.

- Detailed listings of all public members, including descriptions of all method parameters and type values.

Class, interface, and structure descriptions are provided in alphabetic order. For a hierarchical listing, see Section 2.2, “The NDB API Class Hierarachy”.

2.3.1 The AutoGrowSpecification Structure

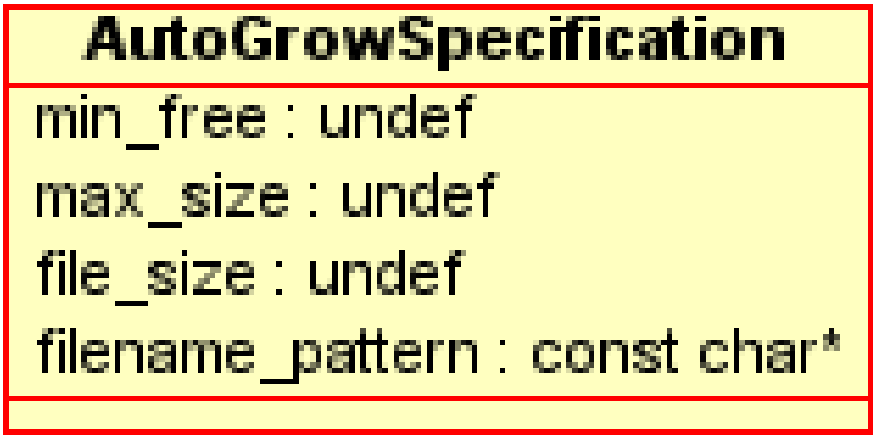
This section describes the AutoGrowSpecification structure.

Parent class. NdbDictionary

Description. The AutoGrowSpecification is a data structure defined in the NdbDictionary class, and is used as a parameter to or return value of some of the methods of the Tablespace and LogfileGroup classes. See Section 2.3.38, “The Tablespace Class”, and Section 2.3.13, “The LogfileGroup Class”, for more information.

Methods. AutoGrowSpecification has the following members, whose types are as shown in the following diagram:

Figure 2.1 NdbDictionary::AutoGrowSpecification



The purpose and use of each member can be found in the following table:

Name	Description
`min_free`	???
`max_size`	???
`file_size`	???
`filename_pattern`	???

2.3.2 The Column Class

This class represents a column in an NDB Cluster table.

Parent class. `NdbDictionary`

Child classes. *None*

Description. Each instance of `Column` is characterized by its type, which is determined by a number of type specifiers:

- Built-in type
- Array length or maximum length
- Precision and scale (*currently not in use*)
- Character set (applicable only to columns using string data types)
- Inline and part sizes (applicable only to `BLOB` columns)

These types in general correspond to MySQL data types and their variants. The data formats are same as in MySQL. The NDB API provides no support for constructing such formats; however, they are checked by the `NDB` kernel.

Methods. The following table lists the public methods of this class and the purpose or use of each method:

Method	Purpose / Use
`Column()`	Class constructor; there is also a copy constructor
`~Column()`	Class destructor
`equal()`	Compares `Column` objects
`getArrayType()`	Gets the column's array type

Method	Purpose / Use
`getCharset()`	Get the character set used by a string (text) column (not applicable to columns not storing character data)
`getColumnNo()`	Gets the column number
`getDefaultValue()`	Returns the column's default value
`getInlineSize()`	Gets the inline size of a `BLOB` column (not applicable to other column types)
`getLength()`	Gets the column's length
`getName()`	Gets the name of the column
`getNullable()`	Checks whether the column can be set to `NULL`
`getPartitionKey()`	Checks whether the column is part of the table's partitioning key
`getPartSize()`	Gets the part size of a `BLOB` column (not applicable to other column types)
`getPrecision()`	Gets the column's precision (used for decimal types only)
`getPrimaryKey()`	Check whether the column is part of the table's primary key
`getScale()`	Gets the column's scale (used for decimal types only)
`getSize()`	Gets the size of an element
`getSizeInBytesForRecord()`	Gets the space required for a column by `NdbRecord`, according to the column's type (added in NDB 7.3.10 and NDB 7.4.7)
`getStripeSize()`	Gets a BLOB column's stripe size (not applicable to other column types)
`getStorageType()`	Gets the storage type used by this column
`getType()`	Gets the column's type (`Type` value)
`setArrayType()`	Sets the column's `ArrayType`
`setCharset()`	Sets the character set used by a column containing character data (not applicable to nontextual columns)
`setDefaultValue()`	Sets the column's default value
`setInlineSize()`	Sets the inline size for a `BLOB` column (not applicable to non-`BLOB` columns)
`setLength()`	Sets the column's length
`setName()`	Sets the column's name
`setNullable()`	Toggles the column's nullability
`setPartitionKey()`	Determines whether the column is part of the table's partitioning key
`setPartSize()`	Sets the part size for a `BLOB` column (not applicable to non-`BLOB` columns)
`setPrecision()`	Sets the column's precision (used for decimal types only)
`setPrimaryKey()`	Determines whether the column is part of the primary key
`setScale()`	Sets the column's scale (used for decimal types only)
`setStorageType()`	Sets the storage type to be used by this column
`setStripeSize()`	Sets the stripe size for a `BLOB` column (not applicable to non-`BLOB` columns)
`setType()`	Sets the column's `Type`

For detailed descriptions, signatures, and examples of use for each of these methods, see Section 2.3.2.4, “Column Methods”.

Important

Columns created using this class cannot be seen by the MySQL Server. This means that they cannot be accessed by MySQL clients, and that they cannot be replicated. For these reasons, it is often preferable to avoid working with them.

Important

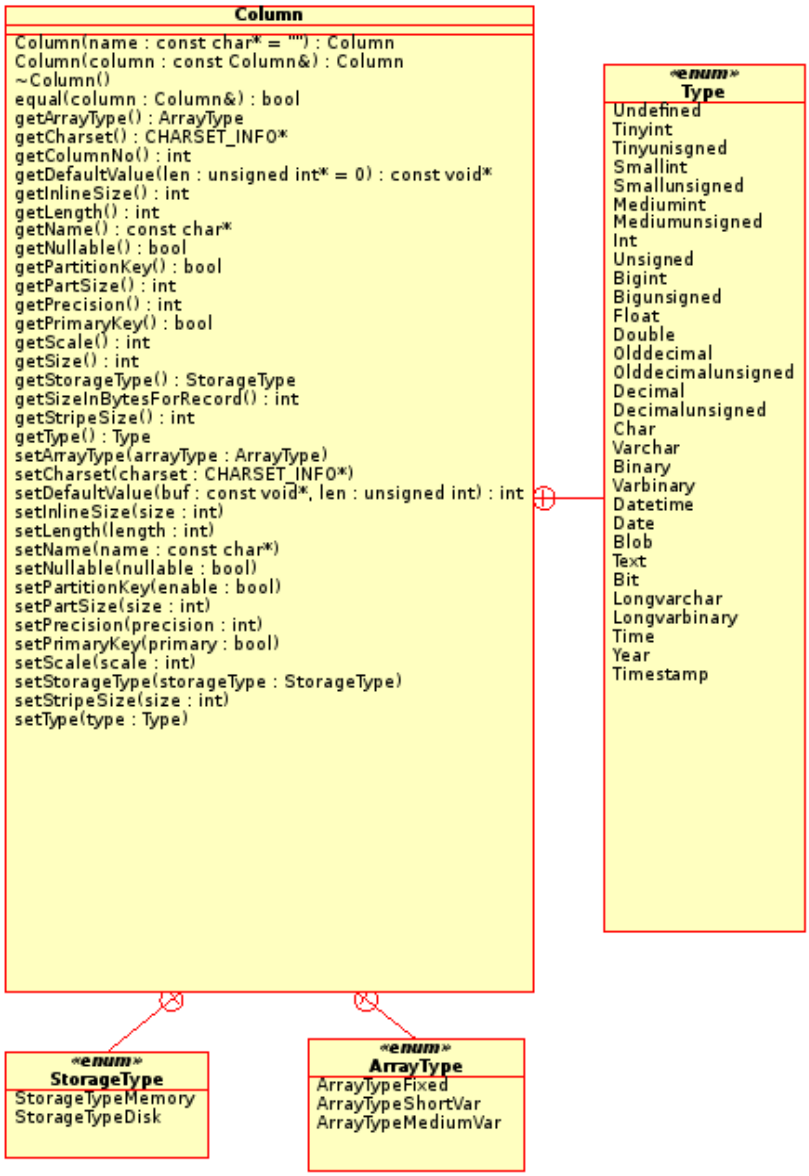
In the NDB API, column names are handled in case-sensitive fashion. (This differs from the MySQL C API.) To reduce the possibility for error, it is recommended that you name all columns consistently using uppercase or lowercase.

Types. These are the public types of the `Column` class:

Type	Purpose / Use
`ArrayType`	Specifies the column's internal storage format
`StorageType`	Determines whether the column is stored in memory or on disk
`Type`	The column's data type. `NDB` columns have the same data types as found in MySQL

Class diagram. This diagram shows all the available methods and enumerated types of the `Column` class:

Figure 2.2 NdbDictionary::Column



2.3.2.1 Column::ArrayType

This type describes the `Column`'s internal attribute format.

Description. The attribute storage format can be either fixed or variable.

Enumeration values. Possible values are shown, along with descriptions, in the following table:

Value	Description
`ArrayTypeFixed`	stored as a fixed number of bytes
`ArrayTypeShortVar`	stored as a variable number of bytes; uses 1 byte overhead
`ArrayTypeMediumVar`	stored as a variable number of bytes; uses 2 bytes overhead

The fixed storage format is faster but also generally requires more space than the variable format. The default is `ArrayTypeShortVar` for `Var*` types and `ArrayTypeFixed` for others. The default is usually sufficient.

2.3.2.2 Column::StorageType

This type describes the storage type used by a `Column` object.

Description. The storage type used for a given column can be either in memory or on disk. Columns stored on disk mean that less RAM is required overall but such columns cannot be indexed, and are potentially much slower to access. The default is `StorageTypeMemory`.

Enumeration values. Possible values are shown, along with descriptions, in the following table:

Value	Description
`StorageTypeMemory`	Store the column in memory
`StorageTypeDisk`	Store the column on disk

2.3.2.3 Column::Type

`Type` is used to describe the `Column` object's data type.

Description. Data types for `Column` objects are analogous to the data types used by MySQL. The types `Tinyint`, `Tinyintunsigned`, `Smallint`, `Smallunsigned`, `Mediumint`, `Mediumunsigned`, `Int`, `Unsigned`, `Bigint`, `Bigunsigned`, `Float`, and `Double` (that is, types `Tinyint` through `Double` in the order listed in the Enumeration Values table) can be used in arrays.

Enumeration values. Possible values are shown, along with descriptions, in the following table:

Value	Description
`Undefined`	Undefined
`Tinyint`	1-byte signed integer
`Tinyunsigned`	1-byte unsigned integer
`Smallint`	2-byte signed integer
`Smallunsigned`	2-byte unsigned integer
`Mediumint`	3-byte signed integer
`Mediumunsigned`	3-byte unsigned integer
`Int`	4-byte signed integer
`Unsigned`	4-byte unsigned integer
`Bigint`	8-byte signed integer
`Bigunsigned`	8-byte signed integer
`Float`	4-byte float
`Double`	8-byte float
`Olddecimal`	Signed decimal as used prior to MySQL 5.0
`Olddecimalunsigned`	Unsigned decimal as used prior to MySQL 5.0
`Decimal`	Signed decimal as used by MySQL 5.0 and later
`Decimalunsigned`	Unsigned decimal as used by MySQL 5.0 and later
`Char`	A fixed-length array of 1-byte characters; maximum length is 255 characters
`Varchar`	A variable-length array of 1-byte characters; maximum length is 255 characters
`Binary`	A fixed-length array of 1-byte binary characters; maximum length is 255 characters

Value	Description
`Varbinary`	A variable-length array of 1-byte binary characters; maximum length is 255 characters
`Datetime`	An 8-byte date and time value, with a precision of 1 second
`Date`	A 4-byte date value, with a precision of 1 day
`Blob`	A binary large object; see Section 2.3.18, “The NdbBlob Class”
`Text`	A text blob
`Bit`	A bit value; the length specifies the number of bits
`Longvarchar`	A 2-byte `Varchar`
`Longvarbinary`	A 2-byte `Varbinary`
`Time`	Time without date
`Year`	1-byte year value in the range 1901-2155 (same as MySQL)
`Timestamp`	Unix time

Caution

Do not confuse `Column::Type` with `Object::Type`.

2.3.2.4 Column Methods

This section documents the public methods of the `Column` class.

Note

The assignment (=) operator is overloaded for this class, so that it always performs a deep copy.

Warning

As with other database objects, `Column` object creation and attribute changes to existing columns done using the NDB API are not visible from MySQL. For example, if you change a column's data type using `Column::setType()`, MySQL will regard the type of column as being unchanged. The only exception to this rule with regard to columns is that you can change the name of an existing column using `Column::setName()`.

Also remember that the NDB API handles column names in case-sensitive fashion.

Column Constructor

Description. You can create a new `Column` or copy an existing one using the class constructor.

Warning

A `Column` created using the NDB API is *not* visible to a MySQL server.

The NDB API handles column names in case-sensitive fashion. For example, if you create a column named “myColumn”, you will not be able to access it later using “Mycolumn” for the name. You can reduce the possibility for error, by naming all columns consistently using only uppercase or only lowercase.

Signature. You can create either a new instance of the `Column` class, or by copying an existing `Column` object. Both of these are shown here:

- Constructor for a new `Column`:

```
Column
    (
      const char* name = ""
    )
```

- Copy constructor:

```
Column
    (
      const Column& column
    )
```

Parameters. When creating a new instance of `Column`, the constructor takes a single argument, which is the name of the new column to be created. The copy constructor also takes one parameter—in this case, a reference to the `Column` instance to be copied.

Return value. A `Column` object.

Destructor. The `Column` class destructor takes no arguments and *None*.

Column::equal()

Description. This method is used to compare one `Column` with another to determine whether the two `Column` objects are the same.

Signature.

```
bool equal
    (
      const Column& column
    ) const
```

Parameters. `equal()` takes a single parameter, a reference to an instance of `Column`.

Return value. `true` if the columns being compared are equal, otherwise `false`.

Column::getArrayType()

Description. This method gets the column's array type.

Signature.

```
ArrayType getArrayType
    (
      void
    ) const
```

Parameters. *None*.

Return value. An `ArrayType`; see Section 2.3.2.1, “Column::ArrayType” for possible values.

Column::getCharset()

Description. This gets the character set used by a text column.

Note

This method is applicable only to columns whose `Type` value is `Char`, `Varchar`, or `Text`.

Important

The NDB API handles column names in case-sensitive fashion; “myColumn” and “Mycolumn” are not considered to refer to the same column. It is recommended that you minimize the possibility of errors from using the wrong lettercase for column names by naming all columns consistently using only uppercase or only lowercase.

Signature.

```
CHARSET_INFO* getCharset
    (
      void
    ) const
```

Parameters. *None*.

Return value. A pointer to a `CHARSET_INFO` structure specifying both character set and collation. This is the same as a MySQL `MY_CHARSET_INFO` data structure; for more information, see mysql_get_character_set_info(),in the MySQL Manual.

Column::getColumnNo()

Description. This method gets the sequence number of a column within its containing table or index. If the column is part of an index (such as when returned by `getColumn()`), it is mapped to its position within that index, and not within the table containing the index.

Important

The NDB API handles column names in case-sensitive fashion, “myColumn” and “Mycolumn” are not considered to be the same column. It is recommended that you minimize the possibility of errors from using the wrong lettercase by naming all columns consistently using only uppercase or only lowercase.

Signature.

```
int getColumnNo
    (
      void
    ) const
```

Parameters. *None*.

Return value. The column number as an integer.

Column::getDefaultValue()

Description. Gets a column's default value data.

To determine whether a table has any columns with default values, use `Table::hasDefaultValues()`.

Signature.

```
const void* getDefaultValue
    (
      unsigned int* len = 0
    ) const
```

Parameters. *`len`* holds either the length of the default value data, or 0 in the event that the column is nullable or has no default value.

Return value. The default value data.

Column::getInlineSize()

Description. This method retrieves the inline size of a `BLOB` column—that is, the number of initial bytes to store in the table's blob attribute. This part is normally in main memory and can be indexed.

Note

This method is applicable only to `BLOB` columns.

Signature.

```
int getInlineSize
    (
      void
    ) const
```

Parameters. *None*.

Return value. The `BLOB` column's inline size, as an integer.

Column::getLength()

Description. This method gets the length of a column. This is either the array length for the column or—for a variable length array—the maximum length.

Important

The NDB API handles column names in case-sensitive fashion; “myColumn” and “Mycolumn” are not considered to refer to the same column. It is recommended that you minimize the possibility of errors from using the wrong lettercase for column names by naming all columns consistently using only uppercase or only lowercase.

Signature.

```
int getLength
    (
      void
    ) const
```

Parameters. *None*.

Return value. The (maximum) array length of the column, as an integer.

Column::getName()

Description. This method returns the name of the column for which it is called.

Important

The NDB API handles column names in case-sensitive fashion. For example, if you retrieve the name “myColumn” for a given column, attempting to access this column using “Mycolumn” for the name fails with an error such as `Column is NULL` or `Table definition has undefined column`. You can reduce the possibility for error, by naming all columns consistently using only uppercase or only lowercase.

Signature.

```
const char* getName
    (
      void
    ) const
```

Parameters. *None*.

Return value. The name of the column.

Column::getNullable()

Description. This method is used to determine whether the column can be set to `NULL`.

Signature.

```
bool getNullable
    (
      void
    ) const
```

Parameters. *None*.

Return value. A Boolean value: `true` if the column can be set to `NULL`, otherwise `false`.

Column::getPartitionKey()

Description. This method is used to check whether the column is part of the table's partitioning key.

Note

A *partitioning key* is a set of attributes used to distribute the tuples onto the data nodes. This key a hashing function specific to the `NDB` storage engine.

An example where this would be useful is an inventory tracking application involving multiple warehouses and regions, where it might be good to use the warehouse ID and district id as the partition key. This would place all data for a specific district and warehouse in the same database node. Locally to each fragment the full primary key will still be used with the hashing algorithm in such a case.

For more information about partitioning, partitioning schemes, and partitioning keys in MySQL, see Partitioning, in the MySQL Manual.

Important

The only type of user-defined partitioning that is supported for use with the `NDB` storage engine is key partitioning, including linear key partitioning.

Signature.

```
bool getPartitionKey
    (
      void
    ) const
```

Parameters. *None*.

Return value. `true` if the column is part of the partitioning key for the table, otherwise `false`.

Column::getPartSize()

Description. This method is used to get the part size of a `BLOB` column—that is, the number of bytes that are stored in each tuple of the blob table.

Note

This method is applicable to `BLOB` columns only.

Signature.

```
int getPartSize
    (
      void
    ) const
```

Parameters. *None*.

Return value. The column's part size, as an integer. In the case of a `Tinyblob` column, this value is `0` (that is, only inline bytes are stored).

Column::getPrecision()

Description. This method gets the precision of a column.

Note

This method is applicable to decimal columns only.

Signature.

```
int getPrecision
    (
      void
    ) const
```

Parameters. *None*.

Return value. The column's precision, as an integer. The precision is defined as the number of significant digits; for more information, see the discussion of the `DECIMAL` data type in Numeric Types, in the MySQL Manual.

Column::getPrimaryKey()

Description. This method is used to determine whether the column is part of the table's primary key.

Important

The NDB API handles column names in case-sensitive fashion; “myColumn” and “Mycolumn” are not considered to refer to the same column. It is recommended that you minimize the possibility of errors from using the wrong lettercase for column names by naming all columns consistently using only uppercase or only lowercase.

Signature.

```
bool getPrimaryKey
    (
      void
    ) const
```

Parameters. *None*.

Return value. A Boolean value: `true` if the column is part of the primary key of the table to which this column belongs, otherwise `false`.

Column::getScale()

Description. This method gets the scale used for a decimal column value.

Note

This method is applicable to decimal columns only.

Signature.

```
int getScale
    (
      void
    ) const
```

Parameters. *None*.

Return value. The decimal column's scale, as an integer. The scale of a decimal column represents the number of digits that can be stored following the decimal point. It is possible for this value to be `0`. For more information, see the discussion of the `DECIMAL` data type in Numeric Types, in the MySQL Manual.

Column::getSize()

Description. This function is used to obtain the size of a column.

Important

The NDB API handles column names in case-sensitive fashion; “myColumn” and “Mycolumn” are not considered to refer to the same column. It is recommended that you minimize the possibility of errors from using the wrong lettercase for column names by naming all columns consistently using only uppercase or only lowercase.

Signature.

```
int getSize
    (
      void
    ) const
```

Parameters. *None*.

Return value. The column's size in bytes (an integer value).

Column::getSizeInBytesForRecord()

Description. Gets the space required for a given column by an `NdbRecord`, depending on the column's type, as follows:

- For a BLOB column, this value is the same as `sizeof(NdbRecord*)`, which is 4 or 8 bytes (the size of a pointer; platform-dependent).
- For columns of all other types, it is the same as the value returned by `getSize()`.

This method was added in NDB 7.3.10 and NDB 7.4.7.

Signature.

```
int getSizeInBytesForRecord
    (
      void
    ) const
```

Parameters. *None*.

Return value. An integer (see Description).

Column::getStorageType()

Description. This method obtains a column's storage type.

Signature.

```
StorageType getStorageType
    (
      void
    ) const
```

Parameters. *None*.

Return value. A `StorageType` value; for more information about this type, see Section 2.3.2.2, “Column::StorageType”.

Column::getStripeSize()

Description. This method gets the stripe size of a `BLOB` column—that is, the number of consecutive parts to store in each node group.

Signature.

```
int getStripeSize
    (
      void
    ) const
```

Parameters. *None*.

Return value. The column's stripe size, as an integer.

Column::getType()

Description. This method gets the column's data type.

Important

The NDB API handles column names in case-sensitive fashion, “myColumn” and “Mycolumn” are not considered to be the same column. It is recommended that you minimize the possibility of errors from using the wrong lettercase by naming all columns consistently using only uppercase or only lowercase.

Signature.

```
Type getType
    (
      void
    ) const
```

Parameters. *None*.

Return value. The `Type` (data type) of the column. For a list of possible values, see Section 2.3.2.3, "Column::Type".

Column::setArrayType()

Description. Sets the array type for the column.

Signature.

```
void setArrayType
    (
      ArrayType type
    )
```

Parameters. A `Column::ArrayType` value. See Section 2.3.2.1, "Column::ArrayType", for more information.

Return value. *None*.

Column::setCharset()

Description. This method can be used to set the character set and collation of a `Char`, `Varchar`, or `Text` column.

Important

This method is applicable to `Char`, `Varchar`, and `Text` columns only.

Changes made to columns using this method are not visible to MySQL.

Signature.

```
void setCharset
    (
      CHARSET_INFO* cs
    )
```

Parameters. This method takes one parameter. *`cs`* is a pointer to a `CHARSET_INFO` structure. For additional information, see Column::getCharset().

Return value. *None*.

Column::setDefaultValue()

Description. This method sets a column value to its default, if it has one; otherwise it sets the column to `NULL`.

To determine whether a table has any columns with default values, use `Table::hasDefaultValues()`.

Signature.

```
int setDefaultValue
    (
      const void* buf,
      unsigned int len
    )
```

Parameters. This method takes 2 arguments: a value pointer *`buf`*; and the length *`len`* of the data, as the number of significant bytes. For fixed size types, this is the type size. For variable length types, the leading 1 or 2 bytes pointed to by *`buffer`* also contain size information as normal for the type.

Return value. 0 on success, 1 on failure..

Column::setInlineSize

Description. This method gets the inline size of a `BLOB` column—that is, the number of initial bytes to store in the table's blob attribute. This part is normally kept in main memory, and can be indexed and interpreted.

Important

This method is applicable to `BLOB` columns only.

Changes made to columns using this method are not visible to MySQL.

Signature.

```
void setInlineSize
    (
      int size
    )
```

Parameters. The integer *`size`* is the new inline size for the `BLOB` column.

Return value. *None*.

Column::setLength()

Description. This method sets the length of a column. For a variable-length array, this is the maximum length; otherwise it is the array length.

Important

Changes made to columns using this method are not visible to MySQL.

The NDB API handles column names in case-sensitive fashion; “myColumn” and “Mycolumn” are not considered to refer to the same column. It is recommended that you minimize the possibility of errors from using the wrong lettercase by naming all columns consistently using only uppercase or only lowercase.

Signature.

```
void setLength
    (
      int length
    )
```

Parameters. This method takes a single argument—the integer value *`length`* is the new length for the column.

Return value. *None*.

Column::setName()

Description. This method is used to set the name of a column.

Important

`setName()` is the only `Column` method whose result is visible from a MySQL Server. MySQL cannot see any other changes made to existing columns using the NDB API.

Signature.

```
void setName
    (
      const char* name
    )
```

Parameters. This method takes a single argument—the new name for the column.

Return value. This method *None*.

Column::setNullable()

Description. This method toggles the nullability of a column.

Important

Changes made to columns using this method are not visible to MySQL.

Signature.

```
void setNullable
    (
      bool nullable
    )
```

Parameters. A Boolean value. Using `true` makes it possible to insert `NULL`s into the column; if *`nullable`* is `false`, then this method performs the equivalent of changing the column to `NOT NULL` in MySQL.

Return value. *None*.

Column::setPartitionKey()

Description. This method makes it possible to add a column to the partitioning key of the table to which it belongs, or to remove the column from the table's partitioning key.

Important

Changes made to columns using this method are not visible to MySQL.

For additional information, see Column::getPartitionKey().

Signature.

```
void setPartitionKey
    (
      bool enable
    )
```

Parameters. The single parameter *`enable`* is a Boolean value. Passing `true` to this method makes the column part of the table's partitioning key; if *`enable`* is `false`, then the column is removed from the partitioning key.

Return value. *None*.

Column::setPartSize()

Description. This method sets the part size of a `BLOB` column—that is, the number of bytes to store in each tuple of the `BLOB` table.

Important

This method is applicable to `BLOB` columns only.

Changes made to columns using this method are not visible to MySQL.

Signature.

```
void setPartSize
    (
      int size
    )
```

Parameters. The integer *`size`* is the number of bytes to store in the `BLOB` table. Using zero for this value means only inline bytes can be stored, in effect making the column's type `TINYBLOB`.

Return value. *None*.

Column::setPrecision()

Description. This method can be used to set the precision of a decimal column.

Important

This method is applicable to decimal columns only.

Changes made to columns using this method are not visible to MySQL.

Signature.

```
void setPrecision
    (
      int precision
    )
```

Parameters. This method takes a single parameter—precision is an integer, the value of the column's new precision. For additional information about decimal precision and scale, see Column::getPrecision(), and Column::getScale().

Return value. *None*.

Column::setPrimaryKey()

Description. This method is used to make a column part of the table's primary key, or to remove it from the primary key.

Important

Changes made to columns using this method are not visible to MySQL.

The NDB API handles column names in case-sensitive fashion; “myColumn” and “Mycolumn” are not considered to refer to the same column. It is recommended that you minimize the possibility of errors from using the wrong lettercase for column names by naming all columns consistently using only uppercase or only lowercase.

Signature.

```
void setPrimaryKey
    (
      bool primary
    )
```

Parameters. This method takes a single Boolean value. If it is `true`, then the column becomes part of the table's primary key; if `false`, then the column is removed from the primary key.

Return value. *None*.

Column::setScale()

Description. This method can be used to set the scale of a decimal column.

Important

This method is applicable to decimal columns only.

Changes made to columns using this method are not visible to MySQL.

Signature.

```
void setScale
    (
      int scale
    )
```

Parameters. This method takes a single parameter—the integer *`scale`* is the new scale for the decimal column. For additional information about decimal precision and scale, see Column::getPrecision(), and Column::getScale().

Return value. *None*.

Column::setStripeSize()

Description. This method sets the stripe size of a `BLOB` column—that is, the number of consecutive parts to store in each node group.

Important

This method is applicable to `BLOB` columns only.

Changes made to columns using this method are not visible to MySQL.

Signature.

```
void setStripeSize
    (
      int size
    )
```

Parameters. This method takes a single argument. The integer *`size`* is the new stripe size for the column.

Return value. *None*.

Column::setStorageType()

Description. Sets the storage type for the column.

Signature.

```
void setStorageType
    (
      StorageType type
    )
```

Parameters. A `Column::StorageType` value. See Section 2.3.2.2, “Column::StorageType”, for more information.

Return value. *None*.

Column::setType()

Description. This method sets the `Type` (data type) of a column.

Important

`setType()` resets *all* column attributes to their (type dependent) default values; it should be the first method that you call when changing the attributes of a given column.

Changes made to columns using this method are not visible to MySQL.

The NDB API handles column names in case-sensitive fashion; “myColumn” and “Mycolumn” are not considered to refer to the same column. It is recommended that you minimize the possibility of errors from using the wrong lettercase for column names by naming all columns consistently using only uppercase or only lowercase.

Signature.

```
void setType
    (
      Type type
    )
```

Parameters. This method takes a single parameter—the new `Column::Type` for the column. The default is `Unsigned`. For a listing of all permitted values, see Section 2.3.2.3, “Column::Type”.

Return value. *None*.

2.3.3 The Datafile Class

This section covers the `Datafile` class.

Parent class. `Object`

Child classes. *None*

Description. The `Datafile` class models a Cluster Disk Data datafile, which is used to store Disk Data table data.

Note

Currently, only unindexed column data can be stored on disk. Indexes and indexed columns are stored in memory.

NDB Cluster prior to MySQL 5.1 did not support Disk Data storage and so did not support datafiles; thus the `Datafile` class is unavailable for NDB API applications written against these older releases.

Methods. The following table lists the public methods of this class and the purpose or use of each method:

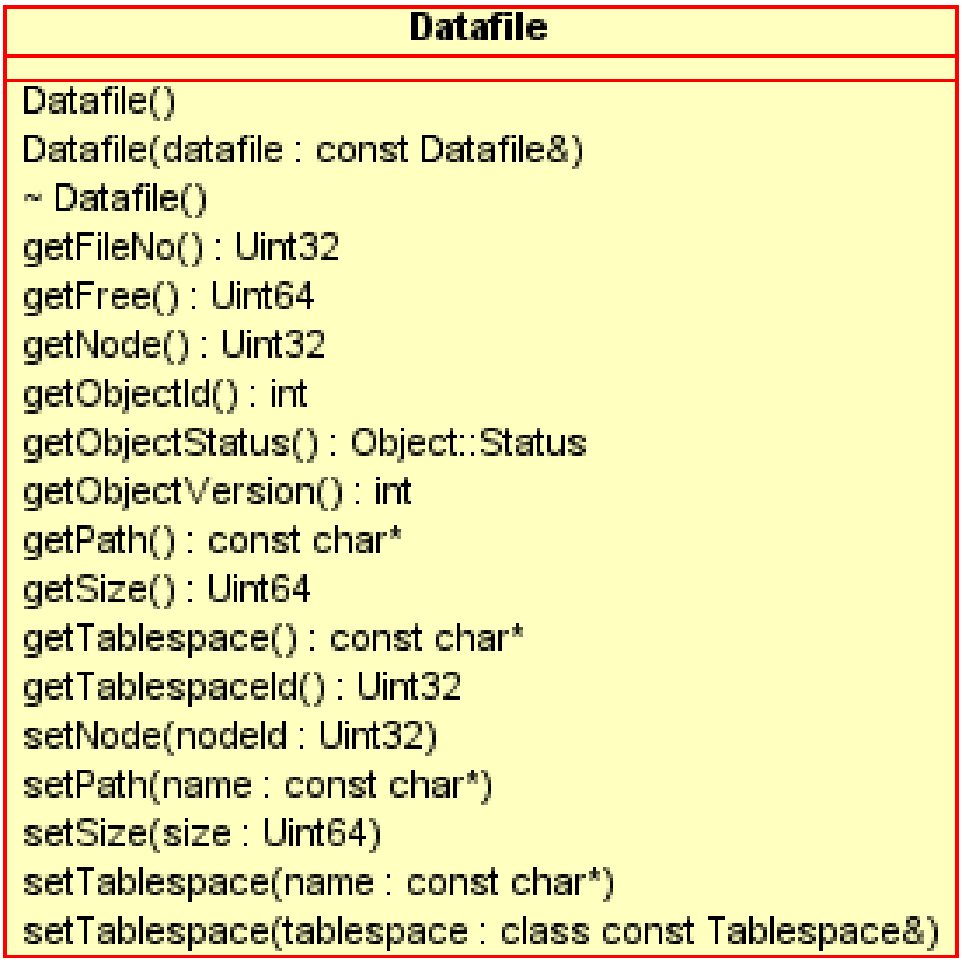
Method	Purpose / Use
`Datafile()`	Class constructor
`~Datafile()`	Destructor

Method	Purpose / Use
`getFileNo()`	Removed in NDB 7.5.0 (Bug #47960, Bug #11756088)
`getFree()`	Gets the amount of free space in the datafile
`getNode()`	Removed in NDB 7.5.0 (Bug #47960, Bug #11756088)
`getObjectId()`	Gets the datafile's object ID
`getObjectStatus()`	Gets the datafile's object status
`getObjectVersion()`	Gets the datafile's object version
`getPath()`	Gets the file system path to the datafile
`getSize()`	Gets the size of the datafile
`getTablespace()`	Gets the name of the tablespace to which the datafile belongs
`getTablespaceId()`	Gets the ID of the tablespace to which the datafile belongs
`setNode()`	Removed in NDB 7.5.0 (Bug #47960, Bug #11756088)
`setPath()`	Sets the name and location of the datafile on the file system
`setSize()`	Sets the datafile's size
`setTablespace()`	Sets the tablespace to which the datafile belongs

Types. The `Datafile` class defines no public types.

Class diagram. This diagram shows all the available methods of the `Datafile` class:

Figure 2.3 NdbDictionary::Object::Datafile



2.3.3.1 Datafile Class Constructor

Description. This method creates a new instance of `Datafile`, or a copy of an existing one.

Signature. To create a new instance:

```
Datafile
    (
      void
    )
```

To create a copy of an existing `Datafile` instance:

```
Datafile
    (
      const Datafile& datafile
    )
```

Parameters. New instance: *None*. Copy constructor: a reference to the `Datafile` instance to be copied.

Return value. A `Datafile` object.

2.3.3.2 Datafile::getFileNo()

Description. This method did not work as intended, and was removed in NDB 7.5.0 (Bug #47960, Bug #11756088).

Signature.

```
Uint32 getFileNo
    (
      void
    ) const
```

Parameters. *None*.

Return value. The file number, as an unsigned 32-bit integer.

2.3.3.3 Datafile::getFree()

Description. This method gets the free space available in the datafile.

Signature.

```
Uint64 getFree
    (
      void
    ) const
```

Parameters. *None*.

Return value. The number of bytes free in the datafile, as an unsigned 64-bit integer.

2.3.3.4 Datafile::getNode()

Description. This method did not work as intended, and was removed in NDB 7.5.0 (Bug #47960, Bug #11756088).

Signature.

```
Uint32 getNode
    (
      void
    ) const
```

Parameters. *None*.

Return value. The node ID as an unsigned 32-bit integer.

2.3.3.5 Datafile::getObjectId()

Description. This method is used to obtain the object ID of the datafile.

Signature.

```
virtual int getObjectId
    (
      void
    ) const
```

Parameters. *None*.

Return value. The datafile's object ID, as an integer.

2.3.3.6 Datafile::getObjectStatus()

Description. This method is used to obtain the datafile's object status.

Signature.

```
virtual Object::Status getObjectStatus
    (
      void
    ) const
```

Parameters. *None*.

Return value. The datafile's `Status`. See Section 2.3.31.4, “Object::Status”.

2.3.3.7 Datafile::getObjectVersion()

Description. This method retrieves the datafile's object version (see NDB Schema Object Versions).

Signature.

```
virtual int getObjectVersion
    (
      void
    ) const
```

Parameters. *None*.

Return value. The datafile's object version, as an integer.

2.3.3.8 Datafile::getPath()

Description. This method returns the file system path to the datafile.

Signature.

```
const char* getPath
    (
      void
    ) const
```

Parameters. *None*.

Return value. The path to the datafile on the data node's file system, a string (character pointer).

2.3.3.9 Datafile::getSize()

Description. This method gets the size of the datafile in bytes.

Signature.

```
Uint64 getSize
    (
      void
    ) const
```

Parameters. *None*.

Return value. The size of the data file, in bytes, as an unsigned 64-bit integer.

2.3.3.10 Datafile::getTablespace()

Description. This method can be used to obtain the name of the tablespace to which the datafile belongs.

Note

You can also access the associated tablespace's ID directly. See Section 2.3.3.11, “Datafile::getTablespaceId()”.

Signature.

```
const char* getTablespace
    (
      void
    ) const
```

Parameters. *None*.

Return value. The name of the associated tablespace (as a character pointer).

2.3.3.11 Datafile::getTablespaceId()

Description. This method gets the ID of the tablespace to which the datafile belongs.

Note

You can also access the name of the associated tablespace directly. See Section 2.3.3.10, “Datafile::getTablespace()”.

Signature.

```
Uint32 getTablespaceId
```

```
    (
      void
    ) const
```

Parameters. *None*.

Return value. This method returns the tablespace ID as an unsigned 32-bit integer.

2.3.3.12 Datafile::setNode()

Description. This method did not work as intended, and was removed in NDB 7.5.0 (Bug #47960, Bug #11756088).

Signature.

```
void setNode
    (
      Uint32 nodeId
    )
```

Parameters. The *`nodeId`* of the node on which the datafile is to be located (an unsigned 32-bit integer value).

Return value. *None*.

2.3.3.13 Datafile::setPath()

Description. This method sets the path to the datafile on the data node's file system.

Signature.

```
const char* setPath
    (
      void
    ) const
```

Parameters. The path to the file, a string (as a character pointer).

Return value. *None*.

2.3.3.14 Datafile::setSize()

Description. This method sets the size of the datafile.

Signature.

```
void setSize
    (
      Uint64 size
    )
```

Parameters. This method takes a single parameter—the desired *`size`* in bytes for the datafile, as an unsigned 64-bit integer.

Return value. *None*.

2.3.3.15 Datafile::setTablespace()

Description. This method is used to associate the datafile with a tablespace.

Signatures. `setTablespace()` can be invoked in either of two ways, listed here:

- Using the name of the tablespace, as shown here:

```
void setTablespace
    (
      const char* name
    )
```

- Using a reference to a `Tablespace` object.

```
void setTablespace
    (
      const class Tablespace& tablespace
    )
```

Parameters. This method takes a single parameter, which can be either one of the following:

- The *`name`* of the tablespace (as a character pointer).
- A reference *`tablespace`* to the corresponding `Tablespace` object.

Return value. *None*.

2.3.4 The Dictionary Class

This section describes the `Dictionary` class.

Parent class. `NdbDictionary`

Child classes. `List`

Description. This is used for defining and retrieving data object metadata. It also includes methods for creating and dropping database objects.

Methods. The following table lists the public methods of this class and the purpose or use of each method:

Method	Purpose / Use
`Dictionary()`	Class constructor method
`~Dictionary()`	Destructor method
`beginSchemaTrans()`	Begins a schema transaction
`createDatafile()`	Creates a datafile
`createEvent()`	Creates an event
`createForeignKey()`	Creates a foreign key
`createHashMap()`	Creates a has map
`createIndex()`	Creates an index
`createLogfileGroup()`	Creates a logfile group
`createRecord()`	Creates an `Ndbrecord` object
`createTable()`	Creates a table
`createTablespace()`	Creates a tablespace
`createUndofile()`	Creates an undofile
`dropDatafile()`	Drops a datafile
`dropEvent()`	Drops an event

Method	Purpose / Use
`dropForeignKey()`	Drops a foreign key
`dropIndex()`	Drops an index
`dropLogfileGroup()`	Drops a logfile group
`dropTable()`	Drops a table
`dropTablespace()`	Drops a tablespace
`dropUndofile()`	Drops an undofile
`endSchemaTrans()`	Ends (commits and closes) a schema transaction
`getDatafile()`	Gets the datafile having the given name
`getDefaultHashMap()`	Gets a table's default hash map
`getEvent()`	Gets the event having the given name
`getForeignKey()`	Gets the foreign key having the given name or reference
`getHashMap()`	Gets the hash map given its name or associated table
`getIndex()`	Gets the index having the given name
`getLogfileGroup()`	Gets the logfile group having the given name
`getNdbError()`	Retrieves the latest error
`getTable()`	Gets the table having the given name
`getTablespace()`	Gets the tablespace having the given name
`getUndofile()`	Gets the undofile having the given name
`hasSchemaTrans()`	Tells whether a schema transaction currently exists
`initDefaultHashMap()`	Initializes a atble' default hash map
`invalidateTable()`	Invalidates a table object
`listObjects()`	Fetches a list of the objects in the dictionary
`listIndexes()`	Fetches a list of the indexes defined on a given table
`listEvents()`	Fetches a list of the events defined in the dictionary
`prepareHashMap()`	Creates or retrieves a hash map that can be updated
`removeCachedTable()`	Removes a table from the local cache
`removeCachedIndex()`	Removes an index from the local cache

Important

Database objects such as tables and indexes created using the `Dictionary::create*()` methods cannot be seen by the MySQL Server. This means that they cannot be accessed by MySQL clients, and that they cannot be replicated. For these reasons, it is often preferable to avoid working with them.

Note

The `Dictionary` class does not have any methods for working directly with columns. You must use `Column` class methods for this purpose—see Section 2.3.2, “The Column Class”, for details.

Types. See Section 2.3.14, “The List Class”, and Section 2.3.5, “The Element Structure”.

Dictionary Class and Subclass Diagram. This diagram shows all the public members of the `Dictionary` class and its subclasses:

Figure 2.4 NdbDictionary::Dictionary

2.3.4.1 Dictionary Class Constructor

Description. This method creates a new instance of the `Dictionary` class.

Note

Both the constructor and destructor for this class are protected methods, rather than public.

Signature.

```
protected Dictionary
    (
      Ndb& ndb
    )
```

Parameters. An `Ndb` object.

Return value. A `Dictionary` object.

Destructor. The destructor takes no parameters and returns nothing.

```
protected ~Dictionary
    (
      void
    )
```

2.3.4.2 Dictionary::beginSchemaTrans()

Description. Starts a schema transaction. An error occurs if a transaction is already active, or if the kernel metadata is locked. You can determine whether a schema transaction already exists using the `hasSchemaTrans()` method.

A *metadata operation* occurs whenever data objects are created, altered, or dropped; such an operation can create additional suboperations in the NDB kernel.

The `Ndb` object and its associated `Dictionary` support one schema transaction at a time. By default, each metadata operation is executed separately; that is, for each operation, a schema transaction is started implicitly, the operation (including any suboperations) is executed, and the transaction is closed.

It is also possible to begin and end a schema transaction explicitly, and execute a set of user-defined operations atomically within its boundaries. In this case, all operations within the schema transaction either succeed, or are aborted and rolled back, as a unit. This is done by following the steps listed here:

1. To begin the schema transaction, call `beginSchemaTrans()`.
2. Execute the desired operations (such as `createTable()`).
3. End the schema transaction by calling `endSchemaTrans`.

Each operation is sent to the NDB kernel, which parses and saves it. A parse failure results in a rollback to the previous user operation before returning, at which point the user can either continue with or abort the entire transaction.

After all operations have been submitted, `endSchemaTrans()` processes and commits them. In the event of an error, the transaction is immediately aborted.

If the user exits before calling `endSchemaTrans()`, the NDB kernel aborts the transaction. If the user exits before the call to `endSchemaTrans()` returns, the kernel continues with the request, and its completion status is reported in the cluster log.

Signature.

```
int beginSchemaTrans
    (
      void
    )
```

Parameters. *None*.

Return value. Returns 0 on success, -1 on error.

2.3.4.3 Dictionary::createDatafile()

Description. This method creates a new datafile, given a `Datafile` object.

Signature.

```
int createDatafile
    (
      const Datafile& dFile
    )
```

Parameters. A single argument—a reference to an instance of `Datafile`—is required.

Return value. `0` on success, `-1` on failure.

2.3.4.4 Dictionary::createEvent()

Description. Creates an event, given a reference to an `Event` object.

You should keep in mind that the NDB API does not track allocated event objects, which means that the user must delete the `Event` that was obtained using `createEvent()`, after this object is no longer required.

Signature.

```
int createEvent
    (
      const Event& event
    )
```

Parameters. A reference *`event`* to an `Event` object.

Return value. `0` on success, `-1` on failure.

2.3.4.5 Dictionary::createForeignKey()

Description. Creates a `ForeignKey` object, given a reference to this object and an `Object` ID.

Signature.

```
int createForeignKey
    (
      const ForeignKey&,
      ObjectId* = 0,
      int flags = 0
    )
```

Parameters. A reference to the `ForeignKey` object, and an `Object` ID. An optional value *`flags`*, if used, allows the creation of the foreign key without performing any foreign key checks. If set, its value must be `CreateFK_NoVerify` (1).

Return value. `0` on success.

2.3.4.6 Dictionary::createHashMap()

Description. Creates a `HashMap`.

Signature.

```
int createHashMap
    (
      const HashMap& hashmap,
      ObjectId* id = 0
    )
```

Parameters. A reference to the hash map, and, optionally, an ID to be assigned to it. Added in NDB 7.2.7.

Return value. Returns 0 on success; on failure, returns -1 and sets an error.

2.3.4.7 Dictionary::createIndex()

Description. This method creates an index given an instance of `Index` and possibly an optional instance of `Table`.

Signature. This method can be invoked with or without a reference to a table object:

```
int createIndex
    (
      const Index& index
    )
```

```
int createIndex
    (
      const Index& index,
      const Table& table
    )
```

Parameters. *Required*: A reference to an `Index` object. *Optional*: A reference to a `Table` object.

Return value. `0` on success, `-1` on failure.

2.3.4.8 Dictionary::createLogfileGroup()

Description. This method creates a new logfile group, given an instance of `LogfileGroup`.

Signature.

```
int createLogfileGroup
    (
      const LogfileGroup& lGroup
    )
```

Parameters. A single argument, a reference to a `LogfileGroup` object, is required.

Return value. `0` on success, `-1` on failure.

2.3.4.9 Dictionary::createRecord()

Description. This method is used to create an `NdbRecord` object for use in table or index scanning operations.

Signature. The signature of this method depends on whether the resulting NdbRecord is to be used in table or index operations:

To create an `NdbRecord` for use in table operations, use the following:

```
NdbRecord* createRecord
    (
```

```
      const Table* table,
      const RecordSpecification* recSpec,
      Uint32 length,
      Uint32 elSize
    )
```

To create an `NdbRecord` for use in index operations, you can use either of the following:

```
NdbRecord* createRecord
    (
      const Index* index,
      const Table* table,
      const RecordSpecification* recSpec,
      Uint32 length,
      Uint32 elSize
    )
```

or

```
NdbRecord* createRecord
    (
      const Index* index,
      const RecordSpecification* recSpec,
      Uint32 length,
      Uint32 elSize
    )
```

Parameters. `Dictionary::createRecord()` takes the following parameters:

- If this `NdbRecord` is to be used with an index, a pointer to the corresponding `Index` object. If the `NdbRecord` is to be used with a table, this parameter is omitted. (See Section 2.3.11, “The Index Class”.)
- A pointer to a `Table` object representing the table to be scanned. If the `Ndbrecord` produced is to be used with an index, then this optionally specifies the table containing that index. (See Section 2.3.37, “The Table Class”.)
- A `RecordSpecification` used to describe a column. (See Section 2.3.34, “The RecordSpecification Structure”.
- The *`length`* of the record.
- The size of the elements making up this record.

Return value. An `NdbRecord` for use in operations involving the given table or index.

Example. See Section 2.3.27, “The NdbRecord Interface”.

2.3.4.10 Dictionary::createTable()

Description. Creates a table given an instance of `Table`.

Note

Tables created using this method cannot be seen by the MySQL Server, cannot be updated by MySQL clients, and cannot be replicated.

Signature.

```
int createTable
    (
      const Table& table
    )
```

Parameters. An instance of `Table`. See Section 2.3.37, “The Table Class”, for more information.

Return value. `0` on success, `-1` on failure.

2.3.4.11 Dictionary::createTablespace()

Description. This method creates a new tablespace, given a `Tablespace` object.

Signature.

```
int createTablespace
    (
      const Tablespace& tSpace
    )
```

Parameters. This method requires a single argument—a reference to an instance of `Tablespace`.

Return value. `0` on success, `-1` on failure.

2.3.4.12 Dictionary::createUndofile()

Description. This method creates a new undofile, given an `Undofile` object.

Signature.

```
int createUndofile
    (
      const Undofile& uFile
    )
```

Parameters. This method requires one argument: a reference to an instance of `Undofile`.

Return value. `0` on success, `-1` on failure.

2.3.4.13 Dictionary::dropDatafile()

Description. This method drops a data file, given a `Datafile` object.

Signature.

```
int dropDatafile
    (
      const Datafile& dFile
    )
```

Parameters. A single argument—a reference to an instance of `Datafile`—is required.

Return value. `0` on success, `-1` on failure.

2.3.4.14 Dictionary::dropEvent()

Description. This method drops an event, given a reference to an `Event` object.

Signature.

```
int dropEvent
    (
      const char* name,
      int         force = 0
    )
```

Parameters. This method takes two parameters:

- The *`name`* of the event to be dropped, as a string.
- By default, `dropEvent()` fails if the event specified does not exist. You can override this behavior by passing any nonzero value for the (optional) *`force`* argument; in this case no check is made as to whether there actually is such an event, and an error is returned only if the event exists but it was for whatever reason not possible to drop it.

Return value. `0` on success, `-1` on failure.

2.3.4.15 Dictionary::dropForeignKey()

Description. This method drops a foreign key, given a reference to an `ForeignKey` object to be dropped.

Signature.

```
int dropForeignKey
    (
      const ForeignKey&
    )
```

Parameters. A reference to the `ForeignKey` to be dropped.

Return value. `0` on success.

2.3.4.16 Dictionary::dropIndex()

Description. This method drops an index given an instance of `Index`, and possibly an optional instance of `Table`.

Signature.

```
int dropIndex
    (
      const Index& index
    )
```

```
int dropIndex
    (
      const Index& index,
      const Table& table
    )
```

Parameters. This method takes two parameters, one of which is optional:

- **Required.** A reference to an `Index` object.
- **Optional.** A reference to a `Table` object.

Return value. `0` on success, `-1` on failure.

2.3.4.17 Dictionary::dropLogfileGroup()

Description. Given an instance of `LogfileGroup`, this method drops the corresponding log file group.

Signature.

```
int dropLogfileGroup
    (
      const LogfileGroup& lGroup
    )
```

Parameters. A single argument, a reference to a `LogfileGroup` object, is required.

Return value. `0` on success, `-1` on failure.

2.3.4.18 Dictionary::dropTable()

Description. Drops a table given an instance of `Table`.

Signature.

```
int dropTable
    (
      const Table& table
    )
```

Note

In NDB 7.3.5 and later, this method drops all foreign key constraints on the *`table`* that is being dropped, whether the dropped table acts as a parent table, child table, or both. (Bug #18069680)

Parameters. An instance of `Table`. See Section 2.3.37, "The Table Class", for more information.

Return value. `0` on success, `-1` on failure.

2.3.4.19 Dictionary::dropTablespace()

Description. This method drops a tablespace, given a `Tablespace` object.

Signature.

```
int dropTablespace
    (
      const Tablespace& tSpace
    )
```

Parameters. This method requires a single argument—a reference to an instance of `Tablespace`.

Return value. `0` on success, `-1` on failure.

2.3.4.20 Dictionary::dropUndofile()

Description. This method drops an undo file, given an `Undofile` object.

Signature.

```
int dropUndofile
    (
      const Undofile& uFile
    )
```

Parameters. This method requires one argument: a reference to an instance of `Undofile`.

Return value. `0` on success, `-1` on failure.

2.3.4.21 Dictionary::endSchemaTrans()

Description. Ends a schema transaction begun with `beginSchemaTrans()`; causes operations to be processed and either committed, or aborted and rolled back. This method combines transaction execution and closing; separate methods for these tasks are not required (or implemented). This method may be called successfully even if no schema transaction is currently active.

Note

As with many other NDB API methods, it is entirely possible for `endSchemaTrans()` to overwrite any current error code. For this reason, you should first check for and save any error code that may have resulted from a previous, failed operation.

Signature.

```
int endSchemaTrans
    (
      Uint32 flags = 0
    )
```

Parameters. The flags determines how the completed transaction is handled. The default is 0, which causes the transaction to be committed.

Dictionary::SchemaTransFlag. You can also use with `endSchemaTrans()` either of the `SchemaTransFlag` values shown here:

- `SchemaTransAbort` (= 1): Causes the transaction to be aborted
- `SchemaTransBackground` (= 2): Causes the transaction to execute in the background; the result is written to the cluster log, while the application continues without waiting for a response.

Return value. Returns 0 on success; in the event of an error, returns -1 and sets an `NdbError` error code.

2.3.4.22 Dictionary::getDatafile()

Description. This method is used to retrieve a `Datafile` object, given the node ID of the data node where a datafile is located and the path to the datafile on that node's file system.

Signature.

```
Datafile getDatafile
    (
      Uint32      nodeId,
      const char* path
    )
```

Parameters. This method must be invoked using two arguments, as shown here:

- The 32-bit unsigned integer *`nodeId`* of the data node where the datafile is located
- The *`path`* to the datafile on the node's file system (string as character pointer)

Return value. A `Datafile` object—see Section 2.3.3, “The Datafile Class”, for details.

2.3.4.23 Dictionary::getDefaultHashMap()

Description. Get a table's default hash map.

Added in NDB 7.2.7.

Signature.

```
int getDefaultHashMap
    (
      HashMap& dst,
      Uint32 fragments
    )
```

or

```
int getDefaultHashMap
    (
      HashMap& dst,
      Uint32 buckets,
      Uint32 fragments
    )
```

Return value. Returns 0 on success; on failure, returns -1 and sets an error.

2.3.4.24 Dictionary::getEvent()

Description. This method is used to obtain a new `Event` object representing an event, given the event's name.

`getEvent()` allocates memory each time it is successfully called. You should keep in mind that successive invocations of this method using the same event name return multiple, distinct objects.

The NDB API does not track allocated event objects, which means that the user must delete each `Event` created using `getEvent()`, after the object is no longer required.

Signature.

```
const Event* getEvent
    (
      const char* eventName
    )
```

Parameters. The *`eventName`*, a string (character pointer).

Return value. A pointer to an `Event` object. See Section 2.3.6, “The Event Class”, for more information.

2.3.4.25 Dictionary::getForeignKey()

Description. This method is used to obtain a new `ForeignKey` object representing an event, given a reference to the foreign key and its name.

Signature.

```
int getForeignKey
    (
      ForeignKey& dst,
      const char* name
    )
```

Parameters. A reference to the foreign key and its *`name`*, a string (character pointer).

Return value. A pointer to a `ForeignKey` object.

2.3.4.26 Dictionary::getHashMap()

Description. Gets a hash map by name or by table.

Added in NDB 7.2.7.

Signature.

```
int getHashMap
    (
      HashMap& dst,
      const char* name
    )
```

or

```
int getHashMap
    (
      HashMap& dst,
      const Table* table
    )
```

Parameters. A reference to the hash map and either a name or a `Table`.

Return value. Returns 0 on success; on failure, returns -1 and sets an error.

2.3.4.27 Dictionary::getIndex()

Description. This method retrieves a pointer to an index, given the name of the index and the name of the table to which the table belongs.

Signature.

```
const Index* getIndex
    (
      const char* iName,
      const char* tName
    ) const
```

Parameters. Two parameters are required:

- The name of the index (*`iName`*)
- The name of the table to which the index belongs (*`tName`*)

Both of these are string values, represented by character pointers.

Return value. A pointer to an `Index`. See Section 2.3.11, “The Index Class”, for information about this object.

2.3.4.28 Dictionary::getLogfileGroup()

Description. This method gets a `LogfileGroup` object, given the name of the logfile group.

Signature.

```
LogfileGroup getLogfileGroup
    (
      const char* name
    )
```

Parameters. The *`name`* of the logfile group.

Return value. An instance of `LogfileGroup`; see Section 2.3.13, “The LogfileGroup Class”, for more information.

2.3.4.29 Dictionary::getNdbError()

Description. This method retrieves the most recent `NDB` API error.

Signature.

```
const struct NdbError& getNdbError
    (
      void
    ) const
```

Parameters. *None*.

Return value. A reference to an `NdbError` object. See Section 2.3.20, “The NdbError Structure”.

2.3.4.30 Dictionary::getTable()

Description. This method can be used to access the table with a known name. See Section 2.3.37, “The Table Class”.

Signature.

```
const Table* getTable
    (
      const char* name
    ) const
```

Parameters. The *`name`* of the table.

Return value. A pointer to the table, or `NULL` if there is no table with the *`name`* supplied.

2.3.4.31 Dictionary::getTablespace()

Description. Given either the name or ID of a tablespace, this method returns the corresponding `Tablespace` object.

Signatures. This method can be invoked in either of ways, as show here:

- Using the tablespace name:

  ```
  Tablespace getTablespace
      (
        const char* name
      )
  ```

- Using the tablespace ID:

  ```
  Tablespace getTablespace
      (
        Uint32 id
      )
  ```

Parameters. Either one of the following:

- The *`name`* of the tablespace, a string (as a character pointer)
- The unsigned 32-bit integer *`id`* of the tablespace

Return value. A `Tablespace` object, as discussed in Section 2.3.38, “The Tablespace Class”.

2.3.4.32 Dictionary::getUndofile()

Description. This method gets an `Undofile` object, given the ID of the node where an undofile is located and the file system path to the file.

Signature.

```
Undofile getUndofile
    (
      Uint32      nodeId,
      const char* path
    )
```

Parameters. This method requires the following two arguments:

- The *`nodeId`* of the data node where the undofile is located; this value is passed as a 32-bit unsigned integer
- The *`path`* to the undofile on the node's file system (string as character pointer)

Return value. An instance of `Undofile`. For more information, see Section 2.3.39, “The Undofile Class”.

2.3.4.33 Dictionary::hasSchemaTrans()

Description. Tells whether an NDB API schema transaction is ongoing.

Signature.

```
bool hasSchemaTrans
    (
      void
    ) const
```

Parameters. *None*.

Return value. Returns boolean `TRUE` if a schema transaction is in progress, otherwise `FALSE`.

2.3.4.34 Dictionary::initDefaultHashMap()

Description. Initialize a default hash map for a table.

Added in NDB 7.2.7.

Signature.

```
int initDefaultHashMap
    (
      HashMap& dst,
      Uint32 fragments
    )
```

or

```
int initDefaultHashMap
    (
      HashMap& dst,
      Uint32 buckets,
```

```
      Uint32 fragments
    )
```

Parameters. A reference to the hash map and the number of fragments. Optionally the number of buckets.

Return value. Returns 0 on success; on failure, returns -1 and sets an error.

2.3.4.35 Dictionary::invalidateIndex()

Description. This method is used to invalidate a cached index object.

Signature. The index invalidated by this method can be referenced either as an `Index` object (using a pointer), or by index name and table name, as shown here:

```
void invalidateIndex
    (
      const char* indexName,
      const char* tableName
    )

void invalidateIndex
    (
      const Index* index
    )
```

Parameters. The names of the index to be removed from the cache and the table to which it belongs (*`indexName`* and *`tableName`*, respectively), or a pointer to the corresponding `Index` object.

Return value. *None*.

2.3.4.36 DIctionary::invalidateTable()

Description. This method is used to invalidate a cached table object.

Signature.

```
void invalidateTable
    (
      const char* name
    )
```

It is also possibloe to use a `Table` object rather than the name of the table, as shown here:

```
void invalidateTable
    (
      const Table* table
    )
```

Parameters. The *`name`* of the table to be removed from the table cache, or a pointer to the corresponding `Table` object.

Return value. *None*.

2.3.4.37 Dictionary::listEvents()

Description. This method returns a list of all events defined within the dictionary.

Signature.

```
```

```
int listEvents
    (
      List& list
    )
```

Parameters. A reference to a `List` object. (See Section 2.3.14, “The List Class”.)

Return value. `0` on success; `-1` on failure.

2.3.4.38 Dictionary::listIndexes()

Description. This method is used to obtain a `List` of all the indexes on a table, given the table's name. (See Section 2.3.14, “The List Class”.)

Signature.

```
int listIndexes
    (
      List&      list,
      const char* table
) const
```

Parameters. `listIndexes()` takes two arguments, both of which are required:

- A reference to the `List` that contains the indexes following the call to the method
- The name of the *`table`* whose indexes are to be listed

Return value. `0` on success, `-1` on failure.

2.3.4.39 Dictionary::listObjects()

Description. This method is used to obtain a list of objects in the dictionary. It is possible to get all of the objects in the dictionary, or to restrict the list to objects of a single type.

Signature. This method has two signatures:

```
int listObjects
    (
      List&        list,
      Object::Type type = Object::TypeUndefined
    ) const
```

and

```
int listObjects
    (
      List&        list,
      Object::Type type,
      bool         fullyQualified
    ) const
```

Parameters. A reference to a `List` object is required—this is the list that contains the dictionary's objects after `listObjects()` is called. (See Section 2.3.14, “The List Class”.) An optional second argument *`type`* may be used to restrict the list to only those objects of the given type—that is, of the specified `Object::Type`. (See Section 2.3.31.6, “Object::Type”.) If *`type`* is not given, then the list contains all of the dictionary's objects.

You can also specify whether or not the object names in the *`list`* are fully qualified (that is, whether the object name includes the database, schema, and possibly the table name). If you specify *`fullyQualified`*, then you must also specify the *`type`*.

Return value. `0` on success, `-1` on failure.

2.3.4.40 Dictionary::prepareHashMap()

Description. Creates or retrieves a hash map suitable for alteration. Requires a schema transaction to be in progress; see Section 2.3.4.2, "Dictionary::beginSchemaTrans()", for more information.

Added in NDB 7.2.7.

Signature.

```
int prepareHashMap
    (
      const Table& oldTable,
      Table& newTable
    )
```

or

```
int prepareHashMap
    (
      const Table& oldTable,
      Table& newTable,
      Uint32 buckets
    )
```

Parameters. References to the old and new tables. Optionally, a number of buckets.

Return value. Returns 0 on success; on failure, returns -1 and sets an error.

2.3.4.41 Dictionary::releaseRecord()

Description. This method is used to free an `NdbRecord` after it is no longer needed.

Signature.

```
void releaseRecord
    (
      NdbRecord* record
    )
```

Parameters. The `NdbRecord` to be cleaned up.

Return value. *None*.

Example. See Section 2.3.27, "The NdbRecord Interface".

2.3.4.42 Dictionary::removeCachedTable()

Description. This method removes the specified table from the local cache.

Signature.

```
void removeCachedTable
    (
      const char* table
    )
```

Parameters. The name of the *`table`* to be removed from the cache.

Return value. *None*.

2.3.4.43 Dictionary::removeCachedIndex()

Description. This method removes the specified index from the local cache.

Signature.

```
void removeCachedIndex
    (
      const char* index,
      const char* table
    )
```

Parameters. The `removeCachedIndex()` requires two arguments:

- The name of the *`index`* to be removed from the cache
- The name of the *`table`* in which the index is found

Return value. *None*.

2.3.5 The Element Structure

This section discusses the `Element` structure.

Parent class. `List`

Description. The `Element` structure models an element of a list; it is used to store an object in a `List` populated by the `Dictionary` methods `listObjects()`, `listIndexes()`, and `listEvents()`.

Attributes. An `Element` has the attributes shown in the following table:

Attribute	Type	Initial Value	Description
`id`	`unsigned int`	`0`	The object's ID
`type`	`Object::Type`	`Object::TypeUndefined`	The object's type—see Section 2.3.31.6, “Object::Type” for possible values
`state`	`Object::State`	`Object::StateUndefined`	The object's state—see Section 2.3.31.3, “Object::State” for possible values
`store`	`Object::Store`	`Object::StoreUndefined`	How the object is stored—see Section 2.3.31.5, “Object::Store” for possible values
`database`	`char*`	`0`	The database in which the object is found
`schema`	`char*`	`0`	The schema in which the object is found
`name`	`char*`	`0`	The object's name

Note

For a graphical representation of this class and its parent-child relationships, see Section 2.3.4, “The Dictionary Class”.

2.3.6 The Event Class

This section discusses the `Event` class, its methods and defined types.

Parent class. `NdbDictionary`

Child classes. *None*

Description. This class represents a database event in an NDB Cluster.

Methods. The following table lists the public methods of the `Event` class and the purpose or use of each method:

Method	Purpose / Use
`Event()`	Class constructor
`~Event()`	Destructor
`addEventColumn()`	Adds a column on which events should be detected
`addEventColumns()`	Adds multiple columns on which events should be detected
`addTableEvent()`	Adds the type of event that should be detected
`getDurability()`	Gets the event's durability
`getEventColumn()`	Gets a column for which an event is defined
`getName()`	Gets the event's name
`getNoOfEventColumns()`	Gets the number of columns for which an event is defined
`getObjectId()`	Gets the event's object ID
`getObjectStatus()`	Gets the event's object status
`getObjectVersion()`	Gets the event's object version
`getReport()`	Gets the event's reporting options
`getTable()`	Gets the `Table` object on which the event is defined
`getTableEvent()`	Checks whether an event is to be detected
`getTableName()`	Gets the name of the table on which the event is defined
`mergeEvents()`	Sets the event's merge flag
`setDurability()`	Sets the event's durability
`setName()`	Sets the event's name
`setReport()`	The the event's reporting options
`setTable()`	Sets the `Table` object on which the event is defined

Improved Event API (NDB 7.4.3 and later). NDB 7.4.3 introduces an epoch-driven Event API that supercedes the earlier GCI-based model. The new version of the API also simplifies error detection and handling. These changes are realized in the NDB API by implementing a number of new methods for `Ndb` and `NdbEventOperation`, deprecating several other methods of both classes, and adding new type values to `TableEvent`.

Some of the new methods directly replace or stand in for deprecated methods, but not all of the deprecated methods map to new ones, some of which are entirely new. Old (deprecated) methods are shown in the first column of the following table, and new methods in the second column; old methods corresponding to new methods are shown in the same row.

Old method	New method
`NdbEventOperation::getEventType()`	`NdbEventOperation::getEventType2()`
`NdbEventOperation::getGCI()`	`NdbEventOperation::getEpoch`

Old method	New method
`NdbEventOperation::getLatestGCI()`	`Ndb::getHighestQueuedEpoch()`
`NdbEventOperation::isOverrun()`	None; use `NdbEventOperation::getEventType2()`
`NdbEventOperation::hasError()`	None; use `NdbEventOperation::getEventType2()`
`NdbEventOperation::clearError()`	None
None	`NdbEventOperation::isEmptyEpoch()`
None	`NdbEventOperation::isErrorEpoch()`
`Ndb::pollEvents()`	`Ndb::pollEvents2()`
`Ndb::nextEvent()`	`Ndb::nextEvent2()`
`Ndb::getLatestGCI()`	`Ndb::getHighestQueuedEpoch()`
`Ndb::getGCIEventOperations()`	`Ndb::getNextEventOpInEpoch2()`
`Ndb::isConsistent()`	None
`Ndb::isConsistentGCI()`	None

Error handling using the new API is accomplished by checking the value returned from `getEventType2()`, and is no longer handled using the methods `hasError()` and `clearError()`, which are now deprecated and subject to removal in a future release of NDB Cluster. In support of this change, the range of possible `TableEvent` types has been expanded by those listed here:

- `TE_EMPTY`: Empty epoch
- `TE_INCONSISTENT`: Inconsistent epoch; missing data or overflow
- `TE_OUT_OF_MEMORY`: Inconsistent data; event buffer out of memory or overflow

The result of these changes is that, in NDB 7.4.3 and later, you can check for errors while checking a table event's type, as shown here:

```
NdbDictionary::Event::TableEvent* error_type = 0;
NdbEventOperation* pOp = nextEvent2();

if (pOp->isErrorEpoch(error_type)
{
  switch (error_type)
  {
    case TE_INCONSISTENT :
      // Handle error/inconsistent epoch...
      break;

    case TE_OUT_OF_MEMORY :
      // Handle error/inconsistent data...
      break;

    //  ...
  }
}
```

For more information, see the detailed descriptions for the `Ndb` and `NdbEventOperation` methods shown in the table previously, as well as Section 2.3.6.23, “Event::TableEvent”.

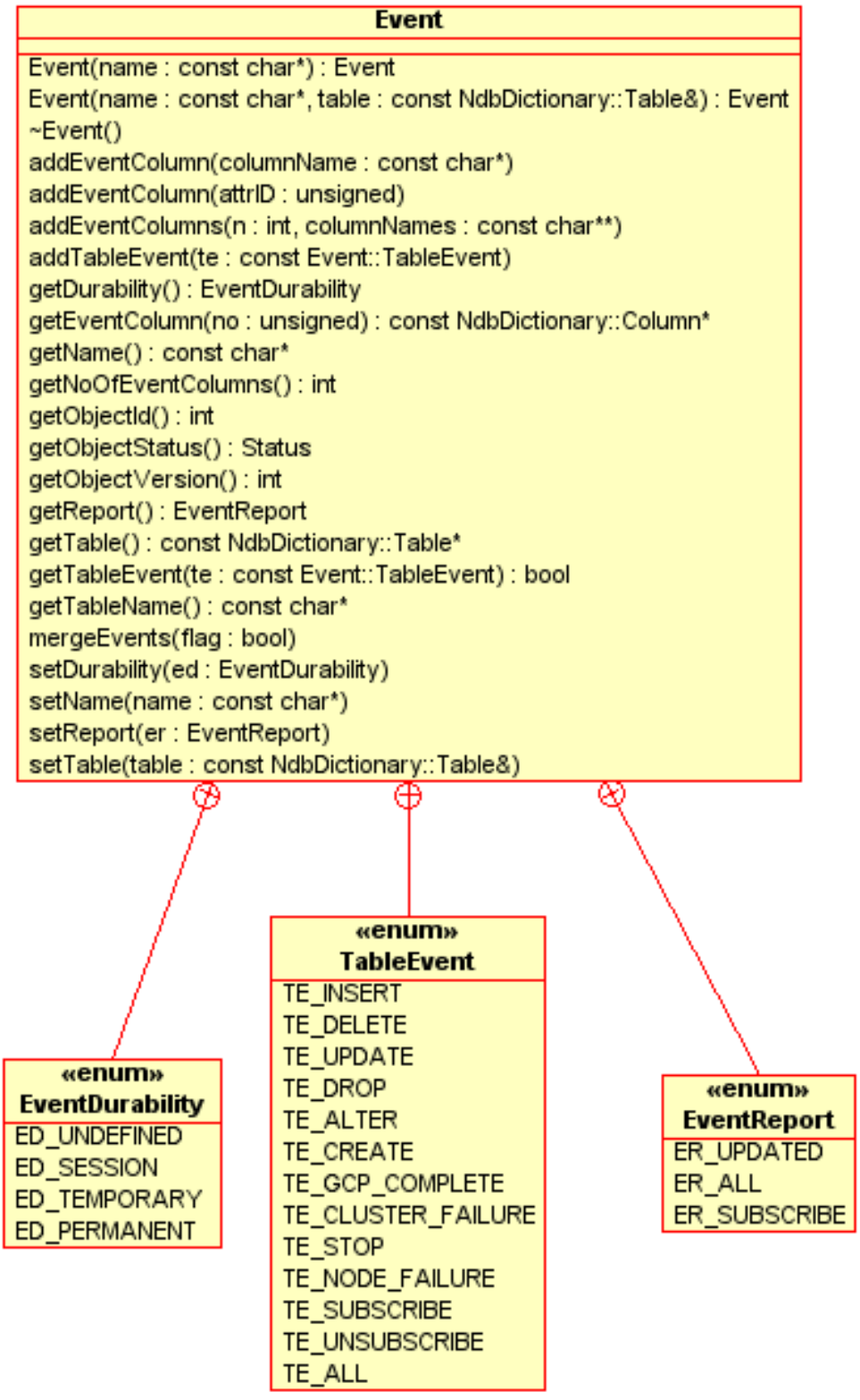
Types. These are the public types of the `Event` class:

Type	Purpose / Use
`TableEvent()`	Represents the type of a table event
`EventDurability()`	Specifies an event's scope, accessibility, and lifetime

Type	Purpose / Use
`EventReport()`	Specifies the reporting option for a table event

Class diagram. This diagram shows all the available methods and enumerated types of the `Event` class:

Figure 2.5 NdbDictionary::Object::Event



2.3.6.1 Event::addEventColumn()

Description. This method is used to add a column on which events should be detected. The column may be indicated either by its ID or its name.

Important

You must invoke `Dictionary::createEvent()` before any errors will be detected. See Section 2.3.4.4, "Dictionary::createEvent()".

Note

If you know several columns by name, you can enable event detection on all of them at one time by using `addEventColumns()`. See Section 2.3.6.2, "Event::addEventColumns()".

Signature. Identifying the event using its column ID:

```
void addEventColumn
    (
      unsigned attrId
    )
```

Identifying the column by name:

```
void addEventColumn
    (
      const char* columnName
    )
```

Parameters. This method takes a single argument, which may be either one of the following:

- The column ID (`attrId`), which should be an integer greater than or equal to `0`, and less than the value returned by `getNoOfEventColumns()`.
- The column's *`name`* (as a constant character pointer).

Return value. *None*.

2.3.6.2 Event::addEventColumns()

Description. This method is used to enable event detection on several columns at the same time. You must use the names of the columns.

Important

As with `addEventColumn()`, you must invoke `Dictionary::createEvent()` before any errors will be detected. See Section 2.3.4.4, "Dictionary::createEvent()".

Signature.

```
void addEventColumns
    (
      int          n,
      const char** columnNames
    )
```

Parameters. This method requires two arguments, listed here:

- The number of columns *`n`* (an integer).
- The names of the columns *`columnNames`*—this must be passed as a pointer to a character pointer.

Return value. *None*.

2.3.6.3 Event::addTableEvent()

Description. This method is used to add types of events that should be detected.

Signature.

```
void addTableEvent
    (
      const TableEvent te
    )
```

Parameters. This method requires a `TableEvent` value.

Return value. *None*.

2.3.6.4 Event Constructor

Description. The `Event` constructor creates a new instance with a given name, and optionally associated with a table.

You should keep in mind that the NDB API does not track allocated event objects, which means that the user must explicitly delete the `Event` thus created after it is no longer in use.

Signatures. It is possible to invoke this method in either of two ways, the first of these being by name only, as shown here:

```
Event
    (
      const char* name
    )
```

Alternatively, you can use the event name and an associated table, like this:

```
Event
    (
      const char*                  name,
      const NdbDictionary::Table& table
    )
```

Parameters. At a minimum, a *`name`* (as a constant character pointer) for the event is required. Optionally, an event may also be associated with a table; this argument, when present, is a reference to a `Table` object (see Section 2.3.37, “The Table Class”).

Return value. A new instance of `Event`.

Destructor. A destructor for this class is supplied as a virtual method which takes no arguments and whose return type is `void`.

2.3.6.5 Event::EventDurability

This section discusses `EventDurability`, a type defined by the `Event` class.

Description. The values of this type are used to describe an event's lifetime or persistence as well as its scope.

Enumeration values. Possible values are shown, along with descriptions, in the following table:

Value	Description
`ED_UNDEFINED`	The event is undefined or of an unsupported type.

Value	Description
ED_SESSION	This event persists only for the duration of the current session, and is available only to the current application. It is deleted after the application disconnects or following a cluster restart. **Important** The value ED_SESSION is reserved for future use and is not yet supported in any NDB Cluster release.
ED_TEMPORARY	Any application may use the event, but it is deleted following a cluster restart. **Important** The value ED_TEMPORARY is reserved for future use and is not yet supported in any NDB Cluster release.
ED_PERMANENT	Any application may use the event, and it persists until deleted by an application—even following a cluster. restart **Important** The value ED_PERMANENT is reserved for future use and is not yet supported in any NDB Cluster release.

2.3.6.6 Event::EventReport

This section discusses `EventReport`, a type defined by the `Event` class.

Description. The values of this type are used to specify reporting options for table events.

Enumeration values. Possible values are shown, along with descriptions, in the following table:

Value	Description
ER_UPDATED	Reporting of update events
ER_ALL	Reporting of all events, except for those not resulting in any updates to the inline parts of BLOB columns
ER_SUBSCRIBE	Reporting of subscription events
ER_DDL	Reporting of DDL events (see Section 2.3.6.20, “Event::setReport()”, for more information)

2.3.6.7 Event::getDurability()

Description. This method gets the event's lifetime and scope (that is, its `EventDurability`).

Signature.

```
EventDurability getDurability
    (
      void
    ) const
```

Parameters. *None*.

Return value. An `EventDurability` value.

2.3.6.8 Event::getEventColumn()

Description. This method is used to obtain a specific column from among those on which an event is defined.

Signature.

```
const Column* getEventColumn
    (
      unsigned no
    ) const
```

Parameters. The number (*no*) of the column, as obtained using `getNoOfColumns()` (see Section 2.3.6.10, “Event::getNoOfEventColumns()”).

Return value. A pointer to the `Column` corresponding to *no*.

2.3.6.9 Event::getName()

Description. This method obtains the name of the event.

Signature.

```
const char* getName
    (
      void
    ) const
```

Parameters. *None*.

Return value. The name of the event, as a character pointer.

2.3.6.10 Event::getNoOfEventColumns()

Description. This method obtains the number of columns on which an event is defined.

Signature.

```
int getNoOfEventColumns
    (
      void
    ) const
```

Parameters. *None*.

Return value. The number of columns (as an integer), or `-1` in the case of an error.

2.3.6.11 Event::getObjectStatus()

Description. This method gets the object status of the event.

Signature.

```
virtual Object::Status getObjectStatus
    (
      void
    ) const
```

Parameters. *None*.

Return value. The object status of the event. For possible values, see Section 2.3.31.4, “Object::Status”.

2.3.6.12 Event::getObjectVersion()

Description. This method gets the event's object version (see NDB Schema Object Versions).

Signature.

```
virtual int getObjectVersion
    (
      void
    ) const
```

Parameters. *None*.

Return value. The object version of the event, as an integer.

2.3.6.13 Event::getObjectId()

Description. This method retrieves an event's object ID.

Signature.

```
virtual int getObjectId
    (
      void
    ) const
```

Parameters. *None*.

Return value. The object ID of the event, as an integer.

2.3.6.14 Event::getReport()

Description. This method is used to obtain the reporting option in force for this event.

Signature.

```
EventReport getReport
    (
      void
    ) const
```

Parameters. *None*.

Return value. One of the reporting options specified in Section 2.3.6.6, “Event::EventReport”.

2.3.6.15 Event::getTable()

Description. This method is used to find the table with which an event is associated. It returns a reference to the corresponding `Table` object. You may also obtain the name of the table directly using `getTableName()`.

Signature.

```
const NdbDictionary::Table* getTable
    (
      void
    ) const
```

Parameters. *None*.

Return value. The table with which the event is associated—if there is one—as a pointer to a `Table` object; otherwise, this method returns `NULL`. (See Section 2.3.37, “The Table Class”.)

2.3.6.16 Event::getTableEvent()

Description. This method is used to check whether a given table event will be detected.

Signature.

```
bool getTableEvent
    (
      const TableEvent te
    ) const
```

Parameters. This method takes a single parameter, the table event's type—that is, a `TableEvent` value.

Return value. This method returns `true` if events of `TableEvent` type *`te`* will be detected. Otherwise, the return value is `false`.

2.3.6.17 Event::getTableName()

Description. This method obtains the name of the table with which an event is associated, and can serve as a convenient alternative to `getTable()`. (See Section 2.3.6.15, "Event::getTable()".)

Signature.

```
const char* getTableName
    (
      void
    ) const
```

Parameters. *None*.

Return value. The name of the table associated with this event, as a character pointer.

2.3.6.18 Event::mergeEvents()

Description. This method is used to set the *merge events flag*, which is `false` by default. Setting it to `true` implies that events are merged as follows:

- For a given `NdbEventOperation` associated with this event, events on the same primary key within the same global checkpoint index (GCI) are merged into a single event.
- A blob table event is created for each blob attribute, and blob events are handled as part of main table events.
- Blob post/pre data from blob part events can be read via `NdbBlob` methods as a single value.

Note

Currently this flag is not inherited by `NdbEventOperation`, and must be set on `NdbEventOperation` explicitly. See Section 2.3.21, "The NdbEventOperation Class".

Signature.

```
void mergeEvents
    (
      bool flag
    )
```

Parameters. A Boolean *`flag`* value.

Return value. *None*.

2.3.6.19 Event::setDurability()

Description. This method sets an event's durability—that is, its lifetime and scope.

Signature.

```
void setDurability(EventDurability ed)
```

Parameters. This method requires a single `EventDurability` value as a parameter.

Return value. *None*.

2.3.6.20 Event::setReport()

Description. This method is used to set a reporting option for an event. Possible option values may be found in Section 2.3.6.6, “Event::EventReport”.

Reporting of DDL events. You must call `setReport()` using the `EventReport` value `ER_DDL` (added in the same NDB Cluster versions).

For example, to enable DDL event reporting on an `Event` object named `myEvent`, you must invoke this method as shown here:

```
myEvent.setReport(NdbDictionary::Event::ER_DDL);
```

Signature.

```
void setReport
    (
      EventReport er
    )
```

Parameters. An `EventReport` option value.

Return value. *None*.

2.3.6.21 Event::setName()

Description. This method is used to set the name of an event. The name must be unique among all events visible from the current application (see Section 2.3.6.7, “Event::getDurability()”).

Note

You can also set the event's name when first creating it. See Section 2.3.6.4, “Event Constructor”.

Signature.

```
void setName
    (
      const char* name
    )
```

Parameters. The *`name`* to be given to the event (as a constant character pointer).

Return value. *None*.

2.3.6.22 Event::setTable()

Description. This method defines a table on which events are to be detected.

Note

By default, event detection takes place on all columns in the table. Use `addEventColumn()` to override this behavior. For details, see Section 2.3.6.1, “Event::addEventColumn()”.

Signature.

```
void setTable
    (
      const NdbDictionary::Table& table
    )
```

NDB 7.2.14, NDB 7.3.3, and later NDB Cluster releases support the use of a pointer with this method, as shown here:

```
void setTable
    (
      const NdbDictionary::Table*; table
    )
```

When so used, this version of `setTable()` returns -1 if the table pointer is `NULL`. (Bug #16329082)

Parameters. This method requires a single parameter, a reference to the table (see Section 2.3.37, "The Table Class") on which events are to be detected. *NDB 7.2.14, NDB 7.3.3, and later*: A reference or a pointer to the table can be used.

Return value. *None*. *NDB 7.2.14, NDB 7.3.3, and later*: -1, if a null table pointer is used.

2.3.6.23 Event::TableEvent

This section describes `TableEvent`, a type defined by the `Event` class.

Description. `TableEvent` is used to classify the types of events that may be associated with tables in the NDB API.

Enumeration values. Possible values are shown, along with descriptions, in the following table:

Value	Description
`TE_INSERT`	Insert event on a table
`TE_DELETE`	Delete event on a table
`TE_UPDATE`	Update event on a table
`TE_DROP`	Occurs when a table is dropped
`TE_ALTER`	Occurs when a table definition is changed
`TE_CREATE`	Occurs when a table is created
`TE_GCP_COMPLETE`	Occurs on the completion of a global checkpoint
`TE_CLUSTER_FAILURE`	Occurs on Cluster failures
`TE_STOP`	Occurs when an event operation is stopped
`TE_NODE_FAILURE`	Occurs when a Cluster node fails
`TE_SUBSCRIBE`	Occurs when a cluster node subscribes to an event
`TE_UNSUBSCRIBE`	Occurs when a cluster node unsubscribes from an event
`TE_EMPTY`	Empty epoch received from data nodes
`TE_INCONSISTENT`	Missing data or buffer overflow at data node
`TE_OUT_OF_MEMORY`	Overflow in event buffer
`TE_ALL`	Occurs when any event occurs on a table (not relevant when a specific event is received)

`TE_EMPTY`, `TE_INCONSISTENT`, and `TE_OUT_OF_MEMORY` were added in NDB 7.4.3.

2.3.7 The EventBufferMemoryUsage Structure

This section describes the `EventBufferMemoryUsage` structure.

Parent class. `Ndb`

Description. This structure was added in NDB 7.4.3 for working with event buffer memory usage statistics. It is used as an argument to `Ndb::get_event_buffer_memory_usage()`.

Attributes. `EventBufferMemoryUsage` has the attributes shown in the following table:

Attribute	Type	Initial Value	Description
`allocated_bytes`	`unsigned`	*none*	The total event buffer memory allocated, in bytes
`used_bytes`	`unsigned`	*none*	The total memory used, in bytes
`usage_percent`	`unsigned`	*none*	Event buffer memory usage, as a percent (`100 * used_bytes / allocated_bytes`)

2.3.8 The ForeignKey Class

This class represents a foreign key on an `NDB` table. It was added to the NDB API in NDB Cluster 7.3.

Parent class. `Object`

Child classes. *None*.

Methods. The following table lists the public methods of the `ForeignKey` class and the purpose or use of each method:

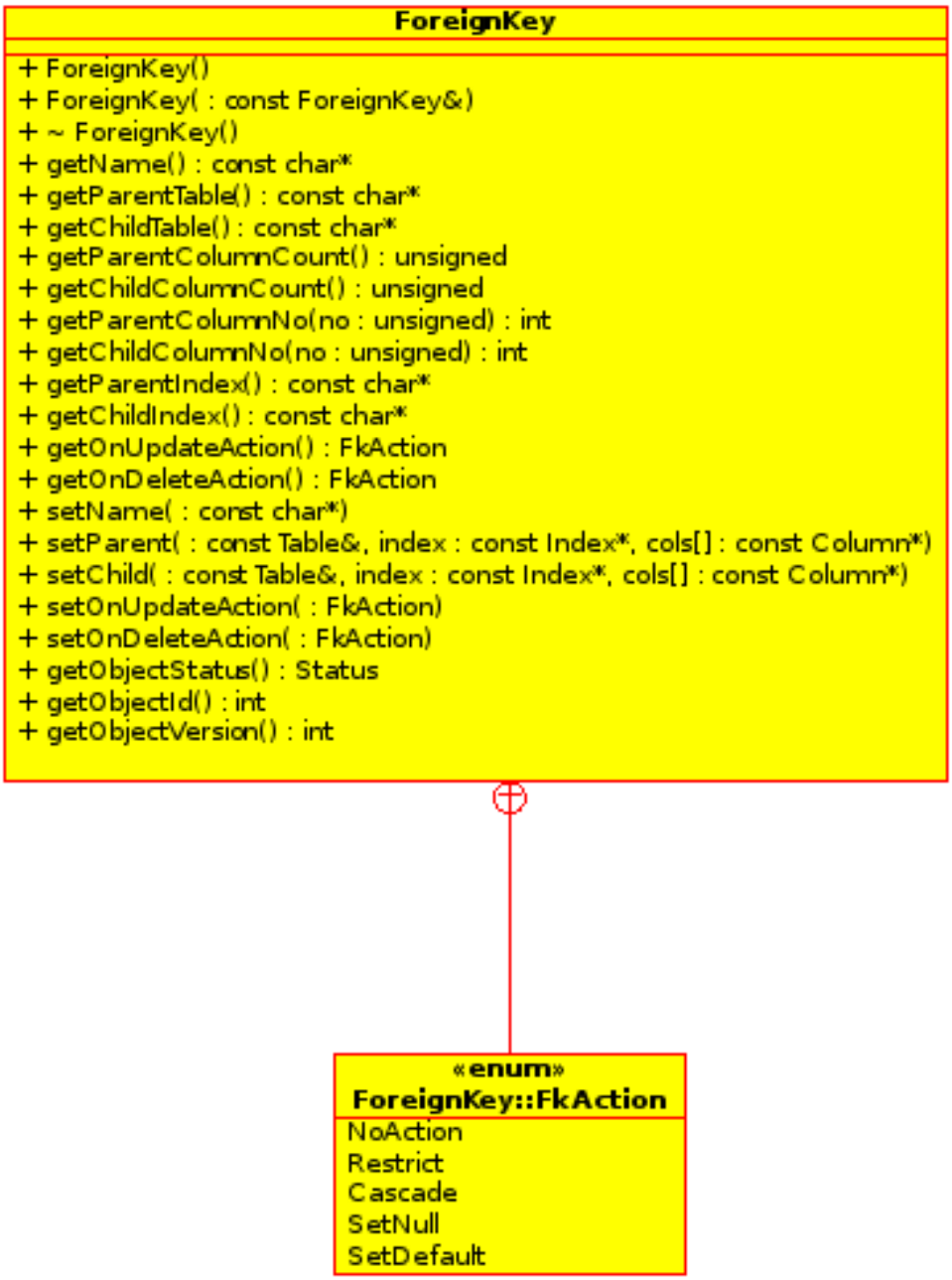
Method	Purpose / Use
`ForeignKey()`	Class constructor
`~ForeignKey()`	Class destructor
`getName()`	Get the foreign key's name
`getParentTable()`	Get the foreign key's parent table
`getChildTable()`	Get the foreign key's child table
`getParentColumnCount()`	Get the number of columns in the parent table
`getChildColumnCount()`	Get the number of columns in the child table
`getParentColumnNo()`	Get the column number in the parent table
`getChildColumnNo()`	Get the column number in the child table
`getParentIndex()`	Returns 0 if key points to parent table's primary key
`getChildIndex()`	Returns 0 if child references resolved using child table's primary key
`getOnUpdateAction()`	Get the foreign's key update action (`FkAction`)
`getOnDeleteAction()`	Get the foreign key's delete action (`FkAction`)
`setName()`	Set the foreign key's name
`setParent()`	Set the foreign key's parent table
`setChild()`	Set a foreign key's child table
`setOnUpdateAction()`	Set the foreign's key update action (`FkAction`)

Method	Purpose / Use
`setOnDeleteAction()`	Set the foreign key's delete action (`FkAction`)
`getObjectStatus()`	Get the object status
`getObjectId()`	Get the object ID
`getObjectVersion()`	Get the object version

Types. The `ForeignKey` class has one public type, the `FkAction` type.

This diagram shows all public members of the `ForeignKey` class:

Figure 2.6 NdbDictionary::Object::ForeignKey



2.3.8.1 ForeignKey()

Description. Create either an entirely new foreign key reference, or a copy of an existing one.

Signature. New instance:

```
ForeignKey
    (
      void
    )
```

Copy constructor:

```
ForeignKey
    (
      const ForeignKey&
    )
```

Parameters. For a new instance: *None*.

For the copy constructor: A reference to an existing instance of `ForeignKey`.

Return value. A new instance of `ForeignKey`.

2.3.8.2 ForeignKey::FkAction

`FkAction` is an enumeration which represents a reference action for a foreign key when an update or delete operation is performed on the parent table.

Enumeration values. Possible values are shown, along with the corresponding reference action, in the following table:

Value	Reference Action/Description
`NoAction`	`NO ACTION`: Deferred check.
`Restrict`	`RESTRICT`: Reject operation on parent table.
`Cascade`	`CASCADE`: Perform operation on row from parent table; perform same operation on matching rows in child table.
`SetNull`	`SET NULL`: Perform operation on row from parent table; set any matching foreign key columns in child table to `NULL`.
`SetDefault`	`SET DEFAULT`: Currently not supported in NDB Cluster.

See also Using FOREIGN KEY Constraints, in the *MySQL Manual*.

2.3.8.3 ForeignKey::getName()

Description. Retrieve the name of the `ForeignKey` instance for which the method is invoked.

Signature.

```
const char* getName
    (
      void
    ) const
```

Parameters. *None*.

Return value. The name of the `ForeignKey`.

2.3.8.4 ForeignKey::getParentTable()

Description. Retrieve the parent table of the `ForeignKey` instance for which the method is invoked.

Signature.

```
const char* getParentTable
      (
        void
      ) const
```

Parameters. *None*.

Return value. A pointer to the parent table of the `ForeignKey`.

2.3.8.5 ForeignKey::getChildTable()

Description. Retrieve the child table of the `ForeignKey` instance for which the method is invoked.

Signature.

```
const char* getChildTable
      (
        void
      ) const
```

Parameters. *None*.

Return value. A pointer to the child table of this `ForeignKey`.

2.3.8.6 ForeignKey::getParentColumnCount()

Description. Retrieve the number of columns in the parent table of this `ForeignKey`.

Signature.

```
unsigned getParentColumnCount
      (
        void
      ) const
```

Parameters. *None*.

Return value. The number of columns in the parent table.

2.3.8.7 ForeignKey::getChildColumnCount()

Description. Retrieve the number of columns in the child table of this `ForeignKey`.

Signature.

```
unsigned getChildColumnCount
      (
        void
      ) const
```

Parameters. *None*.

Return value. The number of columns in the child table.

2.3.8.8 ForeignKey::getParentIndex()

Description. Returns 0 if the child table refers to the parent table's primary key.

Signature.

```
const char* getParentIndex
    (
      void
```

```
    ) const
```

Parameters. *None*.

Return value. See description.

2.3.8.9 ForeignKey::getChildIndex()

Description. Return 0 if child references are resolved using the child table's primary key.

Signature.

```
const char* getChildIndex
    (
      void
    ) const
```

Parameters. *None*.

Return value. See description.

2.3.8.10 ForeignKey::getParentColumnNo()

Description. This method gets the sequence number of a foreign key column in the parent table for a given index. See the documentation for `Column::getColumnNo()` for information about handling columns in the NDB API.

Signature.

```
int getParentColumnNo
    (
      unsigned no
    ) const
```

Parameters. *None*.

Return value. The sequence number of the column.

2.3.8.11 ForeignKey::getChildColumnNo()

Description. This method gets the sequence number of a foreign key column in the child table for a given index. See the documentation for `Column::getColumnNo()` for information about handling columns in the NDB API.

Signature.

```
int getChildColumnNo
    (
      unsigned no
    ) const
```

Parameters. *None*.

Return value. The sequence number of the column.

2.3.8.12 ForeignKey::getOnUpdateAction()

Description. Get the foreign key's `ON UPDATE` action. This is a `ForeignKey::FkAction` and has one of the values `NoAction`, `Restrict`, `Cascade`, or `SetNull`.

Signature.

```
FkAction getOnUpdateAction
```

```
    (
      void
    ) const
```

Parameters. *None*.

Return value. The sequence number of the column.

2.3.8.13 ForeignKey::getOnDeleteAction()

Description. Get the foreign key's `ON DELETE` action. This is a `ForeignKey::FkAction` and has one of the values `NoAction`, `Restrict`, `Cascade`, or `SetNull`.

Signature.

```
FkAction getOnDeleteAction
    (
      void
    ) const
```

Parameters. *None*.

Return value. The sequence number of the column.

2.3.8.14 ForeignKey::setName()

Description. Set the name of the `ForeignKey` instance for which the method is invoked.

Signature.

```
void setName
    (
      const char*
    )
```

Parameters. The name of the `ForeignKey`.

Return value. *None*.

2.3.8.15 ForeignKey::setParent()

Description. Set the parent table of a `ForeignKey`, given a reference to the table, and optionally, an index to use as the foreign key.

Signature.

```
void setParent
    (
      const Table&,
      const Index* index = 0,
      const Column* cols[] = 0
    )
```

Parameters. A reference to a `Table`. Optionally, an index using the indicated column or columns.

Return value. *None*.

2.3.8.16 ForeignKey::setChild()

Description. Set the child table of a `ForeignKey`, given a reference to the table, and optionally, an index to use as the foreign key.

Signature.

```
void setChild
    (
      const Table&,
      const Index* index = 0,
      const Column* cols[] = 0
    )
```

Parameters. A reference to a `Table`. Optionally, an index using the indicated column or columns.

Return value. *None*.

2.3.8.17 ForeignKey::setOnUpdateAction()

Description. Set the foreign key's `ON UPDATE` action.

Signature.

```
void setOnUpdateAction
  (
    FkAction
  )
```

Parameters. The `ON UPDATE` action to be performed. This must be a `ForeignKey::FkAction` having one of the values `NoAction`, `Restrict`, `Cascade`, or `SetNull`.

Return value. *None*

2.3.8.18 ForeignKey::setOnDeleteAction()

Description. Set the foreign key's `ON DELETE` action.

Signature.

```
void setOnUpdateAction
  (
    FkAction
  )
```

Parameters. The `ON UPDATE` action to be performed, of type `ForeignKey::FkAction`. Must be one of the values `NoAction`, `Restrict`, `Cascade`, or `SetNull`.

Return value. *None*

2.3.8.19 ForeignKey::getObjectStatus()

Description. Get the object status (see Section 2.3.31.4, “Object::Status”) for this `ForeignKey` object.

Signature.

```
virtual Object::Status getObjectStatus
    (
      void
    ) const
```

Parameters. *None*.

Return value. The `ForeignKey` object's status, as a value of type `Object::Status`. See this type's documentation for possible values and their interpretation.

2.3.8.20 ForeignKey::getObjectId()

Description. Get the object ID (see Section 2.3.31.7, “Object::getObjectId()”) for this `ForeignKey` object.

Signature.

```
virtual int getObjectId
    (
      void
    ) const
```

Parameters. *None*.

Return value. The `ForeignKey` object's ID, as returned by `Object::getObjectId()`.

2.3.8.21 ForeignKey::getObjectVersion()

Description. Get the object version (see Section 2.3.31.9, “Object::getObjectVersion()”) for this `ForeignKey` object.

Signature.

```
virtual int getObjectVersion
    (
      void
    ) const
```

Parameters. *None*.

Return value. The `ForeignKey` object's version number (an integer), as returned by `Object::getObjectVersion()`.

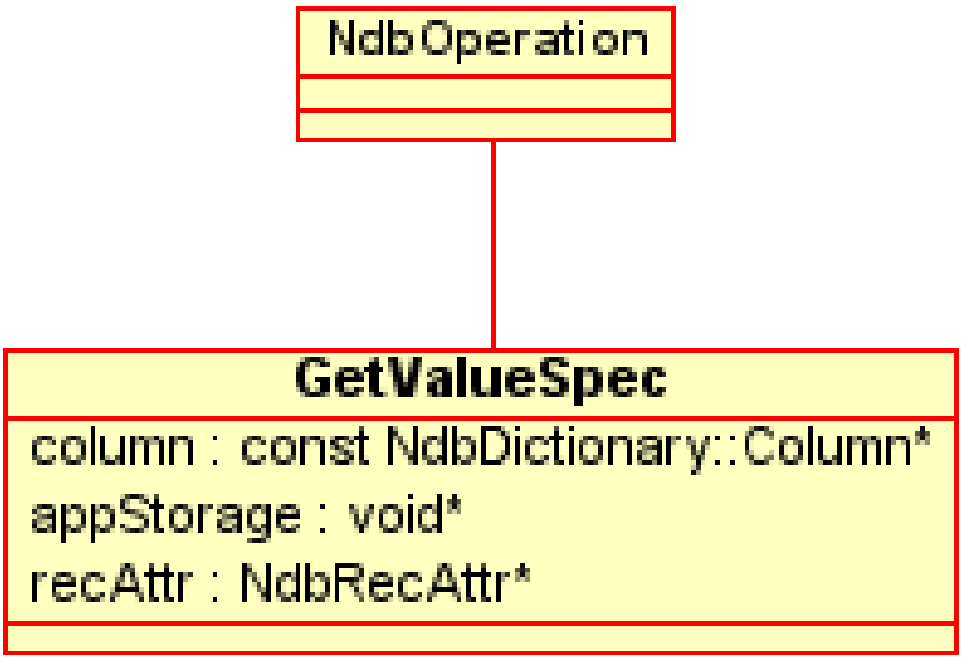
2.3.9 The GetValueSpec Structure

Parent class. `NdbOperation`

Description. This structure is used to specify an extra value to obtain as part of an `NdbRecord` operation.

Members. The makeup of this structure is shown here:

Figure 2.7 NdbOperation::GetValueSpec



The individual members are described in more detail in the following table:

Name	Type	Description
`column`	`const Column*`	To specify an extra value to read, the caller must provide this, as well as (optionally `NULL`) *`appStorage`* pointer.
`appStorage`	`void*`	If this pointer is null, then the received value is stored in memory managed by the `NdbRecAttr` object. Otherwise, the received value is stored at the location pointed to (and is still accessable using the `NdbRecAttr` object). **Important** It is the caller's responsibility to ensure that the following conditions are met: 1. *`appStorage`* points to sufficient space to store any returned data. 2. Memory pointed to by *`appStorage`* is not reused or freed until after the `execute()` call returns.
`recAttr`	`NdbRecAttr*`	After the operation is defined, *`recAttr`* contains a pointer to the `NdbRecAttr` object for receiving the data.

Important

Currently, blob reads cannot be specified using `GetValueSpec`.

For more information, see Section 2.3.27, “The NdbRecord Interface”.

2.3.10 The HashMap Class

This class represents a hash map in an NDB Cluster. It was added to the NDB API in NDB 7.2.7.

Parent class. `Object`

Child classes. *None*.

Methods. The following table lists the public methods of the `HashMap` class and the purpose or use of each method:

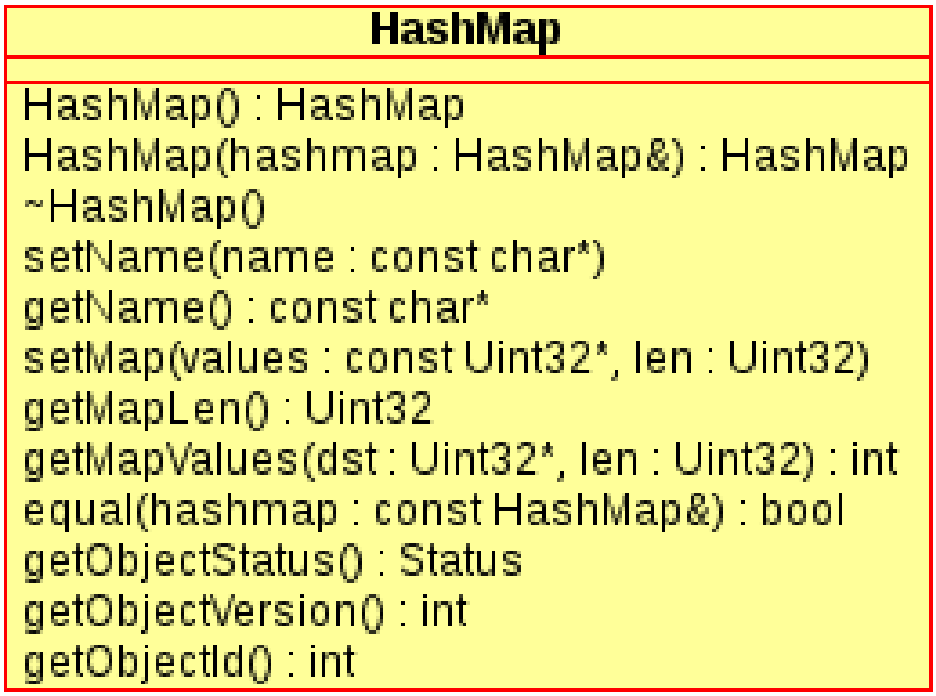
Method	Purpose / Use
`HashMap()`	Class constructor
`~HashMap()`	Class destructor
`setName()`	Set a name for the hashmap
`getName()`	Gets a hashmap's name
`setMap()`	Sets a hashmap's length and values

Method	Purpose / Use
`getMapLen()`	Gets a hashmap's length
`getMapValues()`	Gets the values contained in the hashmap
`equal()`	Compares this hashmap's values with those of another hashmap
`getObjectStatus()`	Gets the hashmap's object status
`getObjectVersion()`	Gets the hashmap's schema object version
`getObjectId()`	Gets the hashmap's ID

Types. The `HashMap` class has no public types.

This diagram shows all public members of the `HashMap` class:

Figure 2.8 NdbDictionary::Object::HashMap



2.3.10.1 HashMap Constructor

Description. The `HashMap` class constructor normally requires no arguments. A copy constructor is also available.

See also Section 2.3.4.6, “Dictionary::createHashMap()”, for more information.

Signature. Base constructor:

```
HashMap HashMap
    (
      void
    )
```

Copy constructor:

```
HashMap HashMap
    (
      const HashMap& hashmap
```

```
    )
```

Destructor:

```
virtual ~HashMap
    (
      void
    )
```

Parameters. *None*, or the address of an existing `HashMap` object to be copied.

Return value. A new instance of `HashMap`, possibly a copy of an existing one.

2.3.10.2 HashMap::setName()

Description. Sets the name of the hash map.

Signature.

```
void setName
    (
      const char* name
    )
```

Parameters. The name to be assigned to the hashmap.

Return value. *None*.

2.3.10.3 HashMap::getName()

Description. Gets the name of the hash map.

Signature.

```
const char* getName
    (
      void
    ) const
```

Parameters. *None*.

Return value. The name of the hash map.

2.3.10.4 HashMap::setMap()

Description. Assigns a set of values to a has map.

Signature.

```
void setMap
    (
      const Uint32* values,
      Uint32 len
    )
```

Parameters. A pointer to a set of *`values`* of length *`len`*.

Return value. *None*.

2.3.10.5 HashMap::getMapLen()

Description. Gets the hash map's length; that is, the number of values which it contains. You can obtain the values using `getMapValues()`.

Signature.

```
Uint32 getMapLen
    (
      void
    ) const
```

Parameters. *None*.

Return value. The length of the hash map.

2.3.10.6 HashMap::getMapValues()

Description. Gets the values listed in the hash map.

Signature.

```
int getMapValues
    (
      Uint32* dst,
      Uint32 len
    ) const
```

Parameters. A pointer to a set of values (*`dst`*) and the number of values (*`len`*).

Return value. Returns 0 on success; on failure, returns -1 and sets error.

2.3.10.7 HashMap::equal()

Description. Compares (only) the values of this `HashMap` with those of another one.

Signature.

```
bool equal
    (
      const HashMap& hashmap
    ) const
```

Parameters. A reference to the hash map to be compared with this one.

Return value. *None*.

2.3.10.8 HashMap::getObjectStatus()

Description. This method retrieves the status of the `HashMap` for which it is invoked. The return value is of type `Object::Status`.

Signature.

```
virtual Status getObjectStatus
    (
      void
    ) const
```

Parameters. *None*.

Return value. Returns the current `Status` of the `HashMap`.

2.3.10.9 HashMap::getObjectVersion()

Description. The method gets the hash map's schema object version.

Signature.

```
virtual int getObjectVersion
    (
      void
    ) const
```

Parameters. *None*.

Return value. The object's version number, an integer.

2.3.10.10 HashMap::getObjectId()

Description. This method retrieves the hash map's ID.

Signature.

```
virtual int getObjectId
    (
      void
    ) const
```

Parameters. *None*.

Return value. The object ID, an integer.

2.3.11 The Index Class

This section provides a reference to the `Index` class and its public members.

Parent class. `NdbDictionary`

Child classes. *None*

Description. This class represents an index on an `NDB Cluster` table column. It is a descendant of the `NdbDictionary` class, using the `Object` class.

Methods. The following table lists the public methods of `Index` and the purpose or use of each method:

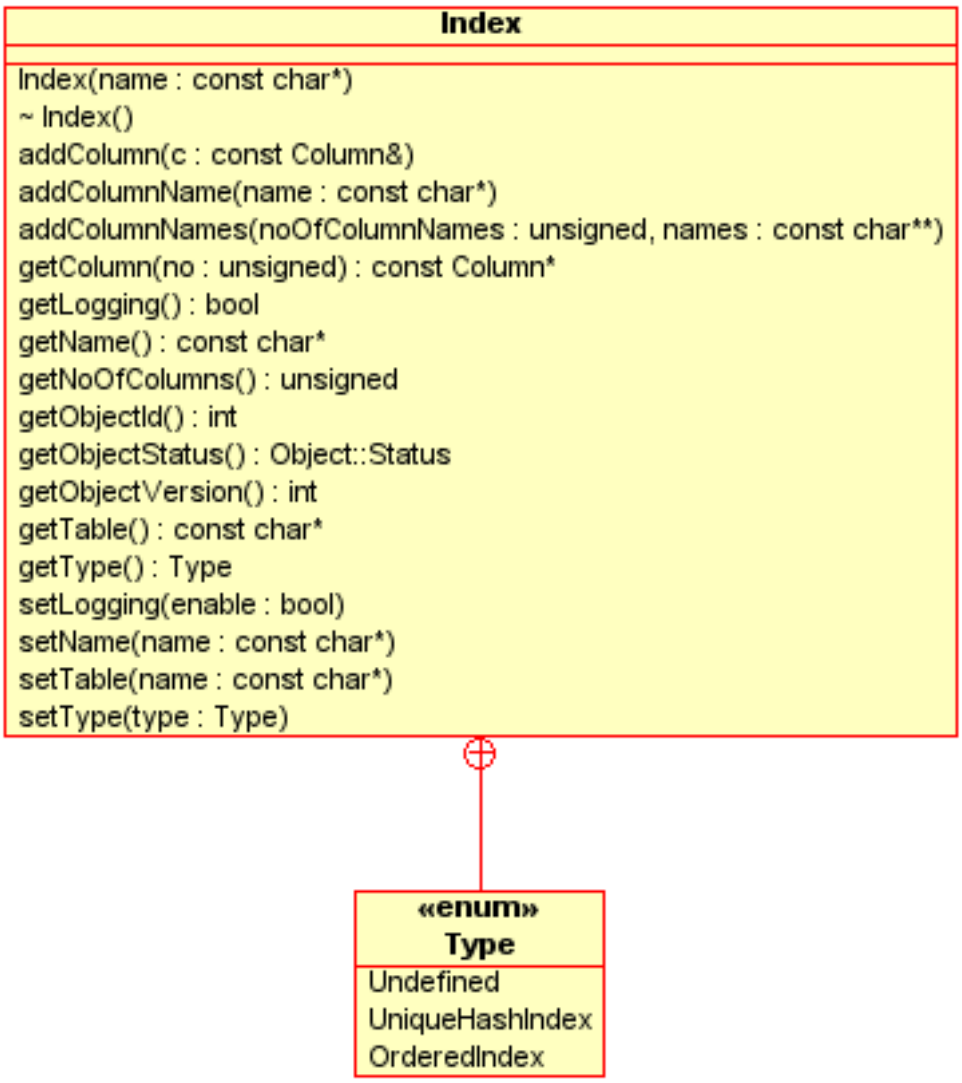
Method	Purpose / Use
`Index()`	Class constructor
`~Index()`	Destructor
`addColumn()`	Adds a `Column` object to the index
`addColumnName()`	Adds a column by name to the index
`addColumnNames()`	Adds multiple columns by name to the index
`getColumn()`	Gets a column making up (part of) the index
`getLogging()`	Checks whether the index is logged to disk
`getName()`	Gets the name of the index
`getNoOfColumns()`	Gets the number of columns belonging to the index
`getObjectStatus()`	Gets the index object status
`getObjectVersion()`	Gets the index object status

Method	Purpose / Use
`getObjectId()`	Gets the index object ID
`getTable()`	Gets the name of the table being indexed
`getType()`	Gets the index type
`setLogging()`	Enable/disable logging of the index to disk
`setName()`	Sets the name of the index
`setTable()`	Sets the name of the table to be indexed
`setType()`	Set the index type

Types. Index has one public type, the `Type` type.

Class diagram. This diagram shows all the available methods and enumerated types of the `Index` class:

Figure 2.9 NdbDictionary::Object::Index



Important

If you create or change indexes using the `NDB` API, these modifications cannot be seen by MySQL. The only exception to this is renaming the index using `Index::setName()`.

2.3.11.1 Index Class Constructor

Description. This is used to create an new instance of `Index`.

Important

Indexes created using the NDB API cannot be seen by the MySQL Server.

Signature.

```
Index
    (
      const char* name = ""
    )
```

Parameters. The name of the new index. It is possible to create an index without a name, and then assign a name to it later using `setName()`. See Section 2.3.11.15, “Index::setName()”.

Return value. A new instance of `Index`.

Destructor. The destructor (`~Index()`) is supplied as a virtual method.

2.3.11.2 Index::addColumn()

Description. This method may be used to add a column to an index.

Note

The order of the columns matches the order in which they are added to the index. However, this matters only with ordered indexes.

Signature.

```
void addColumn
    (
      const Column& c
    )
```

Parameters. A reference *c* to the column which is to be added to the index.

Return value. *None*.

2.3.11.3 Index::addColumnName()

Description. This method works in the same way as `addColumn()`, except that it takes the name of the column as a parameter. See Section 2.3.11.5, “Index::getColumn()”.

Signature.

```
void addColumnName
    (
      const char* name
    )
```

Parameters. The *name* of the column to be added to the index, as a constant character pointer.

Return value. *None*.

2.3.11.4 Index::addColumnNames()

Description. This method is used to add several column names to an index definition at one time.

Note

As with the `addColumn()` and `addColumnName()` methods, the indexes are numbered in the order in which they were added. (However, this matters only for ordered indexes.)

Signature.

```
void addColumnNames
    (
      unsigned     noOfNames,
      const char** names
    )
```

Parameters. This method takes two parameters, listed here:

- The number of columns and names *`noOfNames`* to be added to the index.
- The *`names`* to be added (as a pointer to a pointer).

Return value. *None*.

2.3.11.5 Index::getColumn()

Description. This method retrieves the column at the specified position within the index.

Signature.

```
const Column* getColumn
    (
      unsigned no
    ) const
```

Parameters. The ordinal position number *`no`* of the column, as an unsigned integer. Use the `getNoOfColumns()` method to determine how many columns make up the index—see Section 2.3.11.8, “Index::getNoOfColumns()”, for details.

Return value. The column having position *`no`* in the index, as a pointer to an instance of `Column`. See Section 2.3.2, “The Column Class”.

2.3.11.6 Index::getLogging()

Description. Use this method to determine whether logging to disk has been enabled for the index.

Note

Indexes which are not logged are rebuilt when the cluster is started or restarted.

Ordered indexes currently do not support logging to disk; they are rebuilt each time the cluster is started. (This includes restarts.)

Signature.

```
bool getLogging
    (
      void
    ) const
```

Parameters. *None*.

Return value. A Boolean value:

- `true`: The index is being logged to disk.
- `false`: The index is not being logged.

2.3.11.7 Index::getName()

Description. This method is used to obtain the name of an index.

Signature.

```
const char* getName
    (
      void
    ) const
```

Parameters. *None*.

Return value. The name of the index, as a constant character pointer.

2.3.11.8 Index::getNoOfColumns()

Description. This method is used to obtain the number of columns making up the index.

Signature.

```
unsigned getNoOfColumns
    (
      void
    ) const
```

Parameters. *None*.

Return value. An unsigned integer representing the number of columns in the index.

2.3.11.9 Index::getObjectStatus()

Description. This method gets the object status of the index.

Signature.

```
virtual Object::Status getObjectStatus
    (
      void
    ) const
```

Parameters. *None*.

Return value. A `Status` value—see Section 2.3.31.4, “Object::Status”, for more information.

2.3.11.10 Index::getObjectVersion()

Description. This method gets the object version of the index (see NDB Schema Object Versions).

Signature.

```
virtual int getObjectVersion
```

```
    (
      void
    ) const
```

Parameters. *None*.

Return value. The object version for the index, as an integer.

2.3.11.11 Index::getObjectId()

Description. This method is used to obtain the object ID of the index.

Signature.

```
virtual int getObjectId
    (
      void
    ) const
```

Parameters. *None*.

Return value. The object ID, as an integer.

2.3.11.12 Index::getTable()

Description. This method can be used to obtain the name of the table to which the index belongs.

Signature.

```
const char* getTable
    (
      void
    ) const
```

Parameters. *None*.

Return value. The name of the table, as a constant character pointer.

2.3.11.13 Index::getType()

Description. This method can be used to find the type of index.

Signature.

```
Type getType
    (
      void
    ) const
```

Parameters. *None*.

Return value. An index type. See Section 2.3.11.18, “Index::Type”, for possible values.

2.3.11.14 Index::setLogging

Description. This method is used to enable or disable logging of the index to disk.

Signature.

```
void setLogging
```

```
    (
      bool enable
    )
```

Parameters. `setLogging()` takes a single Boolean parameter *`enable`*. If *`enable`* is `true`, then logging is enabled for the index; if false, then logging of this index is disabled.

Return value. *None*.

2.3.11.15 Index::setName()

Description. This method sets the name of the index.

Note

This is the only `Index::set*()` method whose result is visible to a MySQL Server.

Signature.

```
void setName
    (
      const char* name
    )
```

Parameters. The desired *`name`* for the index, as a constant character pointer.

Return value. *None*.

2.3.11.16 Index::setTable()

Description. This method sets the table that is to be indexed. The table is referenced by name.

Signature.

```
void setTable
    (
      const char* name
    )
```

Parameters. The *`name`* of the table to be indexed, as a constant character pointer.

Return value. *None*.

2.3.11.17 Index::setType()

Description. This method is used to set the index type.

Signature.

```
void setType
    (
      Type type
    )
```

Parameters. The *`type`* of index. For possible values, see Section 2.3.11.18, “Index::Type”.

Return value. *None*.

2.3.11.18 Index::Type

Description. This is an enumerated type which describes the sort of column index represented by a given instance of `Index`.

Caution

Do not confuse this enumerated type with `Object::Type`, or with `Column::Type`.

Enumeration values. Possible values are shown, along with descriptions, in the following table:

Value	Description
`Undefined`	Undefined object type (initial/default value)
`UniqueHashIndex`	Unique unordered hash index (only index type currently supported)
`OrderedIndex`	Nonunique, ordered index

2.3.12 The IndexBound Structure

Parent class. `NdbIndexScanOperation`

Description. `IndexBound` is a structure used to describe index scan bounds for `NdbRecord` scans.

Members. These are shown in the following table:

Name	Type	Description
`low_key`	`const char*`	Row containing lower bound for scan (or `NULL` for scan from the start).
`low_key_count`	`Uint32`	Number of columns in lower bound (for bounding by partial prefix).
`low_inclusive`	`bool`	True for `<=` relation, false for `<`.
`high_key`	`const char*`	Row containing upper bound for scan (or `NULL` for scan to the end).
`high_key_count`	`Uint32`	Number of columns in upper bound (for bounding by partial prefix).
`high_inclusive`	`bool`	True for `>=` relation, false for `>`.
`range_no`	`Uint32`	Value to identify this bound; may be read using the `get_range_no()` method (see Section 2.3.23.4, "NdbIndexScanOperation::get_range_no()"). This value must be less than 8192 (set to zero if it is not being used). For ordered scans, `range_no` must be strictly increasing for each range, or else the result set will not be sorted correctly.

For more information, see Section 2.3.27, "The NdbRecord Interface".

2.3.13 The LogfileGroup Class

This section discusses the `LogfileGroup` class, which represents an NDB Cluster Disk Data logfile group.

Parent class. `NdbDictionary`

Child classes. *None*

Description. This class represents an NDB Cluster Disk Data logfile group, which is used for storing Disk Data undofiles. For general information about logfile groups and undofiles, see NDB Cluster Disk Data Tables, in the MySQL Manual.

Note

Only unindexed column data can be stored on disk. Indexes and indexes columns are always stored in memory.

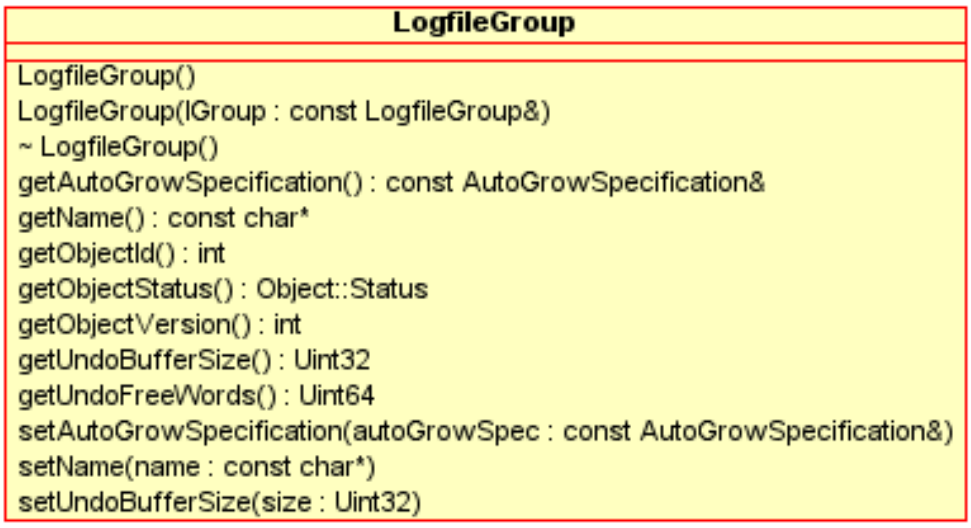
Methods. The following table lists the public methods of this class and the purpose or use of each method:

Method	Purpose / Use
`LogfileGroup()`	Class constructor
`~LogfileGroup()`	Virtual destructor
`getAutoGrowSpecification()`	Gets the logfile group's `AutoGrowSpecification` values
`getName()`	Retrieves the logfile group's name
`getObjectId()`	Get the object ID of the logfile group
`getObjectStatus()`	Gets the logfile group's object status value
`getObjectVersion()`	Retrieves the logfile group's object version
`getUndoBufferSize()`	Gets the size of the logfile group's `UNDO` buffer
`getUndoFreeWords()`	Retrieves the amount of free space in the `UNDO` buffer
`setAutoGrowSpecification()`	Sets `AutoGrowSpecification` values for the logfile group
`setName()`	Sets the name of the logfile group
`setUndoBufferSize()`	Sets the size of the logfile group's `UNDO` buffer.

Types. The `LogfileGroup` class does not itself define any public types. However, two of its methods make use of the `AutoGrowSpecification` data structure as a parameter or return value. For more information, see Section 2.3.1, “The AutoGrowSpecification Structure”.

Class diagram. This diagram shows all the available public methods of the `LogfileGroup` class:

Figure 2.10 NdbDictionary::Object::LogfileGroup



2.3.13.1 LogfileGroup Constructor

Description. The `LogfileGroup` class has two public constructors, one of which takes no arguments and creates a completely new instance. The other is a copy constructor.

Note

The `Dictionary` class also supplies methods for creating and destroying `LogfileGroup` objects. See Section 2.3.4, "The Dictionary Class".

Signatures. New instance:

```
LogfileGroup
    (
      void
    )
```

Copy constructor:

```
LogfileGroup
    (
      const LogfileGroup& logfileGroup
    )
```

Parameters. When creating a new instance, the constructor takes no parameters. When copying an existing instance, the constructor is passed a reference to the `LogfileGroup` instance to be copied.

Return value. A `LogfileGroup` object.

Destructor.

```
virtual ~LogfileGroup
    (
      void
    )
```

Examples.

```
[To be supplied...]
```

2.3.13.2 LogfileGroup::getAutoGrowSpecification()

Description. This method retrieves the `AutoGrowSpecification` associated with the logfile group.

Signature.

```
const AutoGrowSpecification& getAutoGrowSpecification
    (
      void
    ) const
```

Parameters. *None*.

Return value. An `AutoGrowSpecification` data structure. See Section 2.3.1, "The AutoGrowSpecification Structure", for details.

2.3.13.3 LogfileGroup::getName()

Description. This method gets the name of the logfile group.

Signature.

```
const char* getName
    (
      void
    ) const
```

Parameters. *None*.

Return value. The logfile group's name, a string (as a character pointer).

Example.

```
[To be supplied...]
```

2.3.13.4 LogfileGroup::getObjectId()

Description. This method is used to retrieve the object ID of the logfile group.

Signature.

```
virtual int getObjectId
    (
      void
    ) const
```

Parameters. *None*.

Return value. The logfile group's object ID (an integer value).

2.3.13.5 LogfileGroup::getObjectStatus()

Description. This method is used to obtain the object status of the `LogfileGroup`.

Signature.

```
virtual Object::Status getObjectStatus
    (
      void
    ) const
```

Parameters. *None*.

Return value. The logfile group's `Status`—see Section 2.3.31.4, “Object::Status” for possible values.

2.3.13.6 LogfileGroup::getObjectVersion()

Description. This method gets the logfile group's object version (see NDB Schema Object Versions).

Signature.

```
virtual int getObjectVersion
    (
      void
    ) const
```

Parameters. *None*.

Return value. The object version of the logfile group, as an integer.

2.3.13.7 LogfileGroup::getUndoBufferSize()

Description. This method retrieves the size of the logfile group's `UNDO` buffer.

Signature.

```
Uint32 getUndoBufferSize
    (
      void
    ) const
```

Parameters. *None*.

Return value. The size of the `UNDO` buffer, in bytes.

Example.

```
[To be supplied...]
```

2.3.13.8 LogfileGroup::getUndoFreeWords()

Description. This method retrieves the number of bytes unused in the logfile group's `UNDO` buffer.

Signature.

```
Uint64 getUndoFreeWords
    (
      void
    ) const
```

Parameters. *None*.

Return value. The number of bytes free, as a 64-bit integer.

Example.

```
[To be supplied...]
```

2.3.13.9 LogfileGroup::setAutoGrowSpecification()

Description. This method sets the `AutoGrowSpecification` data for the logfile group.

Signature.

```
void setAutoGrowSpecification
    (
      const AutoGrowSpecification& autoGrowSpec
    )
```

Parameters. The data is passed as a single parameter, an `AutoGrowSpecification` data structure—see Section 2.3.1, “The AutoGrowSpecification Structure”.

Return value. *None*.

2.3.13.10 LogfileGroup::setName()

Description. This method is used to set a name for the logfile group.

Signature.

```
void setName
    (
      const char* name
    )
```

Parameters. The *`name`* to be given to the logfile group (character pointer).

Return value. *None*.

Example.

```
[To be supplied...]
```

2.3.13.11 LogfileGroup::setUndoBufferSize()

Description. This method can be used to set the size of the logfile group's `UNDO` buffer.

Signature.

```
void setUndoBufferSize
    (
      Uint32 size
    )
```

Parameters. The *`size`* in bytes for the `UNDO` buffer (using a 32-bit unsigned integer value).

Return value. *None*.

Example.

```
[To be supplied...]
```

2.3.14 The List Class

This section covers the `List` class.

Parent class. Dictionary

Child classes. *None*

Description. The `List` class is a `Dictionary` subclass that is used for representing lists populated by the methods `Dictionary::listObjects()`, `Dictionary::listIndexes()`, and `Dictionary::listEvents()`.

Class Methods. This class has only two methods, a constructor and a destructor. Neither method takes any arguments.

Constructor. Calling the `List` constructor creates a new `List` whose `count` and `elements` attributes are both set equal to `0`.

Destructor. The destructor `~List()` is simply defined in such a way as to remove all elements and their properties. You can find its definition in the file `/storage/ndb/include/ndbapi/NdbDictionary.hpp`.

Attributes. A `List` has the following two attributes:

- `count`, an unsigned integer, which stores the number of elements in the list.

- `elements`, a pointer to an array of `Element` data structures contained in the list. See Section 2.3.5, "The Element Structure".

Types. The `List` class also defines an `Element` structure.

Note

For a graphical representation of this class and its parent-child relationships, see Section 2.3.4, "The Dictionary Class".

2.3.15 The Key_part_ptr Structure

This section describes the `Key_part_ptr` structure.

Parent class. `Ndb`

Description. `Key_part_ptr` provides a convenient way to define key-part data when starting transactions and computing hash values, by passing in pointers to distribution key values. When the distribution key has multiple parts, they should be passed as an array, with the last part's pointer set equal to `NULL`. See Section 2.3.16.34, "Ndb::startTransaction()", and Section 2.3.16.3, "Ndb::computeHash()", for more information about how this structure is used.

Attributes. A `Key_part_ptr` has the attributes shown in the following table:

Attribute	Type	Initial Value	Description
`ptr`	`const void*`	*none*	Pointer to one or more distribution key values
`len`	`unsigned`	*none*	The length of the pointer

2.3.16 The Ndb Class

This class represents the `NDB` kernel; it is the primary class of the NDB API.

Parent class. *None*

Child classes. *None*

Description. Any nontrivial NDB API program makes use of at least one instance of `Ndb`. By using several `Ndb` objects, it is possible to implement a multi-threaded application. You should remember that one `Ndb` object cannot be shared between threads; however, it is possible for a single thread to use multiple `Ndb` objects. A single application process can support a maximum of 4711 `Ndb` objects.

Resource consumption by Ndb objects. An `Ndb` object consumes memory in proportion to the size of the largest operation performed over the lifetime of the object. This is particularly noticeable in cases of large transactions; use of one or both of `BLOB` or `TEXT` columns; or both. This memory is held for the lifetime of the object, and once used in this way by the `Ndb` object, the only way to free this memory is to destroy the object (and then to create a new instance if desired).

Note

The `Ndb` object is multi-thread safe in that each `Ndb` object can be handled by one thread at a time. If an `Ndb` object is handed over to another thread, then the application must ensure that a memory barrier is used to ensure that the new thread sees all updates performed by the previous thread.

Semaphores and mutexes are examples of easy ways to provide memory barriers without having to bother about the memory barrier concept.

It is also possible to use multiple `Ndb` objects to perform operations on different clusters in a single application. See Application-level partitioning, for conditions and restrictions applying to such usage.

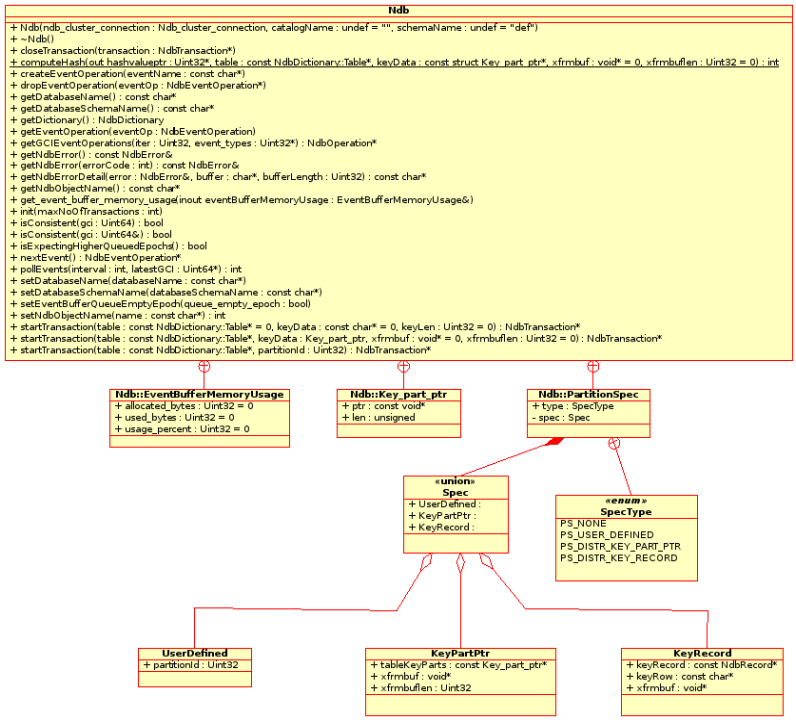
Methods. The following table lists the public methods of this class and the purpose or use of each method:

Method	Purpose / Use
`Ndb()`	Class constructor; represents a connection to an NDB Cluster.
`~Ndb()`	Class destructor; terminates a Cluster connection when it is no longer to be used
`closeTransaction()`	Closes a transaction.
`computeHash()`	Computes a distribution hash value.
`createEventOperation()`	Creates a subscription to a database event. (See Section 2.3.21, “The NdbEventOperation Class”.)
`dropEventOperation()`	Drops a subscription to a database event.
`getDictionary()`	Gets a dictionary, which is used for working with database schema information.
`getDatabaseName()`	Gets the name of the current database.
`getDatabaseSchemaName()`	Gets the name of the current database schema.
`get_eventbuf_max_alloc()`	Gets the current allocated maximum size of the event buffer. Added in NDB 7.2.14 and NDB 7.3.3.
`get_eventbuffer_free_percent()`	Gets the percentage of event buffer memory that should be available before buffering resumes, once the limit has been reached. Added in NDB 7.4.3.
`get_event_buffer_memory_usage()`	Provides event buffer memory usage information. Added in NDB 7.4.3.
`getGCIEventOperations()`	Gets the next event operation from a GCI. Deprecated in NDB 7.4.3.
`getHighestQueuedEpoch()`	Gets the latest epoch in the event queue. Added in NDB 7.4.3.
`getLatestGCI()`	Gets the most recent GCI. Deprecated in NDB 7.4.3.
`getNdbError()`	Retrieves an error. (See Section 2.3.20, “The NdbError Structure”.)
`getNdbErrorDetail()`	Retrieves extra error details.
`getNdbObjectName()`	Retrieves the `Ndb` object name if one was set. Added in NDB 7.2.17 and NDB 7.3.6.
`getNextEventOpInEpoch2()`	Gets the next event operation in this global checkpoint.
`getReference()`	Retrieves a reference or identifier for the `Ndb` object instance.
`init()`	Initializes an `Ndb` object and makes it ready for use.
`isConsistent()`	Whether all received events are consistent. Deprecated in NDB 7.4.3.
`isConsistentGCI()`	Whether all received events for a given global checkpoint are consistent. Deprecated in NDB 7.4.3.
`isExpectingHigherQueuedEpochs()`	Check whether there are new queued epochs, or there was a cluster failure event. Added in NDB 7.2.21, NDB 7.3.10, and NDB 7.4.7.
`nextEvent()`	Gets the next event from the queue. Deprecated in NDB 7.4.3.
`nextEvent2()`	Gets the next event from the queue. Added in NDB 7.4.3.
`pollEvents()`	Waits for an event to occur. Deprecated in NDB 7.4.3.
`pollEvents2()`	Waits for an event to occur. Added in NDB 7.4.3.
`setDatabaseName()`	Sets the name of the current database.

Method	Purpose / Use
`setDatabaseSchemaName()`	Sets the name of the current database schema.
`setEventBufferQueueEmptyEpoch()`	Enables queuing of empty events. Added in NDB 7.4.11 and NDB 7.5.2.
`set_eventbuf_max_alloc()`	Sets the current allocated maximum size of the event buffer. Added in NDB 7.2.14 and NDB 7.3.3.
`set_eventbuffer_free_percent()`	Sets the percentage of event buffer memory that should be available before buffering resumes, once the limit has been reached. Added in NDB 7.4.3.
`setNdbObjectName()`	For debugging purposes: sets an arbitrary name for this `Ndb` object. Added in NDB 7.2.17 and NDB 7.3.6.
`startTransaction()`	Begins a transaction. (See Section 2.3.30, “The NdbTransaction Class”.)

Class diagram. This diagram shows all the available members of the `Ndb` class:

Figure 2.11 Ndb



2.3.16.1 Ndb Class Constructor

Description. This creates an instance of `Ndb`, which represents a connection to the NDB Cluster. All NDB API applications should begin with the creation of at least one `Ndb` object. This requires the creation of at least one instance of `Ndb_cluster_connection`, which serves as a container for a cluster connection string.

Signature.

```
Ndb
    (
      Ndb_cluster_connection* ndb_cluster_connection,
      const char*                    catalogName = "",
      const char*                    schemaName = "def"
    )
```

Parameters. The `Ndb` class constructor can take up to 3 parameters, of which only the first is required:

- *`ndb_cluster_connection`* is an instance of `Ndb_cluster_connection`, which represents a cluster connection string. (See Section 2.3.17, “The Ndb_cluster_connection Class”.)

  Prior to NDB 7.3.8 and NDB 7.4.3, it was possible to delete the `Ndb_cluster_connection` used to create a given instance of `Ndb` without first deleting the dependent `Ndb` object. (Bug #19999242)

- *`catalogName`* is an optional parameter providing a namespace for the tables and indexes created in any connection from the `Ndb` object.

  This is equivalent to what `mysqld` considers “the database”.

  The default value for this parameter is an empty string.

- The optional *`schemaName`* provides an additional namespace for the tables and indexes created in a given catalog.

  The default value for this parameter is the string “def”.

Return value. An `Ndb` object.

~Ndb() (Class Destructor). The destructor for the `Ndb` class should be called in order to terminate an instance of `Ndb`. It requires no arguments, nor any special handling.

2.3.16.2 Ndb::closeTransaction()

Description. This is one of two NDB API methods provided for closing a transaction (the other being `NdbTransaction::close()`). You must call one of these two methods to close the transaction once it has been completed, whether or not the transaction succeeded.

Important

If the transaction has not yet been committed, it is aborted when this method is called. See Section 2.3.16.34, “Ndb::startTransaction()”.

Signature.

```
void closeTransaction
    (
      NdbTransaction *transaction
    )
```

Parameters. This method takes a single argument, a pointer to the `NdbTransaction` to be closed.

Return value. N/A.

2.3.16.3 Ndb::computeHash()

Description. This method can be used to compute a distribution hash value, given a table and its keys.

Important

`computeHash()` can be used only for tables that use native `NDB` partitioning.

Signature.

```
static int computeHash
    (
```

```
    Uint32*                     hashvalueptr,
    const NdbDictionary::Table* table,
    const struct Key_part_ptr*  keyData,
    void*                       xfrmbuf = 0,
    Uint32                      xfrmbuflen = 0
  )
```

Parameters. This method takes the following parameters:

- If the method call is successful, *`hashvalueptr`* is set to the computed hash value.
- A pointer to a *`table`* (see Section 2.3.37, “The Table Class”).
- *`keyData`* is a null-terminated array of pointers to the key parts that are part of the table's distribution key. The length of each key part is read from metadata and checked against the passed value (see Section 2.3.15, “The Key_part_ptr Structure”).
- *`xfrmbuf`* is a pointer to temporary buffer used to calculate the hash value.
- *`xfrmbuflen`* is the length of this buffer.

Note

If *`xfrmbuf`* is `NULL` (the default), then a call to `malloc()` or `free()` is made automatically, as appropriate. `computeHash()` fails if `xfrmbuf` is not `NULL` and *`xfrmbuflen`* is too small.

Previously, it was assumed that the memory returned by the `malloc()` call would always be suitably aligned, which is not always the case. Beginning with NDB Cluster versions 7.2.13 and 7.3.2, when `malloc()` provides a buffer to this method, the buffer is explicitly aligned after it is allocated, and before it is actually used. (Bug #16484617)

Return value. 0 on success, an error code on failure. (If the method call succeeds, the computed hash value is made available via *`hashvalueptr`*.)

2.3.16.4 Ndb::createEventOperation()

Description. This method creates a subscription to a database event.

Note

NDB API event subscriptions do not persist after an NDB Cluster has been restored using `ndb_restore`; in such cases, all of the subscriptions must be recreated explicitly.

Signature.

```
NdbEventOperation* createEventOperation
  (
    const char *eventName
  )
```

Parameters. This method takes a single argument, the unique *`eventName`* identifying the event to which you wish to subscribe.

Return value. A pointer to an `NdbEventOperation` object (or `NULL`, in the event of failure). See Section 2.3.21, “The NdbEventOperation Class”.

2.3.16.5 Ndb::dropEventOperation()

Description. This method drops a subscription to a database event represented by an `NdbEventOperation` object.

Important

Memory used by an event operation which has been dropped is not freed until the event buffer has been completely read. This means you must continue to call `pollEvents()` and `nextEvent()` in such cases until these methods return `0` and `NULL`, respectively in order for this memory to be freed.

Signature.

```
int dropEventOperation
    (
      NdbEventOperation *eventOp
    )
```

Parameters. This method requires a single input parameter, a pointer to an instance of `NdbEventOperation`.

Return value. **0** on success; any other result indicates failure.

2.3.16.6 Ndb::getDictionary()

Description. This method is used to obtain an object for retrieving or manipulating database schema information. This `Dictionary` object contains meta-information about all tables in the cluster.

Note

The dictionary returned by this method operates independently of any transaction. See Section 2.3.4, “The Dictionary Class”, for more information.

Signature.

```
NdbDictionary::Dictionary* getDictionary
    (
      void
    ) const
```

Parameters. *None*.

Return value. An instance of the `Dictionary` class.

2.3.16.7 Ndb::getDatabaseName()

Description. This method can be used to obtain the name of the current database.

Signature.

```
const char* getDatabaseName
    (
      void
    )
```

Parameters. None.

Return value. The name of the current database.

2.3.16.8 Ndb::getDatabaseSchemaName()

Description. This method can be used to obtain the current database schema name.

Signature.

```
const char* getDatabaseSchemaName
    (
      void
    )
```

Parameters. None.

Return value. The name of the current database schema.

2.3.16.9 Ndb::getGCIEventOperations()

Description. Iterates over distinct event operations which are part of the current GCI, becoming valid after calling `nextEvent()`. You can use this method to obtain summary information for the epoch (such as a list of all tables) before processing the event data.

This method is deprecated in NDB 7.4.3, and is subject to removal in a future release. In NDB 7.4.3 and later, you should use `getNextEventOpInEpoch2()` instead.

Signature.

```
const NdbEventOperation* getGCIEventOperations
    (
      Uint32* iter,
      Uint32* event_types
    )
```

Parameters. An iterator and a mask of event types. Set `*iter=0` to start.

Return value. The next event operation; returns `NULL` when there are no more event operations. If *`event_types`* is not `NULL`, then after calling the method it contains a bitmask of the event types received. .

2.3.16.10 Ndb::get_eventbuf_max_alloc()

Description. Gets the maximum memory, in bytes, that can be used for the event buffer. This is the same as reading the value of the `ndb_eventbuffer_max_alloc` system variable in the MySQL Server.

This method was added in NDB 7.2.14 and NDB 7.3.3.

Signature.

```
unsigned get_eventbuf_max_alloc
    (
      void
    )
```

Parameters. *None*.

Return value. The mamximum memory available for the event buffer, in bytes.

2.3.16.11 Ndb::get_eventbuffer_free_percent()

Description. Gets `ndb_eventbuffer_free_percent`—that is, the percentage of event buffer memory that should be available before buffering resumes, once `ndb_eventbuffer_max_alloc` has been reached. This value is calculated as *`used`* `* 100 /` `ndb_eventbuffer_max_alloc`, where *`used`* is the amount of event buffer memory actually used, in bytes.

This method was added in NDB 7.4.3.

Signature.

```
unsigned get_eventbuffer_free_percent
    (
      void
    )
```

Parameters. The percentage (*`pct`*) of event buffer memory that must be present. Valid range is 1 to 99 inclusive.

Return value. *None*.

2.3.16.12 Ndb::get_event_buffer_memory_usage()

Description. Gets event buffer usage as a percentage of `ndb_eventbuffer_max_alloc`. Unlike `get_eventbuffer_free_percent()`, this method makes complete usage information available in the form of an `EventBufferMemoryUsage` data structure.

This method was added in NDB 7.4.3.

Signature.

```
void get_event_buffer_memory_usage
    (
      EventBufferMemoryUsage&
    )
```

Parameters. A reference to an `EventBufferMemoryUsage` structure, which receives the usage data.

Return value. *None*.

2.3.16.13 Ndb::getHighestQueuedEpoch()

Description. Added in NDB 7.4.3, this method supersedes `getLatestGCI()`, which is now deprecated and subject to removal in a future NDB Cluster release.

Prior to NDB 7.4.7, this method returned the highest epoch number in the event queue. In NDB 7.4.7 and later, it returns the highest epoch number found after calling `pollEvents2()` (Bug #20700220).

Signature.

```
Uint64 getHighestQueuedEpoch
    (
      void
    )
```

Parameters. *None*.

Return value. The most recent epoch number, an integer.

2.3.16.14 Ndb::getLatestGCI()

Description. Gets the index for the most recent global checkpoint.

This method is deprecated in NDB 7.4.3, and is subject to removal in a future release. In NDB 7.4.3 and later, you should use `getHighestQueuedEpoch()` instead.

Signature.

```
Uint64 getLatestGCI
    (
      void
    )
```

Parameters. *None*.

Return value. The most recent GCI, an integer.

2.3.16.15 Ndb::getNdbError()

Description. This method provides you with two different ways to obtain an `NdbError` object representing an error condition. For more detailed information about error handling in the NDB API, see NDB Cluster API Errors.

Signature. The `getNdbError()` method actually has two variants.

The first of these simply gets the most recent error to have occurred:

```
const NdbError& getNdbError
    (
      void
    )
```

The second variant returns the error corresponding to a given error code:

```
const NdbError& getNdbError
    (
      int errorCode
    )
```

Regardless of which version of the method is used, the `NdbError` object returned persists until the next NDB API method is invoked.

Parameters. To obtain the most recent error, simply call `getNdbError()` without any parameters. To obtain the error matching a specific *`errorCode`*, invoke the method passing the code (an `int`) to it as a parameter. For a listing of NDB API error codes and corresponding error messages, see Section 2.4, “NDB API Errors and Error Handling”.

Return value. An `NdbError` object containing information about the error, including its type and, where applicable, contextual information as to how the error arose. See Section 2.3.20, “The NdbError Structure”, for details.

2.3.16.16 Ndb::getNdbErrorDetail()

Description. This method provides an easy and safe way to access any extra information about an error. Rather than reading these extra details from the `NdbError` object's `details` property (now now deprecated in favor of `getNdbErrorDetail()`-see Bug #48851). This method enables storage of such details in a user-supplied buffer, returning a pointer to the beginning of this buffer. In the event that the string containing the details exceeds the length of the buffer, it is truncated to fit.

`getErrorDetail()` provides the source of an error in the form of a string. In the case of a unique constraint violation (error 893), this string supplies the fully qualified name of the index where the problem originated, in the format *`database-name`*/*`schema-name`*/*`table-name`*/*`index-name`*, (`NdbError.details`, on the other hand, supplies only an index ID, and it is often not readily apparent to which table this index belongs.) Regardless of the type of error and details concerning this error, the string retrieved by `getErrorDetail()` is always null-terminated.

Signature. The `getNdbErrorDetail()` method has the following signature:

```
const char* getNdbErrorDetail
            (
              const NdbError& error,
              char*           buffer,
              Uint32          bufferLength
            ) const
```

Parameters. To obtain detailed information about an error, call `getNdbErrorDetail()` with a reference to the corresponding `NdbError` object, a *`buffer`*, and the length of this buffer (expressed as an unsigned 32-bit integer).

Return value. When extra details about the *`error`* are available, this method returns a pointer to the beginning of the *`buffer`* supplied. As stated previously, if the string containing the details is longer than *`bufferLength`*, the string is truncated to fit. In the event that no addition details are available, `getNdbErrorDetail()` returns `NULL`.

2.3.16.17 Ndb::getNdbObjectName()

Description. If a name was set for the `Ndb` object prior to its initialization, you can retrieve it using this method. Used for debugging.

Signature.

```
const char* getNdbObjectName
    (
      void
    ) const
```

Parameters. *None*.

Return value. The `Ndb` object name, if one has been set using `setNdbObjectName()`. Otherwise, this method returns 0.

This method was added in NDB 7.2.17, and NDB 7.3.6. (Bug #18419907)

2.3.16.18 Ndb::getNextEventOpInEpoch2()

Description. Iterates over individual event operations making up the current global checkpoint. Use following `nextEvent2()` to obtain summary information for the epoch, such as a listing of all tables, before processing event data.

Note

Exceptional epochs do not have any event operations associated with them.

Signature.

```
const NdbEventOperation* getNextEventOpInEpoch2
    (
      Uint32* iter,
      Uint32* event_types
    )
```

Parameters. Set *`iter`* to 0 initially; this is *`NULL`* when there are no more events within this epoch. If *`event_types`* is not *`NULL`*, it holds a bitmask of the event types received.

Return value. A pointer to the next `NdbEventOperation`, if there is one.

2.3.16.19 Ndb::getReference()

Description. This method can be used to obtain a reference to a given `Ndb` object. This is the same value that is returned for a given operation corresponding to this object in the output of `DUMP 2350`. (See DUMP 2350, for an example.)

Signature.

```
Uint32 getReference
    (
      void
```

```
    )
```

Parameters. *None*.

Return value. A 32-bit unsigned integer.

2.3.16.20 Ndb::init()

Description. This method is used to initialize an `Ndb` object.

Signature.

```
int init
    (
      int maxNoOfTransactions = 4
    )
```

Parameters. The `init()` method takes a single parameter *`maxNoOfTransactions`* of type integer. This parameter specifies the maximum number of parallel `NdbTransaction` objects that can be handled by this instance of `Ndb`. The maximum permitted value for *`maxNoOfTransactions`* is 1024; if not specified, it defaults to 4.

Note

Each scan or index operation uses an extra `NdbTransaction` object.

Return value. This method returns an `int`, which can be either of the following two values:

- **0**: indicates that the `Ndb` object was initialized successfully.
- **-1**: indicates failure.

2.3.16.21 Ndb::isConsistent()

Description. Check if all events are consistent. If a node failure occurs when resources are exhausted, events may be lost and the delivered event data might thus be incomplete. This method makes it possible to determine if this is the case.

This method is deprecated in NDB 7.4.3, and is subject to removal in a future release. In NDB 7.4.3 and later, you should instead use `NdbEventOperation::getEventType2()` to determine the type of event—in this instance, whether the event is of type `TE_INCONSISTENT`. See Section 2.3.6.23, “Event::TableEvent”.

Signature.

```
bool isConsistent
    (
      Uint64& gci
    )
```

Parameters. A reference to a global checkpoint index. This is the first inconsistent GCI found, if any.

Return value. `true` if all events are consistent.

2.3.16.22 Ndb::isConsistentGCI()

Description. If a node failure occurs when resources are exhausted, events may be lost and the delivered event data might thus be incomplete. This method makes it possible to determine if this is the case by checking whether all events in a given GCI are consistent.

This method is deprecated in NDB 7.4.3, and is subject to removal in a future release. In NDB 7.4.3 and later, you should instead use `NdbEventOperation::getEventType2()` to determine the type

of event—in this instance, whether the event is of type `TE_INCONSISTENT`. See Section 2.3.6.23, "Event::TableEvent".

Signature.

```
bool isConsistentGCI
    (
      Uint64 gci
    )
```

Parameters. A global checkpoint index.

Return value. `true` if this GCI is consistent; `false` indicates that the GCI may be possibly inconsistent.

2.3.16.23 Ndb::isExpectingHigherQueuedEpochs()

Description. Check whether higher queued epochs have been seen by the last invocation of `Ndb::pollEvents2()`, or whether a **`TE_CLUSTER_FAILURE`** event was found.

It is possible, after a cluster failure has been detected, for the highest queued epoch returned by `pollEvents2()` not to be increasing any longer. In this case, rather than poll for more events, you should instead consume events with `nextEvent()` until it detects a **`TE_CLUSTER_FAILURE`** is detected, then reconnect to the cluster when it becomes available again.

This method was added in NDB 7.2.21, NDB 7.3.10, and NDB 7.4.7 (Bug #18753887).

Signature.

```
bool isExpectingHigherQueuedEpochs
      (
        void
      )
```

Parameters. *None*.

Return value. True if queued epochs were seen by the last `pollEvents2()` call or, in the event of cluster failure.

2.3.16.24 Ndb::nextEvent()

Description. Returns the next event operation having data from a subscription queue.

This method is deprecated in NDB 7.4.3, and is subject to removal in a future release. In NDB 7.4.3 and later, you should use `nextEvent2()` instead.

Signature.

```
NdbEventOperation* nextEvent
    (
      void
    )
```

Parameters. None.

Return value. This method returns an `NdbEventOperation` object representing the next event in a subscription queue, if there is such an event. If there is no event in the queue, it returns `NULL` instead.

Beginning with NDB 7.2.17 and NDB 7.3.6, this method clears inconsistent data events from the event queue when processing them. In order to able to clear all such events in these and later versions, applications must call this method even in cases when `pollEvents()` has already returned 0. (Bug #18716991)

2.3.16.25 Ndb::nextEvent2()

Description. Returns the event operation associated with the data dequeued from the event queue. This should be called repeatedly after `pollEvents2()` populates the queue, until the event queue is empty.

Added in NDB 7.4.3, this method supersedes `nextEvent()`, which is now deprecated and subject to removal in a future NDB Cluster release.

After calling this method, use `NdbEventOperation::getEpoch()` to determine the epoch, then check the type of the returned event data using `NdbEventOperation::getEventType2()`. Handling must be provided for all exceptional `TableEvent` types, including `TE_EMPTY`, `TE_INCONSISTENT`, and `TE_OUT_OF_MEMORY` (also introduced in NDB 7.4.3). No other `NdbEventOperation` methods than the two named here should be called for an exceptional epoch. Returning empty epochs (`TE_EMPTY`) may flood applications when data nodes are idle. If this is not desirable, applications should filter out any empty epochs.

Signature.

```
NdbEventOperation* nextEvent2
    (
      void
    )
```

Parameters. None.

Return value. This method returns an `NdbEventOperation` object representing the next event in an event queue, if there is such an event. If there is no event in the queue, it returns `NULL` instead.

2.3.16.26 Ndb::pollEvents()

Description. This method waits for a GCP to complete. It is used to determine whether any events are available in the subscription queue.

This method waits for the next *epoch*, rather than the next GCP. See Section 2.3.21, “The NdbEventOperation Class”, for more information.

In NDB 7.4.3 and later, you can (and should) use `pollEvents2()` instead of this method.

Prior to NDB 7.4.7, `pollEvents()` was not compatible with the exceptional `TableEvent` types added in NDB 7.4.3 (Bug #20646496); in NDB 7.4.7 and later, `pollEvents()` is compatible with these event types, as described later in this section.

Signature.

```
int pollEvents
    (
      int     maxTimeToWait,
      Uint64* latestGCI = 0
    )
```

Parameters. This method takes the two parameters listed here:

- The maximum time to wait, in milliseconds, before “giving up” and reporting that no events were available (that is, before the method automatically returns `0`).

  A negative value causes the wait to be indefinite and never time out. This is not recommended (and is not supported by the successor method `pollEvents2()`).

- The index of the most recent global checkpoint. Normally, this may safely be permitted to assume its default value, which is `0`.

Return value. `pollEvents()` returns a value of type `int`, which may be interpreted as follows:

- `> 0`: There are events available in the queue.
- `0`: There are no events available.
- In NDB 7.4.7 and later, a negative value indicates failure and **`NDB_FAILURE_GCI`** (`~(Uint64)0`) indicates cluster failure (Bug #18753887); 1 is returned when encountering an exceptional event, except when only `TE_EMPTY` events are found, as described later in this section.

In NDB 7.4.7 and later, when `pollEvents()` finds an exceptional event at the head of the event queue, the method returns 1 and otherwise behaves as follows:

- Empty events (`TE_EMPTY`) are removed from the event queue head until an event containing data is found. When this results in the entire queue being processed without encountering any data, the method returns 0 (no events available) rather than 1. This behavior makes this event type transparent to an application using `pollEvents()`.
- After encountering an event containing inconsistent data (`TE_INCONSISTENT`) due to data node buffer overflow, the next call to `nextEvent()` call removes the inconsistent data event data from the event queue, and returns `NULL`. You should check the inconsistency by calling `isConsistent()` immediately thereafter.

  Important: Although the inconsistent event data is removed from the event queue by calling `nextEvent()`, information about the inconsistency is removed only by another `nextEvent()` call following this, that actually finds an event containing data.
- When `pollEvents()` finds a data buffer overflow event (`TE_OUT_OF_MEMORY`), the event data is added to the event queue whenever event buffer usage exceeds `ndb_eventbuffer_max_alloc`. In this case, the next call to `nextEvent()` exits the process.

2.3.16.27 Ndb::pollEvents2()

Description. Waits for an event to occur. Returns as soon as any event data is available. This method also moves an epoch's complete event data to the event queue.

Added in NDB 7.4.3, this method supersedes `pollEvents()`, which is now deprecated and subject to removal in a future NDB Cluster release.

Signature.

```
int pollEvents2
    (
      int aMillisecondNumber,
      Uint64* highestQueuedEpoch = 0
    )
```

Parameters. This method takes the two parameters listed here:

- The maximum time to wait, in milliseconds, before giving up and reporting that no events were available (that is, before the method automatically returns `0`).

  In NDB 7.4.7 and later, specifying a negative value for this argument causes `pollEvents2()` to return -1, indicating an error (Bug #20762291).
- The index of the highest queued epoch. Normally, this may safely be permitted to assume its default value, which is `0`. If this value is not `NULL` and new event data is available in the event queue, it is set to the highest epoch found in the available event data.

Return value. `pollEvents2()` returns an integer whose value can be interpreted as follows:

- `> 0`: There are events available in the queue.

- `0`: There are no events available.
- `< 0`: Indicates failure (possible error).

2.3.16.28 Ndb::setDatabaseName()

Description. This method is used to set the name of the current database.

Signature.

```
void setDatabaseName
    (
      const char *databaseName
    )
```

Parameters. `setDatabaseName()` takes a single, required parameter, the name of the new database to be set as the current database.

Return value. N/A.

2.3.16.29 Ndb::setDatabaseSchemaName()

Description. This method sets the name of the current database schema.

Signature.

```
void setDatabaseSchemaName
    (
      const char *databaseSchemaName
    )
```

Parameters. The name of the database schema.

Return value. N/A.

2.3.16.30 Ndb::setEventBufferQueueEmptyEpoch()

Description. Queuing of empty epochs is disabled by default. This method can be used to enable such queuing, in which case any new, empty epochs entering the event buffer following the method call are queued.

When queuing of empty epochs is enabled, `nextEvent()` associates an empty epoch to one and only one of the subscriptions (event operations) connected to the subscribing `Ndb` object. This means that there can be no more than one empty epoch per subscription, even though the user may have many subscriptions associated with the same `Ndb` object.

Signature.

```
void setEventBufferQueueEmptyEpoch
  (
    bool queue_empty_epoch
  )
```

Parameters. This method takes a single input parameter, a boolean. Invoking the method with `true` enables queuing of empty events; passing `false` to the method disables such queuing.

Return value. *None*.

Note

`setEventBufferQueueEmptyEpoch()` has no associated getter method. This is intentional, and is due to the fact this setter applies to queuing *new*

> epochs, whereas the queue itself may still reflect the state of affairs that existed prior to invoking the setter. Thus, during a transition period, an empty epoch might be found in the queue even if queuing is turned off.

`setEventBufferQueueEmptyEpoch()` was added in NDB 7.4.11 and NDB 7.5.2.

2.3.16.31 Ndb::set_eventbuf_max_alloc()

Description. Sets the maximum memory, in bytes, that can be used for the event buffer. This has the same effect as setting the value of the `ndb_eventbuffer_max_alloc` system variable in the MySQL Server.

This method was added in NDB 7.2.14 and NDB 7.3.3.

Signature.

```
void set_eventbuf_max_alloc
    (
      unsigned size
    )
```

Parameters. The desired maximum *`size`* for the event buffer, in bytes.

Return value. *None*.

2.3.16.32 Ndb::set_eventbuffer_free_percent()

Description. Sets `ndb_eventbuffer_free_percent`—that is, the percentage of event buffer memory that should be available before buffering resumes, once `ndb_eventbuffer_max_alloc` has been reached.

This method was added in NDB 7.4.3.

Signature.

```
int set_eventbuffer_free_percent
    (
      unsigned pct
    )
```

Parameters. The percentage (*`pct`*) of event buffer memory that must be present. Valid range is 1 to 99 inclusive.

Return value. The value that was set.

2.3.16.33 Ndb::setNdbObjectName()

Description. Starting with NDB 7.2.17 and NDB 7.3.6, you can set an arbitrary, human-readable name to identify an `Ndb` object for debugging purposes. This name can then be retrieved using `getNdbObjectName()`. (Bug #18419907) This must be done prior to calling `init()` for this object; trying to set a name after initialization fails with an error.

You can set a name only once for a given `Ndb` object; subsequent attempts after the name has already been set fail with an error.

Signature.

```
int setNdbObjectName
    (
      const char* name
    )
```

Parameters. A *`name`* that is intended to be human-readable.

Return value. 0 on success.

2.3.16.34 Ndb::startTransaction()

Description. This method is used to begin a new transaction. There are three variants, the simplest of these using a table and a partition key or partition ID to specify the transaction coordinator (TC). The third variant makes it possible for you to specify the TC by means of a pointer to the data of the key.

Important

When the transaction is completed it must be closed using `NdbTransaction::close()` or `Ndb::closeTransaction()`. Failure to do so aborts the transaction. This must be done regardless of the transaction's final outcome, even if it fails due to an error.

See Section 2.3.16.2, “Ndb::closeTransaction()”, and Section 2.3.30.1, “NdbTransaction::close()”, for more information.

Signature.

```
NdbTransaction* startTransaction
    (
      const NdbDictionary::Table* table = 0,
      const char* keyData = 0,
      Uint32* keyLen = 0
    )
```

Parameters. This method takes the following three parameters:

- *`table`*: A pointer to a `Table` object. This is used to determine on which node the transaction coordinator should run.
- *`keyData`*: A pointer to a partition key corresponding to *`table`*.
- *`keyLen`*: The length of the partition key, expressed in bytes.

Distribution-aware forms of startTransaction(). It is also possible to employ *distribution awareness* with this method; that is, to suggest which node should act as the transaction coordinator.

Signature.

```
NdbTransaction* startTransaction
    (
      const NdbDictionary::Table* table,
      const struct Key_part_ptr*  keyData,
      void*                       xfrmbuf = 0,
      Uint32                      xfrmbuflen = 0
    )
```

Parameters. When specifying the transaction coordinator, this method takes the four parameters listed here:

- A pointer to a *`table`* (`Table` object) used for deciding which node should act as the transaction coordinator.
- A null-terminated array of pointers to the values of the distribution key columns. The length of the key part is read from metadata and checked against the passed value.

  A `Key_part_ptr` is defined as shown in Section 2.3.15, “The Key_part_ptr Structure”.
- A pointer to a temporary buffer, used to calculate the hash value.

- The length of the buffer.

If *xfrmbuf* is NULL (the default), then a call to malloc() or free() is made automatically, as appropriate. startTransaction() fails if xfrmbuf is not NULL and *xfrmbuflen* is too small.

Example. Suppose that the table's partition key is a single BIGINT column. Then you would declare the distribution key array as shown here:

```
Key_part_ptr distkey[2];
```

The value of the distribution key would be defined as shown here:

```
unsigned long long distkeyValue= 23;
```

The pointer to the distribution key array would be set as follows:

```
distkey[0].ptr= (const void*) &distkeyValue;
```

The length of this pointer would be set accordingly:

```
distkey[0].len= sizeof(distkeyValue);
```

The distribution key array must terminate with a NULL element. This is necessary to avoid to having an additional parameter providing the number of columns in the distribution key:

```
distkey[1].ptr= NULL;
distkey[1].len= NULL;
```

Setting the buffer to NULL permits startTransaction() to allocate and free memory automatically:

```
xfrmbuf= NULL;
xfrmbuflen= 0;
```

Note

You can also specify a buffer to save having to make explicit malloc() and free() calls, but calculating an appropriate size for this buffer is not a simple matter; if the buffer is not NULL but its length is too short, then the startTransaction() call fails. However, if you choose to specify the buffer, 1 MB is usually a sufficient size.

Now, when you start the transaction, you can access the node that contains the desired information directly.

Another distribution-aware version of this method makes it possible for you to specify a table and partition (using the partition ID) as a hint for selecting the transaction coordinator, and is defined as shown here:

```
NdbTransaction* startTransaction
    (
      const NdbDictionary::Table* table,
      Uint32 partitionId
    )
```

In the event that the cluster has the same number of data nodes as it has replicas, specifying the transaction coordinator gains no improvement in performance, since each data node contains the entire database. However, where the number of data nodes is greater than the number of replicas (for example, where NoOfReplicas is set equal to 2 in a cluster with 4 data nodes), you should see a marked improvement in performance by using the distribution-aware version of this method.

It is still possible to use this method as before, without specifying the transaction coordinator. In either case, you must still explicitly close the transaction, whether or not the call to `startTransaction()` was successful.

Return value. On success, an `NdbTransaction` object. In the event of failure, `NULL` is returned.

2.3.17 The Ndb_cluster_connection Class

This class represents a connection to a cluster of data nodes.

Parent class. *None*

Child classes. *None*

Description. An NDB application program should begin with the creation of a single `Ndb_cluster_connection` object, and typically makes use of a single `Ndb_cluster_connection`. The application connects to a cluster management server when this object's `connect()` method is called. By using the `wait_until_ready()` method it is possible to wait for the connection to reach one or more data nodes.

Note

An instance of `Ndb_cluster_connection` used to create an `Ndb` object. Prior to NDB 7.3.8 and NDB 7.4.3, it was possible to delete the `Ndb_cluster_connection` used to create a given instance of `Ndb` without first deleting the dependent `Ndb` object. (Bug #19999242)

Application-level partitioning. There is no restriction against instantiating multiple `Ndb_cluster_connection` objects representing connections to different management servers in a single application, nor against using these for creating multiple instances of the `Ndb` class. Such `Ndb_cluster_connection` objects (and the `Ndb` instances based on them) are not required even to connect to the same cluster.

For example, it is entirely possible to perform *application-level partitioning* of data in such a manner that data meeting one set of criteria are “handed off” to one cluster using an `Ndb` object that makes use of an `Ndb_cluster_connection` object representing a connection to that cluster, while data not meeting those criteria (or perhaps a different set of criteria) can be sent to a different cluster through a different instance of Ndb that makes use of an `Ndb_cluster_connection` “pointing” to the second cluster.

It is possible to extend this scenario to develop a single application that accesses an arbitrary number of clusters. However, in doing so, the following conditions and requirements must be kept in mind:

- A cluster management server (`ndb_mgmd`) can connect to one and only one cluster without being restarted and reconfigured, as it must read the data telling it which data nodes make up the cluster from a configuration file (`config.ini`).
- An `Ndb_cluster_connection` object “belongs” to a single management server whose host name or IP address is used in instantiating this object (passed as the *`connection_string`* argument to its constructor); once the object is created, it cannot be used to initiate a connection to a different management server.

  (See Section 2.3.17.1, “Ndb_cluster_connection Class Constructor”.)
- An `Ndb` object making use of this connection (`Ndb_cluster_connection`) cannot be re-used to connect to a different cluster management server (and thus to a different collection of data nodes making up a cluster). Any given instance of `Ndb` is bound to a specific `Ndb_cluster_connection` when created, and that `Ndb_cluster_connection` is in turn bound to a single and unique management server when it is instantiated.

  (See Section 2.3.16.1, “Ndb Class Constructor”.)

- The bindings described above persist for the lifetimes of the `Ndb` and `Ndb_cluster_connection` objects in question.

Therefore, it is imperative in designing and implementing any application that accesses multiple clusters in a single session, that a separate set of `Ndb_cluster_connection` and `Ndb` objects be instantiated for connecting to each cluster management server, and that no confusion arises as to which of these is used to access which NDB Cluster.

It is also important to keep in mind that no direct “sharing” of data or data nodes between different clusters is possible. A data node can belong to one and only one cluster, and any movement of data between clusters must be accomplished on the application level.

For examples demonstrating how connections to two different clusters can be made and used in a single application, see Section 2.5.2, “NDB API Example Using Synchronous Transactions and Multiple Clusters”, and Section 3.6.2, “MGM API Event Handling with Multiple Clusters”.

Methods. The following table lists the public methods of this class and the purpose or use of each method:

Method	Purpose / Use
`Ndb_cluster_connection()`	Constructor; creates a connection to a cluster of data nodes.
`connect()`	Connects to a cluster management server.
`get_auto_reconnect()`	Gets the auto-reconnection setting for API nodes using this `Ndb_cluster_connection`.
`get_latest_error()`	Whether or not the most recent attempt to connect succeeded.
`get_latest_error_msg()`	If the most recent attempt to connect failed, provides the reason.
`get_max_adaptive_send_time()`	Get timeout before adaptive send forces the sending of all pending signals.
`get_num_recv_threads()`	Get number of receive threads.
`get_next_ndb_object()`	Used to iterate through multiple `Ndb` objects.
`get_recv_thread_activation_threshold()`	Get activation level for bound receive threads.
`lock_ndb_objects()`	Disables the creation of new `Ndb` objects.
`set_auto_reconnect()`	Enables or disables auto-reconnection of API nodes using this `Ndb_cluster_connection`.
`set_data_node_neighbour()`	Sets a neighbor node for for optimal transaction coordinator placement
`set_max_adaptive_send_time()`	Set timeout to elapse before adaptive send forces the sending of all pending signals.
`set_name()`	Provides a name for the connection
`set_num_recv_threads()`	Set number of receive threads to be bound.
`set_recv_thread_cpu()`	Set one or more CPUs to bind receive threads to.
`set_optimized_node_selection()`	Used to control node-selection behavior.
`set_timeout()`	Sets a connection timeout
`unlock_ndb_objects()`	Enables the creation of new `Ndb` objects.
`unset_recv_thread_cpu()`	Unset the binding of the receive thread to one or more CPUs.
`wait_until_ready()`	Waits until a connection with one or more data nodes is successful.

Class diagram. This diagram shows all the available methods of the `Ndb_cluster_connection` class:

Figure 2.12 Ndb_cluster_connection

2.3.17.1 Ndb_cluster_connection Class Constructor

Description. This method creates a connection to an NDB Cluster, that is, to a cluster of data nodes. The object returned by this method is required in order to instantiate an `Ndb` object. Thus, every NDB API application requires the use of an `Ndb_cluster_connection`.

`Ndb_cluster_connection` has two constructors. The first of these is shown here:

Signature.

```
Ndb_cluster_connection
    (
      const char* connection_string = 0
    )
```

Parameters. This version of the constructor requires a single *`connection_string`* parameter, pointing to the location of the management server.

The second constructor takes a node ID in addition to the connection string argument. Its signature and parameters are shown here:

Signature.

```
Ndb_cluster_connection
    (
      const char* connection_string,
      int force_api_nodeid
    )
```

Parameters. This version of the constructor takes two arguments, a *`connection_string`* and the node ID (*`force_api_nodeid`*) to be used by this API node. This node ID overrides any node ID value set in the *`connection_string`* argument.

Return value. (*Both versions*:) An instance of `Ndb_cluster_connection`.

2.3.17.2 Ndb_cluster_connection::connect()

Description. This method connects to a cluster management server.

Signature.

```
int connect
    (
      int retries = 30,
      int delay   = 1,
      int verbose = 0
    )
```

Parameters. This method takes three parameters, all of which are optional:

- *`retries`* specifies the number of times to retry the connection in the event of failure. The default value is 30.

  `0` means that no additional attempts to connect are made in the event of failure; using a negative value for *`retries`* results in the connection attempt being repeated indefinitely.

- The *`delay`* represents the number of seconds between reconnect attempts; the default is `1` second.

- *`verbose`* indicates whether the method should output a report of its progress, with `1` causing this reporting to be enabled; the default is `0` (reporting disabled).

Return value. This method returns an `int`, which can have one of the following 3 values:

- **0**: The connection attempt was successful.

- **1**: Indicates a recoverable error.

- **-1**: Indicates an unrecoverable error.

2.3.17.3 Ndb_cluster_connection::get_auto_reconnect()

Description. This method retrieves the current `AutoReconnect` setting for a given `Ndb_cluster_connection`. For more detailed information, see Section 2.3.17.11, “Ndb_cluster_connection::set_auto_reconnect()”.

Signature.

```
int get_auto_reconnect
    (
      void
    )
```

Parameters. *None*.

Return value. An integer value `0` or `1`, corresponding to the current `AutoReconnect` setting in effect for for this connection. `0` forces API nodes to use new connections to the cluster, while `1` enables API nodes to re-use existing connections.

2.3.17.4 Ndb_cluster_connection::get_latest_error()

Description. This method can be used to determine whether or not the most recent `connect()` attempt made by this `Ndb_cluster_connection` succeeded . If the connection succeeded,

`get_latest_error()` returns `0`; otherwise, it returns `1`. If the connection attempt failed, use `Ndb_cluster_connection::get_latest_error_msg()` to obtain an error message giving the reason for the failure.

Signature.

```
int get_latest_error
    (
      void
    ) const
```

Parameters. *None*.

Return value. `1` or `0`. A return value of `1` indicates that the latest attempt to connect failed; if the attempt succeeded, a `0` is returned.

2.3.17.5 Ndb_cluster_connection::get_latest_error_msg()

Description. If the most recent connection attempt by this `Ndb_cluster_connection` failed (as determined by calling `get_latest_error()`), this method provides an error message supplying information about the reason for the failure.

Signature.

```
const char* get_latest_error_msg
    (
      void
    ) const
```

Parameters. *None*.

Return value. A string containing an error message describing a failure by `Ndb_cluster_connection::connect()`. If the most recent connection attempt succeeded, an empty string is returned.

2.3.17.6 Ndb_cluster_connection::get_max_adaptive_send_time()

Description. Get the minimum time in milliseconds that is permit to lapse before the adaptive send mechanism forces all pending signals to be sent.

Signature.

```
Uint32 get_max_adaptive_send_time
    (

    )
```

Parameters. *None*.

Return value. Wait time as a number of milliseconds. This should always be a value between 0 and 10, inclusive.

2.3.17.7 Ndb_cluster_connection::get_next_ndb_object()

Description. This method is used to iterate over a set of `Ndb` objects, retrieving them one at a time.

Signature.

```
const Ndb* get_next_ndb_object
    (
      const Ndb* p
    )
```

Parameters. This method takes a single parameter, a pointer to the last `Ndb` object to have been retrieved or `NULL`.

Return value. Returns the next `Ndb` object, or `NULL` if no more `Ndb` objects are available.

Iterating over Ndb objects. To retrieve all existing `Ndb` objects, perform the following three steps:

1. Invoke the `lock_ndb_objects()` method. This prevents the creation of any new instances of `Ndb` until the `unlock_ndb_objects()` method is called.
2. Retrieve the first available `Ndb` object by passing `NULL` to `get_next_ndb_object()`. You can retrieve the second `Ndb` object by passing the pointer retrieved by the first call to the next `get_next_ndb_object()` call, and so on. When a pointer to the last available `Ndb` instance is used, the method returns `NULL`.
3. After you have retrieved all desired `Ndb` objects, you should re-enable `Ndb` object creation by calling the `unlock_ndb_objects()` method.

2.3.17.8 Ndb_cluster_connection::get_num_recv_threads()

Description. Get the number of receiver threads.

Signature.

```
int get_num_recv_threads
    (
      void
    ) const
```

Parameters. *None*.

Return value. The number of receiver threads.

2.3.17.9

Description. Get the level set for activating the receiver thread bound by `set_recv_thread_cpu()`.

Signature.

```
int get_recv_thread_activation_threshold
    (
      void
    ) const
```

Parameters. *None*.

Return value. An integer threshold value. See Section 2.3.17.17, “”, for information about interpreting this value.

2.3.17.10 ndb_cluster_connection::lock_ndb_objects()

Description. Calling this method prevents the creation of new instances of the `Ndb` class. This method must be called prior to iterating over multiple `Ndb` objects using `get_next_ndb_object()`.

Signature.

```
void lock_ndb_objects
    (
      void
    ) const
```

Parameters. *None*.

Return value. *None*.

This method was made `const` in NDB 7.3.15, 7.4.13, and 7.5.4 (Bug #23709232).

For more information, see Section 2.3.17.7, "Ndb_cluster_connection::get_next_ndb_object()".

2.3.17.11 Ndb_cluster_connection::set_auto_reconnect()

Description. An API node that is disconnected from the cluster is forced to use a new connection object to reconnect, unless this behavior is overridden by setting `AutoReconnect = 1` in the `config.ini` file or calling this method with 1 as the input value. Calling the method with 0 for the value has the same effect as setting the `AutoReconnect` configuration parameter (also introduced in those NDB Cluster versions) to 0; that is, API nodes are forced to create new connections.

Important

When called, this method overrides any setting for `AutoReconnect` made in the `config.ini` file.

For more information, see Defining SQL and Other API Nodes in an NDB Cluster.

Signature.

```
void set_auto_reconnect
    (
      int value
    )
```

Parameters. A *`value`* of 0 or 1 which determines API node reconnection behavior. 0 forces API nodes to use new connections (`Ndb_cluster_connection` objects); 1 permits API nodes to re-use existing connections to the cluster.

Return value. *None*.

2.3.17.12 Ndb_cluster_connection::set_data_node_neighbour()

Description. Set data node neighbor of the connection, used for optimal placement of the transaction coordinator. This method be used after creating the `Ndb_cluster_connection`, but prior to starting any query threads. This is due to the fact that this method may change the internal state of the `Ndb_cluster_connection` shared by the threads using it. This state is not thread-safe; changing it can lead to non-optimal node selection at the time of the change.

You can use the `ndb_data_node_neighbour` server system variable to set a data node neighbor for an NDB Cluster SQL node.

This method was added in NDB 7.5.2.

Signature.

```
void set_data_node_neighbour
    (
      Uint32 neighbour_node
    )
```

Parameters. The ID of the node to be used as the neighbor.

Return value. *None*.

2.3.17.13 Ndb_cluster_connection::set_max_adaptive_send_time()

Description. Set the minimum time in milliseconds that is permit to lapse before the adaptive send mechanism forces all pending signals to be sent.

Signature.

```
void set_max_adaptive_send_time
    (
      Uint32 milliseconds
    )
```

Parameters. Wait time in milliseconds. The range is 0-10, with 10 being the default value.

Return value. *None*.

2.3.17.14 Ndb_cluster_connection::set_name()

Description. Sets a name for the connection. If the name is specified, it is reported in the cluster log.

Signature.

```
void set_name
    (
      const char* name
    )
```

Parameters. The *name* to be used as an identifier for the connection.

Return value. *None*.

2.3.17.15 Ndb_cluster_connection::set_num_recv_threads()

Description. Set the number of receiver threads bound to the CPU (or CPUs) determined using `set_recv_thread_cpu()` and with the threshold set by `set_recv_thread_activation_threshold()`.

This method should be invoked before trying to connect to any other nodes.

Signature.

```
int set_num_recv_threads
    (
      Uint32 num_recv_threads
    )
```

Parameters. The number of receive threads. The only supported value is `1`.

Return value. `-1` indicates an error; any other value indicates success.

2.3.17.16 Ndb_cluster_connection::set_optimized_node_selection()

Description. This method can be used to override the `connect()` method's default behavior as regards which node should be connected to first.

Signature.

```
void set_optimized_node_selection
    (
      int value
    )
```

Parameters. An integer *value*.

Return value. *None*.

2.3.17.17

Description. Set the level for activating the receiver thread bound by `set_recv_thread_cpu()`. Below this level, normal user threads are used to receive signals.

Signature.

```
int set_recv_thread_activation_threshold
    (
      Uint32 threshold
    )
```

Parameters. An integer *`threshold`* value. 16 or higher means that receive threads are never used as receivers. 0 means that the receive thread is always active, and that retains poll rights for its own exclusive use, effectively blocking all user threads from becoming receivers. In such cases care should be taken to ensure that the receive thread does not compete with the user thread for CPU resources; it is preferable for it to be locked to a CPU for its own exclusive use. The default is 8.

Return value. `-1` indicates an error; any other value indicates success.

2.3.17.18

Description. Set the CPU or CPUs to which the receiver thread should be bound. Set the level for activating the receiver thread as a receiver by invoking `set_recv_thread_activation_threshold()`. Unset the binding for this receiver thread by invoking `unset_recv_thread_cpu()`.

Signature.

```
int set_recv_thread_cpu
    (
      Uint16* cpuid_array,
      Uint32 array_len,
      Uint32 recv_thread_id = 0
    )
```

Parameters. This method takes three parameters, listed here:

- An array of one or more CPU IDs to which the receive thread should be bound
- The length of this array
- The thread ID of the receive thread to bind. The default value is `0`.

Return value. `-1` indicates an error; any other value indicates success.

2.3.17.19 Ndb_cluster_connection::set_timeout()

Description. Used to set a timeout for the connection, to limit the amount of time that we may block when connecting.

This method is actually a wrapper for the function `ndb_mgm_set_timeout()`. For more information, see Section 3.2.4.12, “ndb_mgm_set_timeout()”.

Signature.

```
int set_timeout
    (
      int timeout_ms
    )
```

Parameters. The length of the timeout, in milliseconds (*`timeout_ms`*). Currently, only multiples of 1000 are accepted.

Return value. 0 on success; any other value indicates failure.

2.3.17.20 ndb_cluster_connection::unlock_ndb_objects()

Description. This method undoes the effects of the `lock_ndb_objects()` method, making it possible to create new instances of `Ndb`. `unlock_ndb_objects()` should be called after you have finished retrieving `Ndb` objects using the `get_next_ndb_object()` method.

Signature.

```
void unlock_ndb_objects
    (
      void
    ) const
```

Parameters. *None*.

Return value. *None*.

This method was made `const` in NDB 7.3.15, 7.4.13, and 7.5.4 (Bug #23709232).

For more information, see Section 2.3.17.7, “Ndb_cluster_connection::get_next_ndb_object()”.

2.3.17.21

Description. Unset the CPU or CPUs to which the receiver thread was bound using `set_recv_thread_cpu()`.

Signature.

```
int unset_recv_thread_cpu
    (
      Uint32 recv_thread_id
    )
```

Parameters. The thread ID of the receiver thread to be unbound.

Return value. `-1` indicates an error; any other value indicates success.

2.3.17.22 Ndb_cluster_connection::wait_until_ready()

Description. This method waits until the requested connection with one or more data nodes is successful.

Signature.

```
int wait_until_ready
    (
      int timeoutBefore,
      int timeoutAfter
    )
```

Parameters. This method takes two parameters:

- *`timeoutBefore`* determines the number of seconds to wait until the first “live” node is detected. If this amount of time is exceeded with no live nodes detected, then the method immediately returns a negative value.
- *`timeoutAfter`* determines the number of seconds to wait after the first “live” node is detected for all nodes to become active. If this amount of time is exceeded without all nodes becoming active, then the method immediately returns a value greater than zero.

If this method returns `0`, then all nodes are “live”.

Return value. `wait_until_ready()` returns an `int`, whose value is interpreted as follows:

- `= 0`: All nodes are “live”.
- `> 0`: At least one node is “live” (however, it is not known whether *all* nodes are “live”).
- `< 0`: An error occurred.

2.3.18 The NdbBlob Class

This class represents a handle to a `BLOB` column and provides read and write access to `BLOB` column values. This object has a number of different states and provides several modes of access to `BLOB` data; these are also described in this section.

Parent class. *None*

Child classes. *None*

Description. This class has no public constructor. An instance of `NdbBlob` is created using the `NdbOperation::getBlobHandle()` method during the operation preparation phase. (See Section 2.3.25, “The NdbOperation Class”.) This object acts as a handle on a `BLOB` column.

BLOB Data Storage. `BLOB` data is stored in 2 locations:

- The header and inline bytes are stored in the blob attribute.
- The blob's data segments are stored in a separate table named `NDB$BLOB_`*`tid`*`_cid`, where *`tid`* is the table ID, and *`cid`* is the blob column ID.

The inline and data segment sizes can be set using the appropriate `Column` methods when the table is created. See Section 2.3.2, “The Column Class”, for more information about these methods.

Data Access Types. `NdbBlob` supports 3 types of data access: These data access types can be applied in combination, provided that they are used in the order given above.

- In the preparation phase, the `NdbBlob` methods `getValue()` and `setValue()` are used to prepare a read or write of a `BLOB` value of known size.
- Also in the preparation phase, `setActiveHook()` is used to define a routine which is invoked as soon as the handle becomes active.
- In the active phase, `readData()` and `writeData()` are used to read and write `BLOB` values having arbitrary sizes.

BLOB Operations. `BLOB` operations take effect when the next transaction is executed. In some cases, `NdbBlob` is forced to perform implicit execution. To avoid this, you should always operate on complete blob data segments.

Use `NdbTransaction::executePendingBlobOps()` to flush reads and writes, which avoids any execution penalty if no operations are pending. This is not necessary following execution of operations, or after the next scan result.

`NdbBlob` also supports reading post- or pre-blob data from events. The handle can be read after the next event on the main table has been retrieved. The data becomes available immediately. (See Section 2.3.21, “The NdbEventOperation Class”, for more information.)

BLOBs and NdbOperations. `NdbOperation` methods acting on `NdbBlob` objects have the following characteristics:.

- `NdbOperation::insertTuple()` must use `NdbBlob::setValue()` if the `BLOB` attribute is nonnullable.

- `NdbOperation::readTuple()` used with any lock mode can read but not write blob values.

  When the `LM_CommittedRead` lock mode is used with `readTuple()`, the lock mode is automatically upgraded to `LM_Read` whenever blob attributes are accessed.

- `NdbOperation::updateTuple()` can either overwrite an existing value using `NdbBlob::setValue()`, or update it during the active phase.

- `NdbOperation::writeTuple()` always overwrites blob values, and must use `NdbBlob::setValue()` if the `BLOB` attribute is nonnullable.

- `NdbOperation::deleteTuple()` creates implicit, nonaccessible `BLOB` handles.

- A scan with any lock mode can use its blob handles to read blob values but not write them.

  A scan using the `LM_Exclusive` lock mode can update row and blob values using `updateCurrentTuple()`; the operation returned must explicitly create its own blob handle.

  A scan using the `LM_Exclusive` lock mode can delete row values (and therefore blob values) using `deleteCurrentTuple()`; this create implicit nonaccessible blob handles.

- An operation which is returned by `lockCurrentTuple()` cannot update blob values.

Known Issues. The following are known issues or limitations encountered when working with `NdbBlob` objects:

- Too many pending `BLOB` operations can overflow the I/O buffers.

- The table and its `BLOB` data segment tables are not created atomically.

Methods. The following table lists the public methods of this class and the purpose or use of each method:

Method	Purpose / Use
`blobsFirstBlob()`	Gets the first blob in a list.
`blobsNextBlob()`	Gets the next blob in a list
`getBlobEventName()`	Gets a blob event name
`getBlobTableName()`	Gets a blob data segment's table name.
`getColumn()`	Gets a blob column.
`getLength()`	Gets the length of a blob, in bytes
`getNdbError()`	Gets an error (an `NdbError` object)
`getNdbOperation()`	Get a pointer to the operation (`NdbOperation` object) to which this `NdbBlob` object belonged when created.
`getNull()`	Checks whether a blob value is `NULL`
`getPos()`	Gets the current position for reading/writing
`getState()`	Gets the state of an `NdbBlob` object
`getValue()`	Prepares to read a blob value
`getVersion()`	Checks whether a blob is statement-based or event-based
`readData()`	Reads data from a blob
`setActiveHook()`	Defines a callback for blob handle activation
`setNull()`	Sets a blob to `NULL`
`setPos()`	Sets the position at which to begin reading/writing
`setValue()`	Prepares to insert or update a blob value

Method	Purpose / Use
`truncate()`	Truncates a blob to a given length
`writeData()`	Writes blob data

Note

`getBlobTableName()` and `getBlobEventName()` are static methods.

Tip

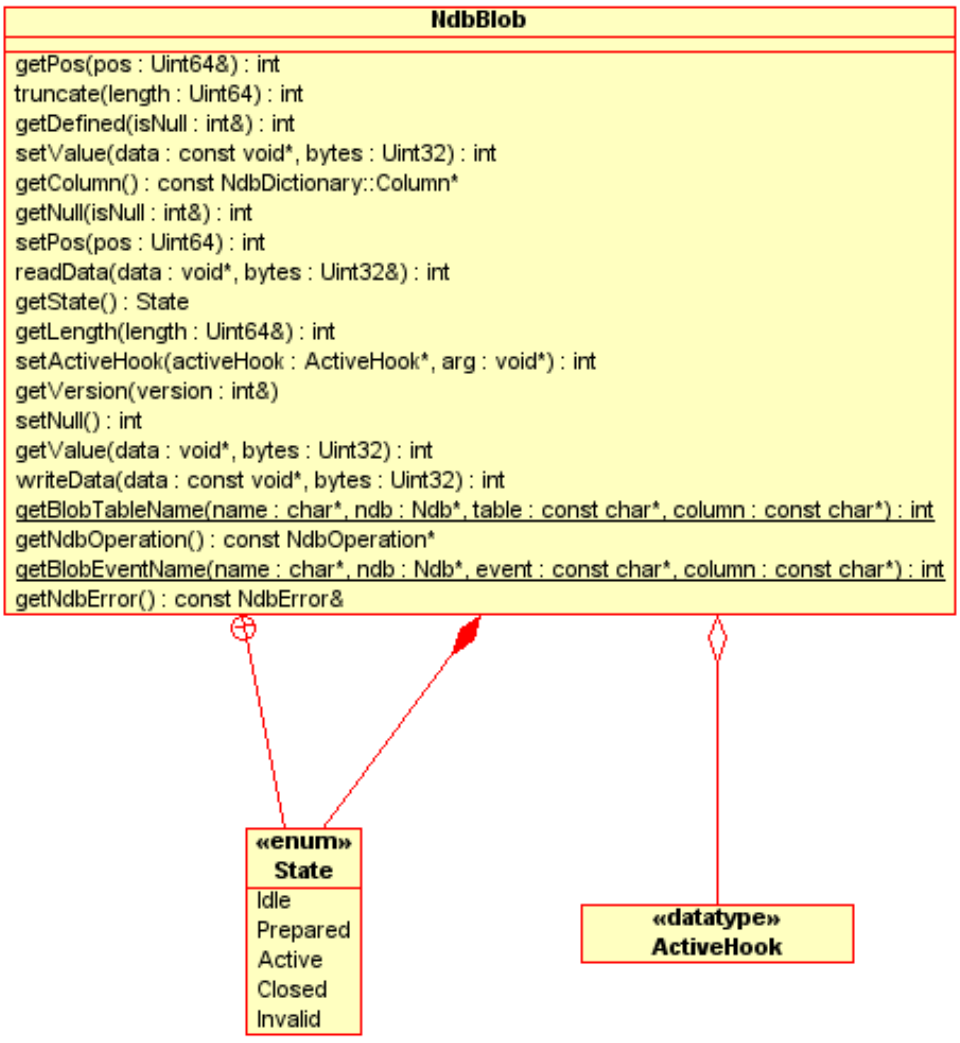
Most `NdbBlob` methods (nearly all of those whose return type is `int`) return `0` on success and `-1` in the event of failure.

Types. The public types defined by `NdbBlob` are shown here:

Type	Purpose / Use
`ActiveHook()`	Callback for `NdbBlob::setActiveHook()`
`State()`	Represents the states that may be assumed by the `NdbBlob`.

Class diagram. This diagram shows all the available methods and types of the `NdbBlob` class:

Figure 2.13 NdbBlob



2.3.18.1 NdbBlob::ActiveHook

ActiveHook is a data type defined for use as a callback for the `setActiveHook()` method. (See Section 2.3.18.16, “NdbBlob::setActiveHook()”.)

Definition. `ActiveHook` is a custom data type defined as shown here:

```
typedef int ActiveHook
    (
      NdbBlob* me,
      void*    arg
    )
```

Description. This is a callback for `NdbBlob::setActiveHook()`, and is invoked immediately once the prepared operation has been executed (but not committed). Any calls to `getValue()` or `setValue()` are performed first. The `BLOB` handle is active so `readData()` or `writeData()` can be used to manipulate the `BLOB` value. A user-defined argument is passed along with the `NdbBlob`. `setActiveHook()` returns a nonzero value in the event of an error.

2.3.18.2 NdbBlob::blobsFirstBlob()

Description. This method initialises a list of blobs belonging to the current operation and returns the first blob in the list.

Signature.

```
NdbBlob* blobsFirstBlob
    (
      void
    )
```

Parameters. None.

Return value. A pointer to the desired blob.

2.3.18.3 NdbBlob::blobsNextBlob()

Description. Use the method to obtain the next in a list of blobs that was initialised using `blobsFirstBlob()`. See Section 2.3.18.2, “NdbBlob::blobsFirstBlob()”.

Signature.

```
NdbBlob* blobsNextBlob
    (
      void
    )
```

Parameters. None.

Return value. A pointer to the desired blob.

2.3.18.4 NdbBlob::getBlobEventName()

Description. This method gets a blob event name. The blob event is created if the main event monitors the blob column. The name includes the main event name.

Signature.

```
static int getBlobEventName
    (
      char*       name,
      Ndb*        ndb,
```

```
      const char* event,
      const char* column
    )
```

Parameters. This method takes the four parameters listed here:

- *name*: The name of the blob event.
- *ndb*: The relevant Ndb object.
- *event*: The name of the main event.
- *column*: The blob column.

Return value. 0 on success, -1 on failure.

2.3.18.5 NdbBlob::getBlobTableName()

Description. This method gets the blob data segment table name.

Note

This method is generally of use only for testing and debugging purposes.

Signature.

```
static int getBlobTableName
    (
      char*       name,
      Ndb*        ndb,
      const char* table,
      const char* column
    )
```

Parameters. This method takes the four parameters listed here:

- *name*: The name of the blob data segment table.
- *ndb*: The relevant Ndb object.
- *table*: The name of the main table.
- *column*: The blob column.

Return value. Returns 0 on success, -1 on failure.

2.3.18.6 NdbBlob::getColumn()

Description. Use this method to get the BLOB column to which the NdbBlob belongs.

Signature.

```
const Column* getColumn
    (
      void
    )
```

Parameters. None.

Return value. A Column object. (See Section 2.3.2, “The Column Class”.)

2.3.18.7 NdbBlob::getLength()

Description. This method gets the blob's current length in bytes.

Signature.

```
int getLength
    (
      Uint64& length
    )
```

Parameters. A reference to the length.

Return value. The blob's length in bytes. For a `NULL` blob, this method returns `0`. to distinguish between a blob whose length is `0` blob and one which is `NULL`, use the `getNull()` method.

2.3.18.8 NdbBlob::getNull()

Description. This method checks whether the blob's value is `NULL`.

Signature.

```
int getNull
    (
      int& isNull
    )
```

Parameters. A reference to an integer *`isNull`*. Following invocation, this parameter has one of the following values, interpreted as shown here:

- `-1`: The blob is undefined. If this is a nonevent blob, this result causes a state error.
- `0`: The blob has a nonnull value.
- `1`: The blob's value is `NULL`.

Return value. *None*.

2.3.18.9 NdbBlob::getNdbError()

Description. Use this method to obtain an error object. The error may be blob-specific or may be copied from a failed implicit operation. The error code is copied back to the operation unless the operation already has a nonzero error code.

Signature.

```
const NdbError& getNdbError
    (
      void
    ) const
```

Parameters. None.

Return value. An `NdbError` object. See Section 2.3.20, “The NdbError Structure”.

2.3.18.10 NdbBlob::getNdbOperation()

Description. This method can be used to find the operation with which the handle for this `NdbBlob` is associated.

Signature.

```
const NdbOperation* getNdbOperation
    (
      void
    ) const
```

Parameters. *None*.

Return value. A pointer to an operation.

Important

The operation referenced by the pointer retruned by this method may be represented by either an `NdbOperation` or `NdbScanOperation` object.

See Section 2.3.25, “The NdbOperation Class”, and Section 2.3.29, “The NdbScanOperation Class”, for more information.

2.3.18.11 NdbBlob::getPos()

Description. This method gets the current read/write position in a blob.

Signature.

```
int getPos
    (
      Uint64& pos
    )
```

Parameters. One parameter, a reference to the position.

Return value. Returns `0` on success, or `-1` on failure. (Following a successful invocation, *`pos`* will hold the current read/write position within the blob, as a number of bytes from the beginning.)

2.3.18.12 NdbBlob::getState()

Description. This method gets the current state of the `NdbBlob` object for which it is invoked. Possible states are described in Section 2.3.18.20, “NdbBlob::State”.

Signature.

```
State getState
    (
      void
    )
```

Parameters. None.

Return value. A `State` value. For possible values, see Section 2.3.18.20, “NdbBlob::State”.

2.3.18.13 NdbBlob::getValue()

Description. Use this method to prepare to read a blob value; the value is available following invocation. Use `getNull()` to check for a `NULL` value; use `getLength()` to get the actual length of the blob, and to check for truncation. `getValue()` sets the current read/write position to the point following the end of the data which was read.

Signature.

```
int getValue
    (
      void*  data,
      Uint32 bytes
```

```
    )
```

Parameters. This method takes two parameters. The first of these is a pointer to the *`data`* to be read; the second is the number of *`bytes`* to be read.

Return value. `0` on success, `-1` on failure.

2.3.18.14 NdbBlob::getVersion()

Description. This method is used to distinguish whether a blob operation is statement-based or event-based.

Signature.

```
void getVersion
    (
      int& version
    )
```

Parameters. This method takes a single parameter, an integer reference to the blob version (operation type).

Return value. One of the following three values:

- `-1`: This is a “normal” (statement-based) blob.
- `0`: This is an event-operation based blob, following a change in its data.
- `1`: This is an event-operation based blob, prior to any change in its data.

Note

`getVersion()` is always successful, assuming that it is invoked as a method of a valid `NdbBlob` instance.

2.3.18.15 NdbBlob::readData()

Description. This method is used to read data from a blob.

Signature.

```
int readData
    (
      void*     data,
      Uint32&  bytes
    )
```

Parameters. `readData()` accepts a pointer to the *`data`* to be read, and a reference to the number of *`bytes`* read.

Return value. Returns `0` on success, `-1` on failure. Following a successful invocation, *`data`* points to the data that was read, and *`bytes`* holds the number of bytes read.

2.3.18.16 NdbBlob::setActiveHook()

Description. This method defines a callback for blob handle activation. The queue of prepared operations will be executed in no-commit mode up to this point; then, the callback is invoked. For additional information, see Section 2.3.18.1, “NdbBlob::ActiveHook”.

Signature.

```
int setActiveHook
    (
      ActiveHook*  activeHook,
      void*        arg
    )
```

Parameters. This method requires the two parameters listed here:

- A pointer to an `ActiveHook` value; this is a callback as explained in Section 2.3.18.1, “NdbBlob::ActiveHook”.
- A pointer to `void`, for any data to be passed to the callback.

Return value. `0` on success, `-1` on failure.

2.3.18.17 NdbBlob::setNull()

Description. This method sets the value of a blob to `NULL`.

Signature.

```
int setNull
    (
      void
    )
```

Parameters. None.

Return value. `0` on success; `-1` on failure.

2.3.18.18 NdbBlob::setPos()

Description. This method sets the position within the blob at which to read or write data.

Signature.

```
int setPos
    (
      Uint64 pos
    )
```

Parameters. The setPos() method takes a single parameter *`pos`* (an unsigned 64-bit integer), which is the position for reading or writing data. The value of *`pos`* must be between `0` and the blob's current length.

Important

“Sparse” blobs are not supported in the NDB API; in other words, there can be no unused data positions within a blob.

Return value. `0` on success, `-1` on failure.

2.3.18.19 NdbBlob::setValue()

Description. This method is used to prepare for inserting or updating a blob value. Any existing blob data that is longer than the new data is truncated. The data buffer must remain valid until the operation has been executed. `setValue()` sets the current read/write position to the point following the end of the data. You can set *`data`* to a null pointer (`0`) in order to create a `NULL` value.

Signature.

```
int setValue
    (
      const void*  data,
      Uint32       bytes
    )
```

Parameters. This method takes the two parameters listed here:

- The *`data`* that is to be inserted or used to overwrite the blob value.
- The number of *`bytes`*—that is, the length—of the *`data`*.

Return value. `0` on success, `-1` on failure.

2.3.18.20 NdbBlob::State

This is an enumerated data type which represents the possible states of an `NdbBlob` instance.

Description. An `NdbBlob` may assume any one of these states

Enumeration values. Possible values are shown, along with descriptions, in the following table:

Value	Description
`Idle`	The `NdbBlob` has not yet been prepared for use with any operations.
`Prepared`	This is the state of the `NdbBlob` prior to operation execution.
`Active`	This is the `BLOB` handle's state following execution or the fetching of the next result, but before the transaction is committed.
`Closed`	This state occurs after the transaction has been committed.
`Invalid`	This follows a rollback or the close of a transaction.

2.3.18.21 NdbBlob::truncate()

Description. This method is used to truncate a blob to a given length.

Signature.

```
int truncate
    (
      Uint64 length = 0
    )
```

Parameters. `truncate()` takes a single parameter which specifies the new *`length`* to which the blob is to be truncated. This method has no effect if *`length`* is greater than the blob's current length (which you can check using `getLength()`).

Return value. `0` on success, `-1` on failure.

2.3.18.22 NdbBlob::writeData()

Description. This method is used to write data to an `NdbBlob`. After a successful invocation, the read/write position will be at the first byte following the data that was written to the blob.

Note

A write past the current end of the blob data extends the blob automatically.

Signature.

```
int writeData
    (
      const void*  data,
      Uint32       bytes
    )
```

Parameters. This method takes two parameters, a pointer to the *`data`* to be written, and the number of *`bytes`* to write.

Return value. `0` on success, `-1` on failure.

2.3.19 The NdbDictionary Class

This class provides meta-information about database objects, such as tables, columns, and indexes.

While the preferred method of database object creation and deletion is through the MySQL Server, `NdbDictionary` also permits the developer to perform these tasks through the NDB API.

Parent class. *None*

Child classes. `Dictionary`, `Column`, `Object`

Description. This is a data dictionary class that supports enquiries about tables, columns, and indexes. It also provides ways to define these database objects and to remove them. Both sorts of functionality are supplied using inner classes that model these objects. These include the following inner classes:

- `Table` for working with tables
- `Column` for creating table columns
- `Index` for working with secondary indexes
- `Dictionary` for creating database objects and making schema enquiries
- `Event` for working with events in the cluster.

Additional `Object` subclasses model the tablespaces, logfile groups, datafiles, and undofiles required for working with NDB Cluster Disk Data tables (introduced in MySQL 5.1), as well as foreign keys (NDB Cluster 7.3 and later).

Warning

Tables and indexes created using `NdbDictionary` cannot be viewed from the MySQL Server.

Dropping indexes through the NDB API that were created originally from an NDB Cluster causes inconsistencies. It is possible that a table from which one or more indexes have been dropped using the NDB API will no longer be usable by MySQL following such operations. In this event, the table must be dropped, and then re-created using MySQL to make it accessible to MySQL once more.

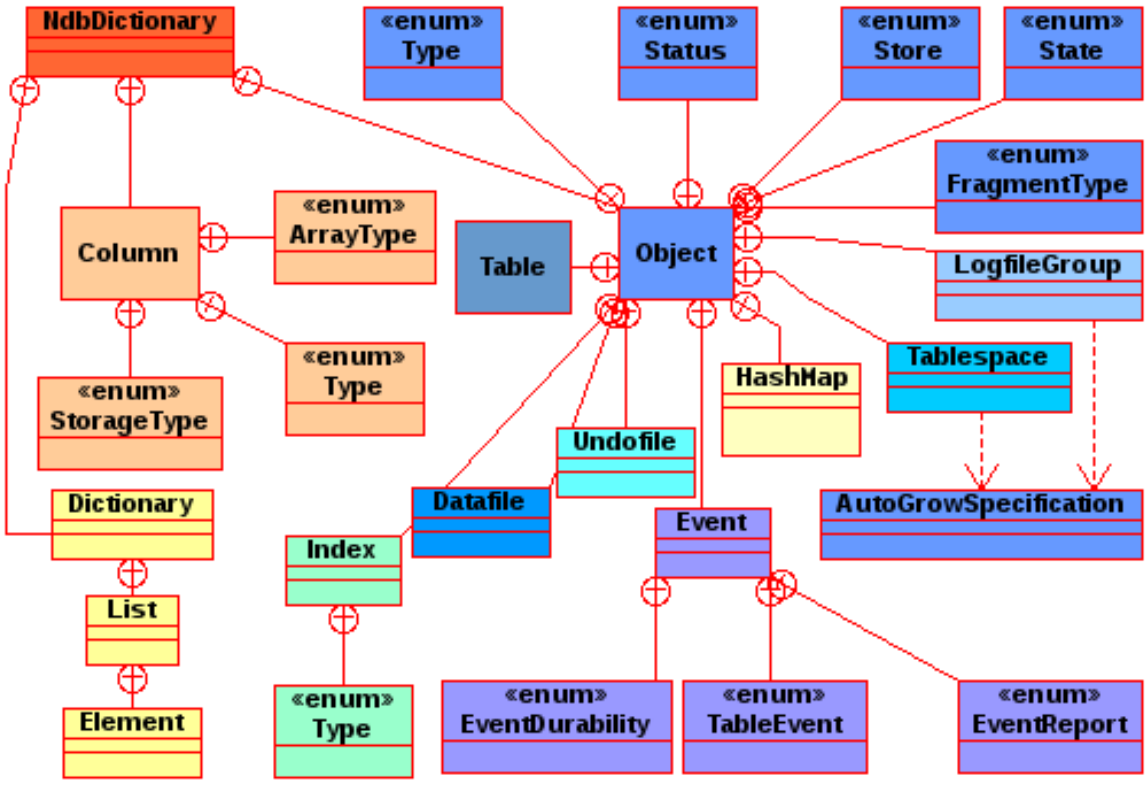
Methods. `NdbDictionary` itself has no public instance methods, only static methods that are used for working with `NdbRecord` objects. Operations not using `NdbRecord` are accomplished by means of `NdbDictionary` subclass instance methods. The following table lists the public methods of `NdbDictionary` and the purpose or use of each method:

Method	Purpose / Use
`getEmptyBitmask()`	Returns an empty column presence bitmask which can be used with `NdbRecord`

Method	Purpose / Use
`getFirstAttrId()`	Get the first attribute ID specified by a given `NdbRecord` object
`getRecordIndexName()`	Gets the name of the index object referred to by an NdbRecord
`getRecordRowLength()`	Get the number of bytes needed to store one row of data using a given `NdbRecord`
`getRecordTableName()`	Gets the name of the table object referred to by an NdbRecord
`getRecordType()`	Gets the RecordType of an NdbRecord
`getValuePtr()`	Returns a pointer to the beginning of stored data specified by attribute ID, using `NdbRecord`
`isNull()`	Show whether the null bit for a column is true or false
`setNull()`	Set a column's null bit

NdbDictionary Subclass Hierarchy. This diagram shows the hierarchy made up of the `NdbDictionary` class, its subclasses, and their enumerated data types:

Figure 2.14 NdbDictionary subclasses



Note

For the numeric equivalents to enumerations of `NdbDictionary` subclasses, see the file `/storage/ndb/include/ndbapi/NdbDictionary.hpp` in the NDB Cluster source tree.

2.3.19.1 NdbDictionary::getEmptyBitmask()

Description. Returns an empty column presence bitmask which can be used with any `NdbRecord` to specify that no `NdbRecord` columns are to be included in the operation.

Signature.

```
static const unsigned char* getEmptyBitmask
    (
      void
    )
```

Parameters. *None*.

Return value. An empty bitmask.

2.3.19.2 NdbDictionary::getFirstAttrId()

Description. Get the first attribute ID specified by an `NdbRecord` object. Returns `false` if no attribute ID is specified.

Signature.

```
static bool getFirstAttrId
    (
      const NdbRecord* record,
      Uint32& firstAttrId
    )
```

Parameters. A pointer to an `NdbRecord` and a reference to the attribute (*`firstAttrID`*).

Return value. Boolean `false`, when no attribute ID can be obtained.

2.3.19.3 NdbDictionary::getNextAttrId()

Description. Get the next attribute ID specified by an `NdbRecord` object following the attribute ID passed in. Returns `false` when there are no more attribute IDs to be returned.

Signature.

```
static bool getNextAttrId
    (
      const NdbRecord* record,
      Uint32& attrId
    )
```

Parameters. A pointer to an `NdbRecord` and a reference to an attribute ID.

Return value. Boolean `false`, when no attribute ID can be obtained.

2.3.19.4 NdbDictionary::getNullBitOffset()

Description. Get the offset of the given attribute ID's null bit from the start of the `NdbRecord` row. Returns `false` if the attribute ID is not present.

Signature.

```
static bool getNullBitOffset
    (
      const NdbRecord* record,
      Uint32 attrId,
      Uint32& bytes,
      Uint32& bit
    )
```

Parameters. An `NdbRecord` *`record`* in which to get the null bit offset of the given attribute ID (*`attrId`*). The offset is expressed as a number of bytes (*`bytes`*) plus a number of bits within the last byte (*`bit`*).

Return value. Boolean `false`, if the attribute with the given ID is not present.

2.3.19.5 NdbDictionary::getOffset()

Description. Get the offset of the given attribute ID's storage from the start of the `NdbRecord` row. Returns `false` if the attribute id is not present

Signature.

```
static bool getOffset
    (
      const NdbRecord* record,
      Uint32 attrId,
      Uint32& offset
    )
```

Parameters. The *`offset`* of the given attribute ID's storage from the start of the `NdbRecord` row.

Return value. Boolean `false`, if no attribute ID can be found.

2.3.19.6 NdbDictionary::getRecordIndexName()

Description. Get the name of the `Index` object that the `NdbRecord` refers to.

If the `NdbRecord` object is not an `IndexAccess` `NdbRecord`, the method returns null.

Signature.

```
static const char* getRecordIndexName
    (
      const NdbRecord* record
    )
```

Parameters. A pointer to the `NdbRecord` for which to get the name.

Return value. The name, if any. Otherwise, or if the `NdbRecord` object is not of the `IndexAccess` type, this method returns null.

2.3.19.7 NdbDictionary::getRecordRowLength()

Description. Get the number of bytes needed to store one row of data laid out as described by the `NdbRecord` structure passed in to this method.

Signature.

```
static Uint32 getRecordRowLength
    (
      const NdbRecord* record
    )
```

Parameters. An `NdbRecord` object.

Return value. The number of bytes needed per row.

2.3.19.8 NdbDictionary::getRecordTableName()

Description. Return the name of the table object that the `NdbRecord` refers to. This method returns null if the record is not a `TableAccess`.

Signature.

```
static const char* getRecordTableName
    (
      const NdbRecord* record
    )
```

Parameters. The *record* (NdbRecord object) for which to get the table name.

Return value. The name of the table, or null if the NdbRecord object' type is not TableAccess.

2.3.19.9 NdbDictionary::getRecordType()

Description. Return the type of the NdbRecord object passed.

Signature.

```
static RecordType getRecordType
    (
      const NdbRecord* record
    )
```

Parameters. An NdbRecord object.

Return value. The RecordType of the NdbRecord (IndexAccess or TableAccess).

2.3.19.10 NdbDictionary::getValuePtr()

Description. Returns a pointer to the beginning of stored data specified by attribute ID, by looking up the offset of the column stored in the NdbRecord object and returning the sum of the row position and the offset.

Note

This method provides both row-const and non-row-const versions.

Signature.

```
static const char* getValuePtr
    (
      const NdbRecord* record,
      const char* row,
      Uint32 attrId
    )

static char* getValuePtr
    (
      const NdbRecord* record,
      char* row,
      Uint32 attrId
    )
```

Parameters. A pointer to an NdbRecord object describing the row format, a pointer to the start of the row data (const in the const version of this method), and the attribute ID of the column,

Return value. A pointer to the start of the attribute in the row. This is null if the attribute is not part of the NdbRecord definition.

2.3.19.11 NdbDictionary::isNull()

Description. Indicate whether the null bit for the given column is set to true or false. The location of the null bit in relation to the row pointer is obtained from the passed NdbRecord object. If the

column is not nullable, or if the column is not part of the `NdbRecord` definition, the method returns `false`.

Signature.

```
static bool isNull
    (
      const NdbRecord* record,
      const char* row,
      Uint32 attrId
    )
```

Parameters. A pointer to an `NdbRecord` object describing the row format, a pointer to the start of the row data, and the attribute ID of the column to check.

Return value. Boolean `true` if the attribute ID exists in this `NdbRecord`, is nullable, and this row's null bit is set; otherwise, Boolean `false`.

2.3.19.12 NdbDictionary::setNull()

Description. Set the null bit for the given column to the supplied value. The offset for the null bit is obtained from the passed `NdbRecord` object. If the attribute ID is not part of the `NdbRecord`, or if it is not nullable, this method returns an error (-1).

Signature.

```
static int setNull
    (
      const NdbRecord* record,
      char* row,
      Uint32 attrId,
      bool value
    )
```

Parameters. A pointer to the *`record`* (`NdbRecord` object) describing the row format; a pointer to the start of the *`row`* data; the attribute ID of the column (*`attrId`*); and the *`value`* to set the null bit to (`true` or `false`).

Return value. Returns 0 on success; returns -1 if the *`attrId`* is not part of the *`record`*, or is not nullable.

2.3.20 The NdbError Structure

This section discusses the `NdbError` data structure, which contains status and other information about errors, including error codes, classifications, and messages.

Description. An `NdbError` consists of six parts, listed here, of which one is deprecated:

1. *Error status*: This describes the impact of an error on the application, and reflects what the application should do when the error is encountered.

   The error status is described by a value of the `Status` type. See Section 2.3.20.2, “NdbError::Status”, for possible `Status` values and how they should be interpreted.

2. *Error classification*: This represents a logical error type or grouping.

   The error classification is described by a value of the `Classification` type. See Section 2.3.20.1, “NdbError::Classification”, for possible classifications and their interpretation. Additional information is provided in Section 2.4.4, “NDB Error Classifications”.

3. *Error code*: This is an NDB API internal error code which uniquely identifies the error.

Important

It is *not* recommended to write application programs which are dependent on specific error codes. Instead, applications should check error status and classification. More information about errors can also be obtained by checking error messages and (when available) error detail messages. However—like error codes—these error messages and error detail messages are subject to change.

A listing of current error codes, broken down by classification, is provided in Section 2.4.2, "NDB Error Codes: by Type". This listing is updated with new NDB Cluster releases. You can also check the file `storage/ndb/src/ndbapi/ndberror.c` in the NDB Cluster sources.

4. *MySQL Error code*: This is the corresponding MySQL Server error code. MySQL error codes are not discussed in this document; please see Server Error Codes and Messages, in the MySQL Manual, for information about these.

5. *Error message*: This is a generic, context-independent description of the error.

6. *Error details*: This can often provide additional information (not found in the error message) about an error, specific to the circumstances under which the error is encountered. However, it is not available in all cases.

   Where not specified, the error detail message is `NULL`.

Note

This property is deprecated and scheduled for eventual removal. For obtaining error details, you should use the `Ndb::getNdbErrorDetail()` method instead.

Important

Specific NDB API error codes, messages, and detail messages are subject to change without notice.

Definition. The NdbError structure contains the following members, whose types are as shown here:

- `Status status`: The error status.
- `Classification classification`: The error type (classification).
- `int code`: The NDB API error code.
- `int mysql_code`: The MySQL error code.
- `const char* message`: The error message.
- `char* details`: The error detail message.

Note

`details` is deprecated and scheduled for eventual removal. You should use the `Ndb::getNdbErrorDetail()` method instead. (Bug #48851)

See the *Description* for more information about these members and their types.

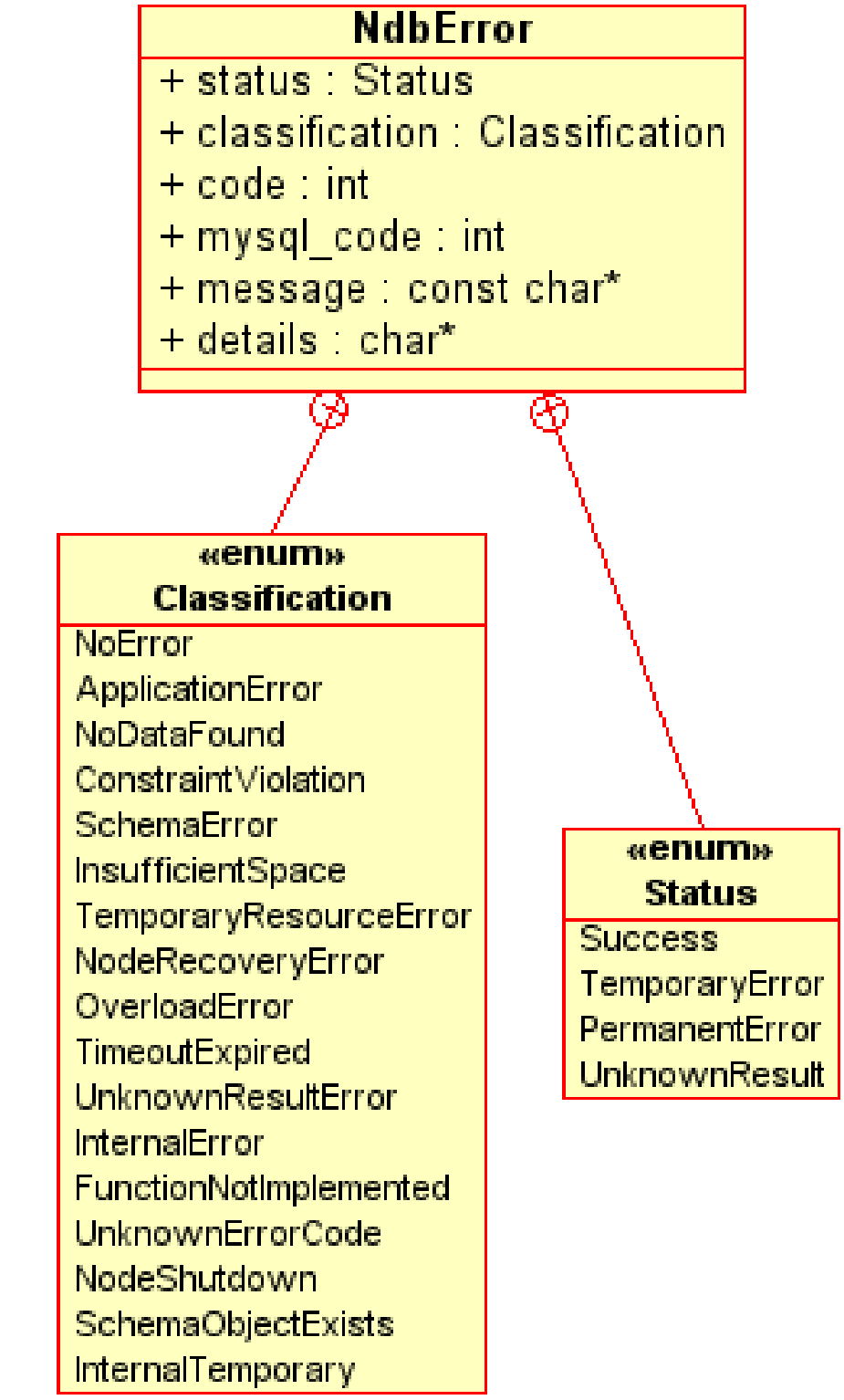
Types. `NdbError` defines the two data types listed here:

- `Classification`: The type of error or the logical grouping to which the error belongs.

- `Status`: The error status.

Structure Diagram. This diagram shows all the available members and types of the `NdbError` structure:

Figure 2.15 NdbError



2.3.20.1 NdbError::Classification

Description. This type describes the type of error, or the logical group to which it belongs.

Enumeration values. Possible values are shown, along with descriptions, in the following table:

Value	Description
`NoError`	Indicates success (no error occurred)
`ApplicationError`	An error occurred in an application program
`NoDataFound`	A read operation failed due to one or more missing records.
`ConstraintViolation`	A constraint violation occurred, such as attempting to insert a tuple having a primary key value already in use in the target table.
`SchemaError`	An error took place when trying to create or use a table.
`InsufficientSpace`	There was insufficient memory for data or indexes.
`TemporaryResourceError`	This type of error is typically encountered when there are too many active transactions.
`NodeRecoveryError`	This is a temporary failure which was likely caused by a node recovery in progress, some examples being when information sent between an application and `NDB` is lost, or when there is a distribution change.
`OverloadError`	This type of error is often caused when there is insufficient logfile space.
`TimeoutExpired`	A timeout, often caused by a deadlock.
`UnknownResultError`	It is not known whether a transaction was committed.
`InternalError`	A serious error has occurred in `NDB` itself.
`FunctionNotImplemented`	The application attempted to use a function which is not yet implemented.
`UnknownErrorCode`	This is seen where the `NDB` error handler cannot determine the correct error code to report.
`NodeShutdown`	This is caused by a node shutdown.
`SchemaObjectExists`	The application attempted to create a schema object that already exists.
`InternalTemporary`	A request was sent to a node other than the master.

Note

Related information specific to certain error conditions may be found in Section 2.4.2, “NDB Error Codes: by Type”, and in Section 2.4.4, “NDB Error Classifications”.

2.3.20.2 NdbError::Status

Description. This type is used to describe an error's status.

Enumeration values. Possible values are shown, along with descriptions, in the following table:

Value	Description
`Success`	No error has occurred
`TemporaryError`	A temporary and usually recoverable error; the application should retry the operation giving rise to the error
`PermanentError`	Permanent error; not recoverable

Value	Description
`UnknownResult`	The operation's result or status is unknown

Note

Related information specific to certain error conditions may be found in Section 2.4.4, “NDB Error Classifications”.

2.3.21 The NdbEventOperation Class

This section describes the `NdbEventOperation` class, which is used to monitor changes (events) in a database. It provides the core functionality used to implement NDB Cluster Replication.

Parent class. *None*

Child classes. *None*

Description. `NdbEventOperation` represents a database event.

Creating an Instance of NdbEventOperation. This class has no public constructor or destructor. Instead, instances of `NdbEventOperation` are created as the result of method calls on `Ndb` and `NdbDictionary` objects, subject to the following conditions:

1. There must exist an event which was created using `Dictionary::createEvent()`. This method returns an instance of the `Event` class.
2. An `NdbEventOperation` object is instantiated using `Ndb::createEventOperation()`, which acts on an instance of `Event`.

An instance of this class is removed by invoking `Ndb::dropEventOperation`.

Tip

A detailed example demonstrating creation and removal of event operations is provided in Section 2.5.8, “NDB API Event Handling Example”.

Known Issues. The following issues may be encountered when working with event operations in the NDB API:

- The maximum number of active `NdbEventOperation` objects is currently fixed at compile time at 2 * `MaxNoOfTables`.
- Currently, all `INSERT`, `DELETE`, and `UPDATE` events—as well as all attribute changes—are sent to the API, even if only some attributes have been specified. However, these are hidden from the user and only relevant data is shown after calling `Ndb::nextEvent()`.

  Note that false exits from `Ndb::pollEvents()` may occur, and thus the following `nextEvent()` call returns zero, since there was no available data. In such cases, simply call `pollEvents()` again.

  See Section 2.3.16.26, “Ndb::pollEvents()”, and Section 2.3.16.24, “Ndb::nextEvent()”.
- Event code does *not* check the table schema version. When a table is dropped, make sure that you drop any associated events.
- If you have received a complete epoch, events from this epoch are not re-sent, even in the event of a node failure. However, if a node failure has occurred, subsequent epochs may contain duplicate events, which can be identified by duplicated primary keys.

  In the NDB Cluster replication code, duplicate primary keys on `INSERT` operations are normally handled by treating such inserts as `REPLACE` operations.

Tip

To view the contents of the system table containing created events, you can use the `ndb_select_all` utility as shown here:

```
ndb_select_all -d sys 'NDB$EVENTS_0'
```

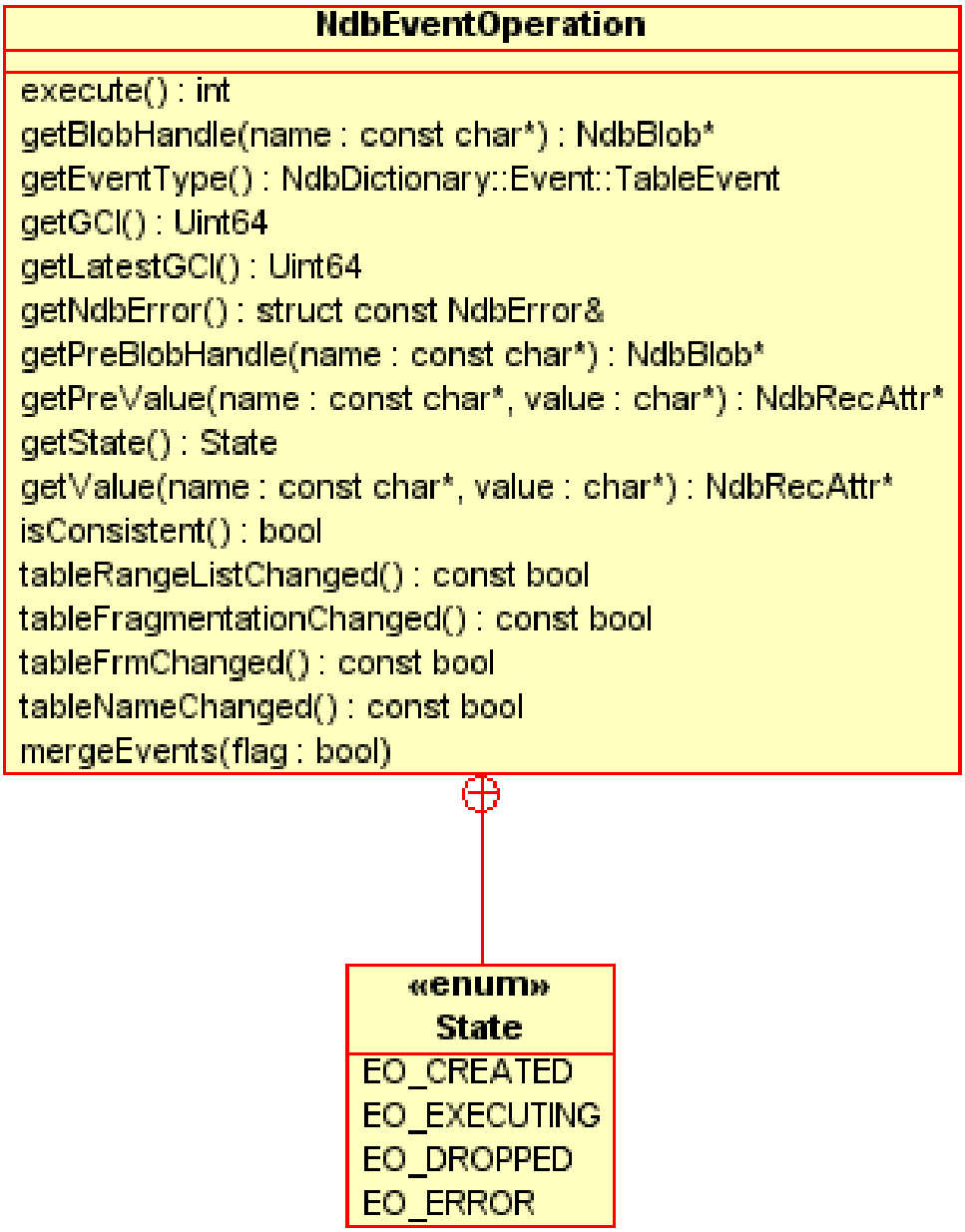
Methods. The following table lists the public methods of this class and the purpose or use of each method:

Method	Purpose / Use
`clearError()`	Clears the most recent error. Deprecated in NDB 7.4.3.
`execute()`	Activates the `NdbEventOperation`
`getBlobHandle()`	Gets a handle for reading blob attributes
`getEpoch()`	Retrieves the epoch for the event data most recently retrieved. Added in NDB 7.4.3.
`getEventType()`	Gets the event type. Deprecated in NDB 7.4.3.
`getEventType2()`	Gets the event type. Added in NDB 7.4.3.
`getGCI()`	Retrieves the GCI of the most recently retrieved event. Deprecated in NDB 7.4.3.
`getLatestGCI()`	Retrieves the most recent GCI (whether or not the corresponding event has been retrieved). Deprecated in NDB 7.4.3.
`getNdbError()`	Gets the most recent error
`getPreBlobHandle()`	Gets a handle for reading the previous blob attribute
`getPreValue()`	Retrieves an attribute's previous value
`getState()`	Gets the current state of the event operation
`getValue()`	Retrieves an attribute value
`hasError()`	Whether an error has occurred as part of this operation. Deprecated in NDB 7.4.3.
`isConsistent()`	Detects event loss caused by node failure. Deprecated in NDB 7.4.3.
`isEmptyEpoch()`	Detects an empty epoch. Added in NDB 7.4.3.
`isErrorEpoch()`	Detects an error epoch, and retrieves the error if there is one. Added in NDB 7.4.3.
`isOverrun()`	Whether event loss has taken place due to a buffer overrun. Deprecated in NDB 7.4.3.
`mergeEvents()`	Makes it possible for events to be merged
`tableFragmentationChanged()`	Checks to see whether the fragmentation for a table has changed
`tableFrmChanged()`	Checks to see whether a table `.FRM` file has changed
`tableNameChanged()`	Checks to see whether the name of a table has changed
`tableRangeListChanged()`	Checks to see whether a table range partition list name has changed

Types. `NdbEventOperation` defines one enumerated type, the `State` type.

Class diagram. This diagram shows all the available members of the `NdbEventOperation` class:

Figure 2.16 NdbEventOperation



2.3.21.1 NdbEventOperation::clearError()

Description. Clears the error most recently associated with this event operation.

This method is deprecated in NDB 7.4.3, and is subject to removal in a future release.

Signature.

```
void clearError
    (
      void
    )
```

Parameters. *None*.

Return value. *None*.

2.3.21.2 NdbEventOperation::execute()

Description. Activates the `NdbEventOperation`, so that it can begin receiving events. Changed attribute values may be retrieved after `Ndb::nextEvent()` has returned a value other than `NULL`.

One of `getValue()`, `getPreValue()`, `getBlobValue()`, or `getPreBlobValue()` must be called before invoking `execute()`.

Important

Before attempting to use this method, you should have read the explanations provided in Section 2.3.16.24, “Ndb::nextEvent()”, and Section 2.3.21.13, “NdbEventOperation::getValue()”. Also see Section 2.3.21, “The NdbEventOperation Class”.

Signature.

```
int execute
    (
      void
    )
```

Parameters. *None*.

Return value. This method returns `0` on success and `-1` on failure.

2.3.21.3 NdbEventOperation::getBlobHandle()

Description. This method is used in place of `getValue()` for blob attributes. The blob handle (`NdbBlob`) returned by this method supports read operations only.

Note

To obtain the previous value for a blob attribute, use `getPreBlobHandle()`.

Signature.

```
NdbBlob* getBlobHandle
    (
      const char* name
    )
```

Parameters. The *`name`* of the blob attribute.

Return value. A pointer to an `NdbBlob` object.

2.3.21.4 NdbEventOperation::getEpoch()

Description. Gets the epoch for the latest event data retrieved.

Added in NDB 7.4.3, this method supersedes `getGCI()`, which is now deprecated and subject to removal in a future NDB Cluster release.

Signature.

```
Uint64 getEpoch
  (
    void
```

```
  ) const
```

Parameters. *None*.

Return value. An epoch number (an integer).

2.3.21.5 NdbEventOperation::getEventType()

Description. This method is used to obtain the event's type (`TableEvent`).

This method is deprecated in NDB 7.4.3, and is subject to removal in a future release. In NDB 7.4.3 and later, you should use `getEventType2()` instead.

Signature.

```
NdbDictionary::Event::TableEvent getEventType
    (
      void
    ) const
```

Parameters. *None*.

Return value. A `TableEvent` value.

2.3.21.6 NdbEventOperation::getEventType2()

Description. This method is used to obtain the event's type (`TableEvent`).

Added in NDB 7.4.3, this method supersedes `getEventType()`, which is now deprecated and subject to removal in a future NDB Cluster release.

Signature.

```
NdbDictionary::Event::TableEvent getEventType2
    (
      void
    ) const
```

Parameters. *None*.

Return value. A `TableEvent` value.

2.3.21.7 NdbEventOperation::getGCI()

Description. This method retrieves the GCI for the most recently retrieved event.

This method is deprecated in NDB 7.4.3, and is subject to removal in a future release. In NDB 7.4.3 and later, you should use `getEpoch()` instead.

Signature.

```
Uint64 getGCI
    (
      void
    ) const
```

Parameters. *None*.

Return value. The global checkpoint index of the most recently retrieved event (an integer).

2.3.21.8 NdbEventOperation::getLatestGCI()

Description. This method retrieves the most recent GCI.

This method returns the latest epoch number.

This method is deprecated in NDB 7.4.3, and is subject to removal in a future release. In NDB 7.4.3 and later, you should use `Ndb::getHighestQueuedEpoch()` instead.

Note

The GCI obtained using this method is not necessarily associated with an event.

Signature.

```
Uint64 getLatestGCI
    (
      void
    ) const
```

Parameters. *None*.

Return value. The index of the latest global checkpoint, an integer.

2.3.21.9 NdbEventOperation::getNdbError()

Description. This method retrieves the most recent error.

Signature.

```
const struct NdbError& getNdbError
    (
      void
    ) const
```

Parameters. *None*.

Return value. A reference to an `NdbError` structure.

2.3.21.10 NdbEventOperation::getPreBlobHandle()

Description. This function is the same as `getBlobHandle()`, except that it is used to access the previous value of the blob attribute. See Section 2.3.21.3, "NdbEventOperation::getBlobHandle()".

Signature.

```
NdbBlob* getPreBlobHandle
  (
    const char* name
  )
```

Parameters. The *`name`* of the blob attribute.

Return value. A pointer to an `NdbBlob`.

2.3.21.11 NdbEventOperation::getPreValue()

Description. This method performs identically to getValue(), except that it is used to define a retrieval operation of an attribute's previous value rather than the current value. See Section 2.3.21.13, "NdbEventOperation::getValue()", for details.

Signature.

```
NdbRecAttr* getPreValue
    (
      const char* name,
      char*       value = 0
    )
```

Parameters. This method takes the two parameters listed here:

- The *`name`* of the attribute (as a constant character pointer).
- A pointer to a *`value`*, such that:
  - If the attribute value is not `NULL`, then the attribute value is returned in this parameter.
  - If the attribute value is `NULL`, then the attribute value is stored only in the `NdbRecAttr` object returned by this method.

  See *`value`* Buffer Memory Allocation, for more information regarding this parameter.

Return value. An `NdbRecAttr` object to hold the value of the attribute, or a `NULL` pointer indicating that an error has occurred.

2.3.21.12 NdbEventOperation::getState()

Description. This method gets the event operation's current state.

Signature.

```
State getState
    (
      void
    )
```

Parameters. *None*.

Return value. A `State` value. See Section 2.3.21.20, “NdbEventOperation::State”.

2.3.21.13 NdbEventOperation::getValue()

Description. This method defines the retrieval of an attribute value. The NDB API allocates memory for the `NdbRecAttr` object that is to hold the returned attribute value.

Important

This method does *not* fetch the attribute value from the database, and the `NdbRecAttr` object returned by this method is not readable or printable before calling the `execute()` method and `Ndb::nextEvent()` has returned a non-`NULL` value.

If a specific attribute has not changed, the corresponding `NdbRecAttr` will be in the state `UNDEFINED`. This can be checked by using `NdbRecAttr::isNULL()` which in such cases returns `-1`.

***`value`* Buffer Memory Allocation.** It is the application's responsibility to allocate sufficient memory for the *`value`* buffer (if not `NULL`), and this buffer must be aligned appropriately. The buffer is used directly (thus avoiding a copy penalty) only if it is aligned on a 4-byte boundary and the attribute size in bytes (calculated as `NdbRecAttr::get_size_in_bytes()`) is a multiple of 4.

Note

`getValue()` retrieves the current value. Use `getPreValue()` for retrieving the previous value.

Signature.

```
NdbRecAttr* getValue
    (
      const char* name,
      char*       value = 0
    )
```

Parameters. This method takes the two parameters listed here:

- The *name* of the attribute (as a constant character pointer).
- A pointer to a *value*, such that:
  - If the attribute value is not `NULL`, then the attribute value is returned in this parameter.
  - If the attribute value is `NULL`, then the attribute value is stored only in the `NdbRecAttr` object returned by this method.

  See *value* Buffer Memory Allocation, for more information regarding this parameter.

Return value. An `NdbRecAttr` object to hold the value of the attribute, or a `NULL` pointer indicating that an error has occurred.

2.3.21.14 NdbEventOperation::hasError()

Description. This method is used to determine whether there is an error associated with this event operation.

This method is deprecated in NDB 7.4.3, and is subject to removal in a future release. In NDB 7.4.3 and later, you should instead use `getEventType2()` to determine the event type. See Section 2.3.6.23, “Event::TableEvent”.

Signature.

```
int hasError
    (
      void
    ) const
```

Parameters. *None*.

Return value. If event loss has taken place, then this method returns 0; otherwise, it returns 1.

2.3.21.15 NdbEventOperation::isConsistent()

Description. This method is used to determine whether event loss has taken place following the failure of a node.

This method is deprecated in NDB 7.4.3, and is subject to removal in a future release. In NDB 7.4.3 and later, you should instead use `getEventType2()` to determine whether the event is of type `TE_INCONSISTENT`. See Section 2.3.6.23, “Event::TableEvent”.

Signature.

```
bool isConsistent
    (
      void
    ) const
```

Parameters. *None*.

Return value. If event loss has taken place, then this method returns `false`; otherwise, it returns `true`.

2.3.21.16 NdbEventOperation::isEmptyEpoch()

Description. This method is used to determine whether consumed event data marks an empty epoch.

This method was added in NDB 7.4.3.

Signature.

```
bool isEmptyEpoch
    (
      void
    )
```

Parameters. *None*.

Return value. If this epoch is empty, the method returns `true`; otherwise, it returns `false`.

2.3.21.17 NdbEventOperation::isErrorEpoch()

Description. This method is used to determine whether consumed event data marks an empty epoch.

This method was added in NDB 7.4.3.

Signature.

```
bool isErrorEpoch
  (
    NdbDictionary::Event::TableEvent* error_type = 0
  )
```

Parameters. If this is an error epoch, *`error_type`* contains the `TableEvent` value corresponding to the error.

Return value. If this epoch is in error, the method returns `true`; otherwise, it returns `false`.

2.3.21.18 NdbEventOperation::isOverrun()

Description. This method is used to determine whether event loss has taken place due to a buffer overrun.

Signature.

```
bool isOverrun
    (
      void
    ) const
```

Parameters. *None*.

Return value. If the event buffer has been overrun, then this method returns `true`, otherwise, it returns `false`.

2.3.21.19 NdbEventOperation::mergeEvents()

Description. This method is used to set the merge events flag. For information about event merging, see Section 2.3.6.18, “Event::mergeEvents()”.

Note

The merge events flag is `false` by default.

Signature.

```
void mergeEvents
    (
      bool flag
    )
```

Parameters. A Boolean *`flag`*.

Return value. *None*.

2.3.21.20 NdbEventOperation::State

Description. This type describes the event operation's state.

Enumeration values. Possible values are shown, along with descriptions, in the following table:

Value	Description
`EO_CREATED`	The event operation has been created, but `execute()` has not yet been called.
`EO_EXECUTING`	The `execute()` method has been invoked for this event operation.
`EO_DROPPED`	The event operation is waiting to be deleted, and is no longer usable.
`EO_ERROR`	An error has occurred, and the event operation is unusable.

A `State` value is returned by the `getState()` method.

2.3.21.21 NdbEventOperation::tableFragmentationChanged()

Description. This method is used to test whether a table's fragmentation has changed in connection with a `TE_ALTER` event. (See Section 2.3.6.23, “Event::TableEvent”.)

Signature.

```
bool tableFragmentationChanged
    (
      void
    ) const
```

Parameters. *None*.

Return value. Returns `true` if the table's fragmentation has changed; otherwise, the method returns `false`.

2.3.21.22 NdbEventOperation::tableFrmChanged()

Description. Use this method to determine whether a table `.FRM` file has changed in connection with a `TE_ALTER` event. (See Section 2.3.6.23, “Event::TableEvent”.)

Signature.

```
bool tableFrmChanged
    (
```

```
      void
    ) const
```

Parameters. *None*.

Return value. Returns `true` if the table `.FRM` file has changed; otherwise, the method returns `false`.

2.3.21.23 NdbEventOperation::tableNameChanged()

Description. This method tests whether a table name has changed as the result of a `TE_ALTER` table event. (See Section 2.3.6.23, "Event::TableEvent".)

Signature.

```
bool tableNameChanged
    (
      void
    ) const
```

Parameters. *None*.

Return value. Returns `true` if the name of the table has changed; otherwise, the method returns `false`.

2.3.21.24 NdbEventOperation::tableRangeListChanged()

Description. Use this method to check whether a table range partition list name has changed in connection with a `TE_ALTER` event.

Signature.

```
bool tableRangeListChanged
    (
      void
    ) const
```

Parameters. *None*.

Return value. This method returns `true` if range or list partition name has changed; otherwise it returns `false`.

2.3.22 The NdbIndexOperation Class

This section describes the `NdbIndexOperation` class and its public methods.

Parent class. `NdbOperation`

Child classes. *None*

Description. `NdbIndexOperation` represents an index operation for use in transactions. This class inherits from `NdbOperation`.

`NdbIndexOperation` can be used only with unique hash indexes; to work with ordered indexes, use `NdbIndexScanOperation`.

Important

This class has no public constructor. To create an instance of `NdbIndexOperation`, it is necessary to use the `NdbTransaction::getNdbIndexOperation()` method.

Methods. The following table lists the public methods of this class and the purpose or use of each method:

Method	Purpose / Use
`deleteTuple()`	Removes a tuple from a table
`getIndex()`	Gets the index used by the operation
`readTuple()`	Reads a tuple from a table
`updateTuple()`	Updates an existing tuple in a table

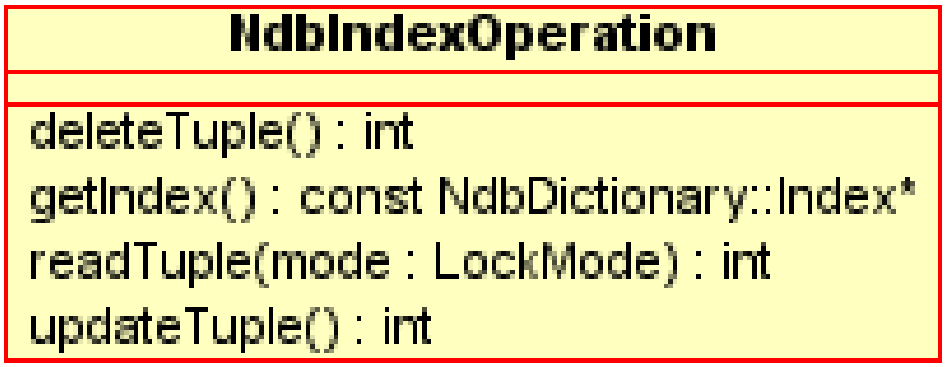
Note

Index operations are not permitted to insert tuples.

Types. The `NdbIndexOperation` class defines no public types of its own.

Class diagram. This diagram shows all the available methods of the `NdbIndexOperation` class:

Figure 2.17 NdbIndexOperation



For more information about the use of `NdbIndexOperation`, see Single-row operations.

2.3.22.1 NdbIndexOperation::deleteTuple()

Description. This method defines the `NdbIndexOperation` as a `DELETE` operation. When the `NdbTransaction::execute()` method is invoked, the operation deletes a tuple from the table.

Signature.

```
int deleteTuple
    (
      void
    )
```

Parameters. *None*.

Return value. `0` on success, `-1` on failure.

2.3.22.2 NdbIndexOperation::getIndex()

Description. Gets the index, given an index operation.

Signature.

```
const NdbDictionary::Index* getIndex
    (
      void
    ) const
```

Parameters. *None*.

Return value. A pointer to an `Index` object.

2.3.22.3 NdbIndexOperation::readTuple()

Description. This method defines the `NdbIndexOperation` as a `READ` operation. When the `NdbTransaction::execute()` method is invoked, the operation reads a tuple.

Signature.

```
int readTuple
    (
      LockMode mode
    )
```

Parameters. *`mode`* specifies the locking mode used by the read operation. See Section 2.3.25.15, "NdbOperation::LockMode", for possible values.

Return value. `0` on success, `-1` on failure.

2.3.22.4 NdbIndexOperation::updateTuple()

Description. This method defines the `NdbIndexOperation` as an `UPDATE` operation. When the `NdbTransaction::execute()` method is invoked, the operation updates a tuple found in the table.

Signature.

```
int updateTuple
    (
      void
    )
```

Parameters. *None*.

Return value. `0` on success, `-1` on failure.

2.3.23 The NdbIndexScanOperation Class

This section discusses the `NdbIndexScanOperation` class and its public members.

Parent class. `NdbScanOperation`

Child classes. *None*

Description. The `NdbIndexScanOperation` class represents a scan operation using an ordered index. This class inherits from `NdbScanOperation` and `NdbOperation`.

Note

`NdbIndexScanOperation` is for use with ordered indexes only; to work with unique hash indexes, use `NdbIndexOperation`.

Methods. The following table lists the public methods of this class and the purpose or use of each method:

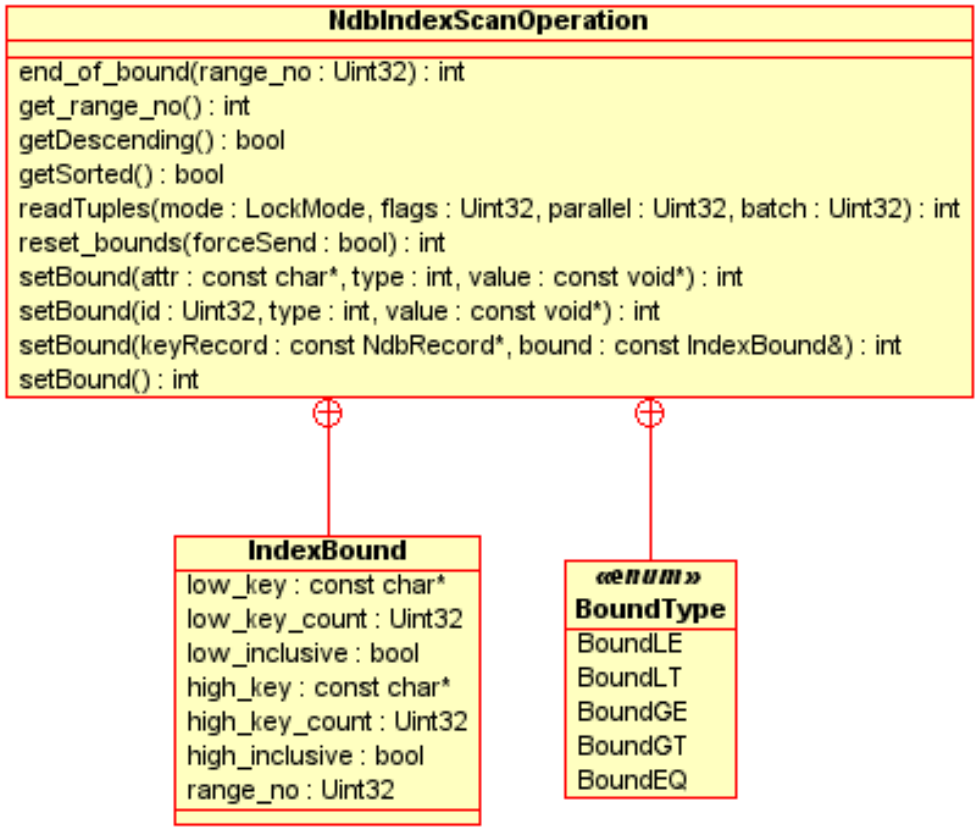
Method	Purpose / Use
`end_of_bound()`	Marks the end of a bound
`get_range_no()`	Gets the range number for the current row
`getDescending()`	Checks whether the current scan is sorted
`getSorted()`	Checks whether the current scan is sorted
`readTuples()`	Reads tuples using an ordered index
`reset_bounds()`	Resets bounds, puts the operation in the send queue
`setBound()`	Defines a bound on the index key for a range scan

Types. The `NdbIndexScanOperation` class defines one public type `BoundType`.

This class also defines an `IndexBound` struct, for use with operations employing `NdbRecord`.

Class diagram. This diagram shows all the public members of the `NdbIndexScanOperation` class:

Figure 2.18 NdbIndexScanOperation



For more information about the use of `NdbIndexScanOperation`, see Scan Operations, and Using Scans to Update or Delete Rows

2.3.23.1 NdbIndexScanOperation::BoundType

Description. This type is used to describe an ordered key bound.

Tip

The numeric values are fixed in the API and can be used explicitly; in other words, it is “safe” to calculate the values and use them.

Enumeration values. Possible values are shown, along with descriptions, in the following table:

Value	Numeric Value	Description
`BoundLE`	`0`	Lower bound
`BoundLT`	`1`	Strict lower bound
`BoundGE`	`2`	Upper bound
`BoundGT`	`3`	Strict upper bound
`BoundEQ`	`4`	Equality

2.3.23.2 NdbIndexScanOperation::end_of_bound()

Description. This method is used to mark the end of a bound; it is used when batching index reads (that is, when employing multiple ranges).

Signature.

```
int end_of_bound
    (
      Uint32 range_no
    )
```

Parameters. The number of the range on which the bound occurs.

Return value. `0` indicates success; `-1` indicates failure.

2.3.23.3 NdbIndexScanOperation::getDescending()

Description. This method is used to check whether the scan is descending.

Signature.

```
bool getDescending
    (
      void
    ) const
```

Parameters. *None*.

Return value. This method returns `true` if the scan is sorted in descending order; otherwise, it returns `false`.

2.3.23.4 NdbIndexScanOperation::get_range_no()

Description. This method returns the range number for the current row.

Signature.

```
int get_range_no
    (
      void
    )
```

Parameters. *None*.

Return value. The range number (an integer).

2.3.23.5 NdbIndexScanOperation::getSorted()

Description. This method is used to check whether the scan is sorted.

Signature.

```
bool getSorted
    (
      void
    ) const
```

Parameters. *None*.

Return value. `true` if the scan is sorted, otherwise `false`.

2.3.23.6 NdbIndexScanOperation::readTuples()

Description. This method is used to read tuples, using an ordered index.

Signature.

```
virtual int readTuples
    (
      LockMode mode = LM_Read,
      Uint32   flags = 0,
      Uint32   parallel = 0,
      Uint32   batch = 0

    )
```

Parameters. The `readTuples()` method takes the three parameters listed here:

- The lock *`mode`* used for the scan. This is a `LockMode` value; see Section 2.3.25.15, “NdbOperation::LockMode” for more information, including permitted values.
- One or more scan flags; multiple *`flags`* are `OR`'ed together as they are when used with `NdbScanOperation::readTuples()`. See Section 2.3.29.9, “NdbScanOperation::ScanFlag” for possible values.
- The number of fragments to scan in *`parallel`*; use `0` to specify the maximum automatically.
- The *`batch`* parameter specifies how many records will be returned to the client from the server by the next `NdbScanOperation::nextResult(true)` method call. Use `0` to specify the maximum automatically.

Note

This parameter was ignored prior to MySQL 5.1.12, and the maximum was used.(Bug #20252)

Return value. An integer: `0` indicates success; `-1` indicates failure.

2.3.23.7 NdbIndexScanOperation::reset_bounds()

Description. Resets the bounds, and puts the operation into the list that will be sent on the next `NdbTransaction::execute()` call.

Signature.

```
int reset_bounds
    (
      bool forceSend = false
    )
```

Parameters. Set *forceSend* to `true` in order to force the operation to be sent immediately.

Return value. Returns `0` on success, `-1` on failure.

2.3.23.8 NdbIndexScanOperation::setBound()

Description. This method defines a bound on an index key used in a range scan, and sets bounds for index scans defined using `NdbRecord`.

"Old" API usage (prior to introduction of NdbRecord). Each index key can have a lower bound, upper bound, or both. Setting the key equal to a value defines both upper and lower bounds. Bounds can be defined in any order. Conflicting definitions gives rise to an error.

Bounds must be set on initial sequences of index keys, and all but possibly the last bound must be nonstrict. This means, for example, that “a >= 2 AND b > 3” is permissible, but “a > 2 AND b >= 3” is not.

The scan may currently return tuples for which the bounds are not satisfied. For example, a `<= 2 && b <= 3` not only scans the index up to `(a=2, b=3)`, but also returns any `(a=1, b=4)` as well.

When setting bounds based on equality, it is better to use `BoundEQ` instead of the equivalent pair `BoundLE` and `BoundGE`. This is especially true when the table partition key is a prefix of the index key.

`NULL` is considered less than any non-`NULL` value and equal to another `NULL` value. To perform comparisons with `NULL`, use `setBound()` with a null pointer (`0`).

An index also stores all-`NULL` keys as well, and performing an index scan with an empty bound set returns all tuples from the table.

Signature (“Old” API). Using the “old” API, this method could be called in either of two ways. Both of these use the bound type and value; the first also uses the name of the bound, as shown here:

```
int setBound
    (
      const char* name,
      int         type,
      const void* value
    )
```

The second way to invoke this method under the “old” API uses the bound's ID rather than the name, as shown here:

```
int setBound
    (
      Uint32      id,
      int         type,
      const void* value
    )
```

Parameters (“Old” API). This method takes 3 parameters:

- Either the *name* or the *id* of the attribute on which the bound is to be set.
- The bound `type`—see Section 2.3.23.1, “NdbIndexScanOperation::BoundType”.
- A pointer to the bound *value* (use `0` for `NULL`).

As used with NdbRecord. This method is called to add a range to an index scan operation which has been defined with a call to `NdbTransaction::scanIndex()`. To add more than one range, the index scan operation must have been defined with the `SF_MultiRange` flag set. (See Section 2.3.29.9, “NdbScanOperation::ScanFlag”.)

Note

Where multiple numbered ranges are defined with multiple calls to `setBound()`, and the scan is ordered, the range number for each range must be larger than the range number for the previously defined range.

Signature.

```
int setBound
    (
      const NdbRecord* keyRecord,
      const IndexBound& bound
    )
```

Parameters. As used with `NdbRecord`, this method takes 2 parameters, listed here:

- *`keyRecord`*: This is an `NdbRecord` structure corresponding to the key on which the index is defined.
- The *`bound`* to add (see Section 2.3.12, “The IndexBound Structure”).

An additional version of this method can be used when the application knows that rows in-range will be found only within a particular partition. This is the same as that shown previously, except for the addition of a `PartitionSpec`. Doing so limits the scan to a single partition, improving system efficiency.

Signature (when specifying a partition).

```
int setBound
    (
      const NdbRecord* keyRecord,
      const IndexBound& bound,
      const Ndb::PartitionSpec* partInfo,
      Uint32 sizeOfPartInfo = 0
    )
```

Parameters (when specifying a partition). This method can also be invoked with the following four parameters:

- *`keyRecord`*: This is an `NdbRecord` structure corresponding to the key on which the index is defined.
- The *`bound`* to be added to the scan (see Section 2.3.12, “The IndexBound Structure”).
- *`partInfo`*: This is a pointer to a `PartitionSpec`, which provides extra information making it possible to scan a reduced set of partitions.
- *`sizeOfPartInfo`*: The length of the partition specification.

Note

`keyRecord` and *`bound`* are defined and used in the same way as with the 2-parameter version of this method.

Return value. Returns `0` on success, `-1` on failure.

2.3.24 The NdbInterpretedCode Class

This section discusses the `NdbInterpretedCode` class, which can be used to prepare and execute an NDB API interpreted program.

Parent class. *None*.

Child classes. *None*.

Description. `NdbInterpretedCode` represents an interpreted program for use in operations created using `NdbRecord`, or with scans created using the old API. The `NdbScanFilter` class can also be used to generate an NDB interpreted program using this class.

Important

This interface is still under development, and so is subject to change without notice. The `NdbScanFilter` API is a more stable API for defining scanning and filtering programs.

Using NdbInterpretedCode. To create an `NdbInterpretedCode` object, invoke the constructor, optionally supplying a table for the program to operate on, and a buffer for program storage and finalization. If no table is supplied, then only instructions which do not access table attributes can be used.

Note

Each NDB API operation applies to one table, and so does any `NdbInterpretedCode` program attached to that operation.

If no buffer is supplied, then an internal buffer is dynamically allocated and extended as necessary. Once the `NdbInterpretedCode` object is created, you can add instructions and labels to it by calling the appropriate methods as described later in this section. When the program has completed, finalize it by calling the `finalise()` method, which resolves any remaining internal branches and calls to label and subroutine offsets.

Note

A single finalized `NdbInterpretedCode` program can be used by more than one operation. It need not be re-prepared for successive operations.

To use the program with `NdbRecord` operations and scans, pass it at operation definition time using the `OperationOptions` or `ScanOptions` parameter. When the program is no longer required, the `NdbInterpretedCode` object can be deleted, along with any user-supplied buffer.

Error checking. For reasons of efficiency, methods of this class provide minimal error checking.

Methods. The following table lists the public methods of this class and the purpose or use of each method:

Method	Purpose / Use
`NdbInterpretedCode()`	Class constructor
`add_reg()`	Add two register values and store the result in a third register
`add_val()`	Add a value to a table column value
`branch_col_and_mask_eq_mask()`	Jump if a column value `AND`ed with a bitmask is equal to the bitmask
`branch_col_and_mask_eq_zero()`	Jump if a column value `AND`ed with a bitmask is equal to `0`
`branch_col_and_mask_ne_mask()`	Jump if a column value `AND`ed with a bitmask is not equal to the bitmask
`branch_col_and_mask_ne_zero()`	Jump if a column value `AND`ed with a bitmask is not equal to `0`
`branch_col_eq()`	Jump if a column value is equal to another
`branch_col_eq_null()`	Jump if a column value is `NULL`
`branch_col_ge()`	Jump if a column value is greater than or equal to another
`branch_col_gt()`	Jump if a column value is greater than another

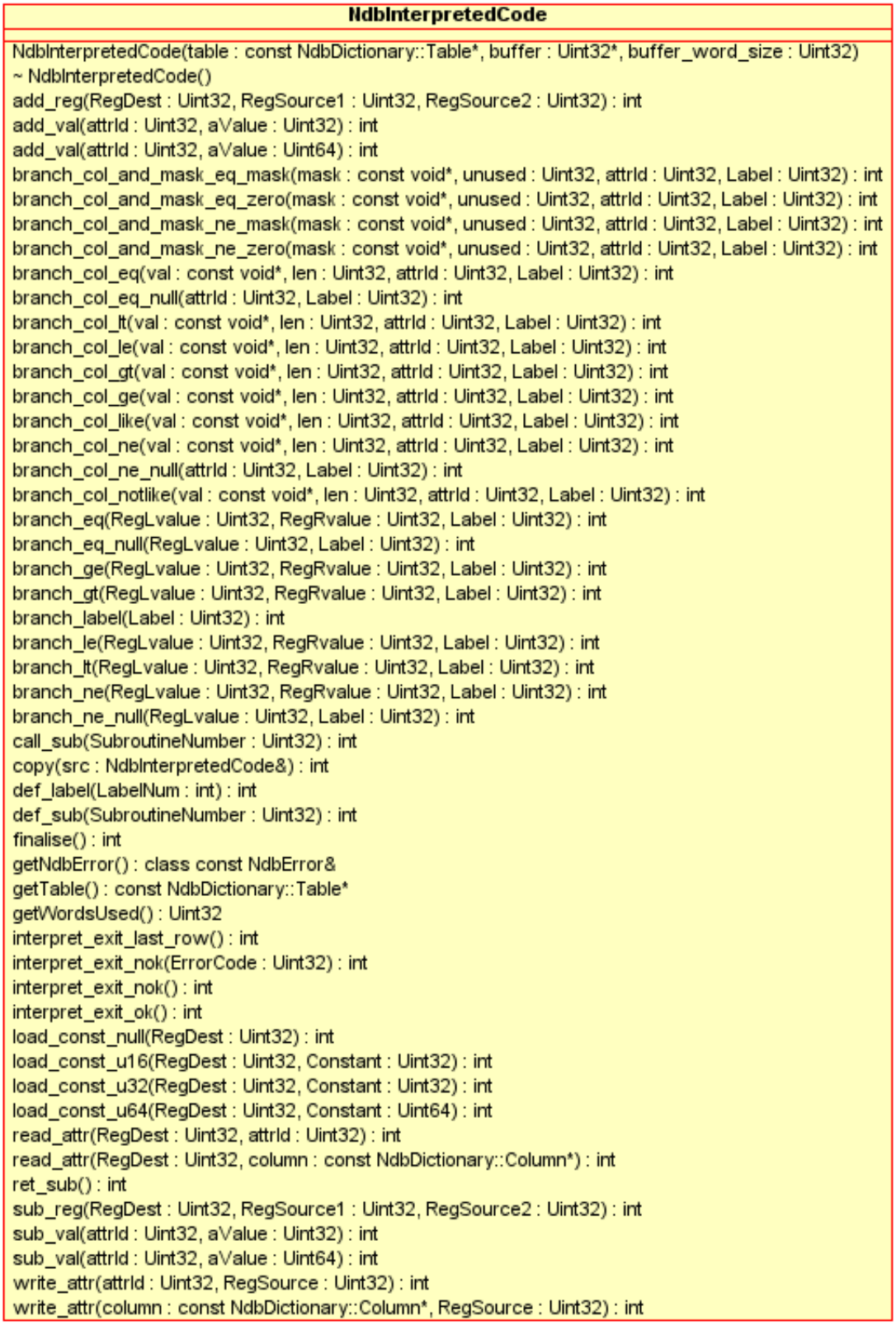
Method	Purpose / Use
branch_col_le()	Jump if a column value is less than or equal to another
branch_col_like()	Jump if a column value matches a pattern
branch_col_lt()	Jump if a column value is less than another
branch_col_ne()	Jump if a column value is not equal to another
branch_col_ne_null()	Jump if a column value is not NULL
branch_col_notlike()	Jump if a column value does not match a pattern
branch_eq()	Jump if one register value is equal to another
branch_eq_null()	Jump if a register value is NULL
branch_ge()	Jump if one register value is greater than or equal to another
branch_gt()	Jump if one register value is greater than another
branch_label()	Unconditional jump to a label
branch_le()	Jump if one register value is less than or equal to another
branch_lt()	Jump if one register value is less than another
branch_ne()	Jump if one register value is not equal to another
branch_ne_null()	Jump if a register value is not NULL
call_sub()	Call a subroutine
copy()	Make a deep copy of an NdbInterpretedCode object
def_label()	Create a label for use within the interpreted program
def_sub()	Define a subroutine
finalise()	Completes interpreted program and prepares it for use
getNdbError()	Gets the most recent error associated with this NdbInterpretedCode object
getTable()	Gets the table on which the program is defined
getWordsUsed()	Gets the number of words used in the buffer
interpret_exit_last_row()	Return a row as part of the result, and do not check any more rows in this fragment
interpret_exit_nok()	Do not return a row as part of the result
interpret_exit_ok()	Return a row as part of the result
load_const_null()	Load a NULL value into a register
load_const_u16()	Load a 16-bit numeric value into a register
load_const_u32()	Load a 32-bit numeric value into a register
load_const_u64()	Load a 64-bit numeric value into a register
read_attr()	Read a register value into a table column
ret_sub()	Return from a subroutine
sub_reg()	Subtract two register values and store the result in a third register
sub_val()	Subtract a value from a table column value
write_attr()	Write a table column value into a register

See also Section 2.3.24.1, “Using NdbInterpretedCode”.

Types. This class defines no public types.

Class diagram. This diagram shows all the available methods of the NdbInterpretedCode class:

Figure 2.19 NdbInterpretedCode



2.3.24.1 Using NdbInterpretedCode

The next few sections provide information about performing different types of operations with `NdbInterpretedCode` methods, including resource usage.

NdbInterpretedCode Methods for Loading Values into Registers

The methods described in this section are used to load constant values into `NdbInterpretedCode` program registers. The space required by each of these methods is shown in the following table:

Method	Buffer (words)	Request message (words)
`load_const_null()`	1	1
`load_const_u16()`	1	1
`load_const_u32()`	2	2
`load_const_u64()`	3	3

NdbInterpretedCode Methods for Copying Values Between Registers and Table Columns

`NdbInterpretedCode` provides two methods for copying values between a column in the current table row and a program register. The `read_attr()` method is used to copy a table column value into a program register; `write_attr()` is used to copy a value from a program register into a table column. Both of these methods require that the table being operated on was specified when creating the `NdbInterpretedCode` object for which they are called.

The space required by each of these methods is shown in the following table:

Method	Buffer (words)	Request message (words)
`read_attr()`	1	1
`write_attr()`	1	1

For more information, see Section 2.3.24.43, “NdbInterpretedCode::read_attr()”, and Section 2.3.24.47, “NdbInterpretedCode::write_attr()”.

NdbInterpretedCode Register Arithmetic Methods

`NdbInterpretedCode` provides two methods for performing arithmetic operations on registers. Using `add_reg()`, you can load the sum of two registers into another register; `sub_reg()` lets you load the difference of two registers into another register.

The space required by each of these methods is shown in the following table:

Method	Buffer (words)	Request message (words)
`add_reg()`	1	1
`sub_reg()`	1	1

For mroe information, see Section 2.3.24.3, “NdbInterpretedCode::add_reg()”, and Section 2.3.24.45, “NdbInterpretedCode::sub_reg()”.

NdbInterpretedCode: Labels and Branching

The `NdbInterpretedCode` class lets you define labels within interpreted programs and provides a number of methods for performing jumps to these labels based on any of the following types of conditions:

- Comparison between two register values
- Comparison between a column value and a given constant
- Whether or not a column value matches a given pattern

To define a label, use the `def_label()` method.

To perform an unconditional jump to a label, use the `branch_label()` method.

To perform a jump to a given label based on a comparison of register values, use one of the `branch_*()` methods (`branch_ge()`, `branch_gt()`, `branch_le()`, `branch_lt()`, `branch_eq()`, `branch_ne()`, `branch_ne_null()`, or `branch_eq_null()`). See Register-Based NdbInterpretedCode Branch Operations.

To perform a jump to a given label based on a comparison of table column values, use one of the `branch_col_*()` methods (`branch_col_ge()`, `branch_col_gt()`, `branch_col_le()`, `branch_col_lt()`, `branch_col_eq()`, `branch_col_ne()`, `branch_col_ne_null()`, or `branch_col_eq_null()`). See Column-Based NdbInterpretedCode Branch Operations.

To perform a jump based on pattern-matching of a table column value, use one of the methods `branch_col_like()` or `branch_col_notlike()`. See Pattern-Based NdbInterpretedCode Branch Operations.

Register-Based NdbInterpretedCode Branch Operations

Most of these are used to branch based on the results of register-to-register comparisons. There are also two methods used to compare a register value with `NULL`. All of these methods require as a parameter a label defined using the `def_label()` method.

These methods can be thought of as performing the following logic:

```
if(register_value1 condition register_value2)
  goto Label
```

The space required by each of these methods is shown in the following table:

Method	Buffer (words)	Request message (words)
`branch_ge()`	1	1
`branch_gt()`	1	1
`branch_le()`	1	1
`branch_lt()`	1	1
`branch_eq()`	1	1
`branch_ne()`	1	1
`branch_ne_null()`	1	1
`branch_eq_null()`	1	1

Column-Based NdbInterpretedCode Branch Operations

The methods described in this section are used to perform branching based on a comparison between a table column value and a given constant value. Each of these methods expects the attribute ID of the column whose value is to be tested rather than a reference to a `Column` object.

These methods, with the exception of `branch_col_eq_null()` and `branch_col_ne_null()`, can be thought of as performing the following logic:

```
if(constant_value condition column_value)
  goto Label
```

In each case (once again excepting `branch_col_eq_null()` and `branch_col_ne_null()`), the arbitrary constant is the first parameter passed to the method.

The space requirements for each of these methods is shown in the following table, where *L* represents the length of the constant value:

Method	Buffer (words)	Request message (words)
`branch_col_eq_null()`	2	2
`branch_col_ne_null()`	2	2
`branch_col_eq()`	2	2 + `CEIL(L / 8)`
`branch_col_ne()`	2	2 + `CEIL(L / 8)`
`branch_col_lt()`	2	2 + `CEIL(L / 8)`
`branch_col_le()`	2	2 + `CEIL(L / 8)`
`branch_col_gt()`	2	2 + `CEIL(L / 8)`
`branch_col_ge()`	2	2 + `CEIL(L / 8)`

Note

The expression `CEIL(L / 8)` is the number of whole 8-byte words required to hold the constant value to be compared.

Pattern-Based NdbInterpretedCode Branch Operations

The `NdbInterpretedCode` class provides two methods which can be used to branch based on a comparison between a column containing character data (that is, a `CHAR`, `VARCHAR`, `BINARY`, or `VARBINARY` column) and a regular expression pattern.

The pattern syntax supported by the regular expression is the same as that supported by the MySQL Server's `LIKE` and `NOT LIKE` operators, including the `_` and `%` metacharacters. For more information about these, see String Comparison Functions.

Note

This is the same regular expression pattern syntax that is supported by `NdbScanFilter`; see Section 2.3.28.3, “NdbScanFilter::cmp()”, for more information.

The table being operated upon must be supplied when the `NdbInterpretedCode` object is instantiated. The regular expression pattern should be in plain `CHAR` format, even if the column is actually a `VARCHAR` (in other words, there should be no leading length bytes).

These functions behave as shown here:

```
if (column_value [NOT] LIKE pattern)
  goto Label;
```

The space requirements for these methods are shown in the following table, where *L* represents the length of the constant value:

Method	Buffer (words)	Request message (words)
`branch_col_like()`	2	2 + `CEIL(L / 8)`
`branch_col_notlike()`	2	2 + `CEIL(L / 8)`

Note

The expression `CEIL(L / 8)` is the number of whole 8-byte words required to hold the constant value to be compared.

NdbInterpretedCode Bitwise Comparison Operations

These instructions are used to branch based on the result of a logical `AND` comparison between a `BIT` column value and a bitmask pattern.

Use of these methods requires that the table being operated upon was supplied when the `NdbInterpretedCode` object was constructed. The mask value should be the same size as the bit column being compared. `BIT` values are passed into and out of the NDB API as 32-bit words with bits set in order from the least significant bit to the most significant bit. The endianness of the platform on which the instructions are executed controls which byte contains the least significant bits. On x86, this is the first byte (byte 0); on SPARC and PPC, it is the last byte.

The buffer length and the request length for each of the methods listed here each requires an amount of space equal to 2 words plus the column width rounded (up) to the nearest whole word:

- `branch_col_and_mask_eq_mask()`
- `branch_col_and_mask_ne_mask()`
- `branch_col_and_mask_eq_zero()`
- `branch_col_and_mask_ne_zero()`

NdbInterpretedCode Result Handling Methods

The methods described in this section are used to tell the interpreter that processing of the current row is complete, and—in the case of scans—whether or not to include this row in the results of the scan.

The space requirements for these methods are shown in the following table, where *L* represents the length of the constant value:

Method	Buffer (words)	Request message (words)
`interpret_exit_ok()`	1	1
`interpret_exit_nok()`	1	1
`interpret_exit_last_row()`	1	1

NdbInterpretedCode Convenience Methods

The methods described in this section can be used to insert multiple instructions (using specific registers) into an interpreted program.

Important

In addition to updating the table column, these methods use interpreter registers 6 and 7, replacing any existing contents of register 6 with the original column value and any existing contents of register 7 with the modified column value. The table itself must be previously defined when instantiating the `NdbInterpretedCode` object for which the method is invoked.

The space requirements for these methods are shown in the following table, where *L* represents the length of the constant value:

Method	Buffer (words)	Request message (words)
`add_val()`	4	1; if the supplied value >= 2^{16}: 2; if >= 2^{32}: 3
`sub_val()`	4	1; if the supplied value >= 2^{16}: 2; if >= 2^{32}: 3

Using Subroutines with NdbInterpretedCode

`NdbInterpretedCode` supports subroutines which can be invoked from within interpreted programs, with each subroutine being identified by a unique number. Subroutines can be defined only following all main program instructions.

Important

Numbers used to identify subroutines must be contiguous; however, they do not have to be in any particular order.

- The beginning of a subroutine is indicated by invoking the `def_sub()` method;
- `ret_sub()` terminates the subroutine; all instructions following the call to `def_sub()` belong to the subroutine until it is terminated using this method.
- A subroutine is called using the `call_sub()` method.

Once the subroutine has completed, the program resumes execution with the instruction immediately following the one which invoked the subroutine. Subroutines can also be invoked from other subroutines; currently, the maximum subroutine stack depth is 32.

NdbInterpretedCode Utility Methods

Some additional utility methods supplied by `NdbInterpretedCode` are listed here:

- `copy()`: Copies an existing interpreted program by performing a deep copy on the associated `NdbInterpretedCode` object.
- `finalise()`: Prepares an interpreted program by resolving all branching instructions and subroutine calls.
- `getTable()`: Get a reference to the table for which the `NdbInterpretedCode` object was defined.
- `getNdbError()`: Get the most recent error associated with this `NdbInterpretedCode` object.
- `getWordsUsed()`: Obtain the number of words used from the buffer.

2.3.24.2 NdbInterpretedCode Constructor

Description. This is the `NdbInterpretedCode` class constuctor.

Signature.

```
NdbInterpretedCode
    (
      const NdbDictionary::Table* table = 0,
      Uint32* buffer = 0,
      Uint32 buffer_word_size = 0
    )
```

Parameters. The `NdbInterpretedCode` constructor takes three parameters, as described here:

- The *`table`* against which this program is to be run. This parameter must be supplied if the program is table-specific—that is, if it reads from or writes to columns in a table.
- A pointer to a *`buffer`* of 32-bit words used to store the program.
- *`buffer_word_size`* is the length of the buffer passed in. If the program exceeds this length then adding new instructions will fail with error `4518 Too many instructions in interpreted program`.

  Alternatively, if no buffer is passed, a buffer will be dynamically allocated internally and extended to cope as instructions are added.

Return value. An instance of `NdbInterpretedCode`.

2.3.24.3 NdbInterpretedCode::add_reg()

Description. This method sums the values stored in any two given registers and stores the result in a third register.

Signature.

```
int add_reg
    (
      Uint32 RegDest,
      Uint32 RegSource1,
      Uint32 RegSource2
    )
```

Parameters. This method takes three parameters. The first of these is the register in which the result is to be stored (*`RegDest`*). The second and third parameters (*`RegSource1`* and *`RegSource2`*) are the registers whose values are to be summed.

Note

It is possible to re-use for storing the result one of the registers whose values are summed; that is, *`RegDest`* can be the same as *`RegSource1`* or *`RegSource2`*.

Return value. Returns `0` on success, `-1` on failure.

2.3.24.4 NdbInterpretedCode::add_val()

Description. This method adds a specified value to the value of a given table column, and places the original and modified column values in registers 6 and 7. It is equivalent to the following series of `NdbInterpretedCode` method calls, where *`attrId`* is the table column' attribute ID and *`aValue`* is the value to be added:

```
read_attr(6, attrId);
load_const_u32(7, aValue);
add_reg(7,6,7);
write_attr(attrId, 7);
```

`aValue` can be a 32-bit or 64-bit integer.

Signature. This method can be invoked in either of two ways, depending on whether *`aValue`* is 32-bit or 64-bit.

32-bit *`aValue`*:

```
int add_val
    (
      Uint32 attrId,
      Uint32 aValue
    )
```

64-bit *`aValue`*:

```
int add_val
    (
      Uint32 attrId,
      Uint64 aValue
    )
```

Parameters. A table column attribute ID and a 32-bit or 64-bit integer value to be added to this column value.

Return value. Returns `0` on success, `-1` on failure.

2.3.24.5 NdbInterpretedCode::branch_col_and_mask_eq_mask()

Description. This method is used to compare a `BIT` column value with a bitmask; if the column value `AND`ed together with the bitmask is equal to the bitmask, then execution jumps to a specified label specified in the method call.

Signature.

```
int branch_col_and_mask_eq_mask
    (
      const void* mask,
      Uint32 unused,
      Uint32 attrId,
      Uint32 Label
    )
```

Parameters. This method can accept four parameters, of which three are actually used. These are described in the following list:

- A pointer to a constant *`mask`* to compare the column value to
- A `Uint32` value which is currently *`unused`*.
- The *`attrId`* of the column to be compared.
- A program *`Label`* to jump to if the condition is true.

Return value. This method returns `0` on success and `-1` on failure.

2.3.24.6 NdbInterpretedCode::branch_col_and_mask_eq_zero()

Description. This method is used to compare a `BIT` column value with a bitmask; if the column value `AND`ed together with the bitmask is equal to `0`, then execution jumps to a specified label specified in the method call.

Signature.

```
int branch_col_and_mask_eq_zero
    (
      const void* mask,
      Uint32 unused,
      Uint32 attrId,
      Uint32 Label
    )
```

Parameters. This method can accept the following four parameters, of which three are actually used:

- A pointer to a constant *`mask`* to compare the column value to.
- A `Uint32` value which is currently *`unused`*.
- The *`attrId`* of the column to be compared.
- A program *`Label`* to jump to if the condition is true.

Return value. This method returns `0` on success and `-1` on failure.

2.3.24.7 NdbInterpretedCode::branch_col_and_mask_ne_mask()

Description. This method is used to compare a `BIT` column value with a bitmask; if the column value `AND`ed together with the bitmask is not equal to the bitmask, then execution jumps to a specified label specified in the method call.

Signature.

```
int branch_col_and_mask_ne_mask
    (
      const void* mask,
      Uint32 unused,
      Uint32 attrId,
      Uint32 Label
    )
```

Parameters. This method accepts four parameters, of which three are actually used. These described in the following list:

- A pointer to a constant *`mask`* to compare the column value to.
- A `Uint32` value which is currently *`unused`*.
- The *`attrId`* of the column to be compared.
- A program *`Label`* to jump to if the condition is true.

Return value. This method returns `0` on success and `-1` on failure.

2.3.24.8 NdbInterpretedCode::branch_col_and_mask_ne_zero()

Description. This method is used to compare a `BIT` column value with a bitmask; if the column value `AND`ed together with the bitmask is not equal to `0`, then execution jumps to a specified label specified in the method call.

Signature.

```
int branch_col_and_mask_ne_zero
    (
      const void* mask,
      Uint32 unused,
      Uint32 attrId,
      Uint32 Label
    )
```

Parameters. This method accepts the following four parameters, of which three are actually used:

- A pointer to a constant *`mask`* to compare the column value to.
- A `Uint32` value which is currently *`unused`*.
- The *`attrId`* of the column to be compared.
- A program *`Label`* to jump to if the condition is true.

Return value. This method returns `0` on success and `-1` on failure.

2.3.24.9 NdbInterpretedCode::branch_col_eq()

Description. This method compares a table column value with an arbitrary constant and jumps to the specified program label if the values are equal.

Signature.

```
int branch_col_eq
    (
      const void* val,
      Uint32 len,
      Uint32 attrId,
      Uint32 Label
    )
```

Parameters. This method takes the following four parameters:

- A constant value (*val*)
- The length of the value (in bytes)
- The attribute ID of the table column whose value is to be compared with *val*
- A *Label* (previously defined using def_label()) to jump to if the compared values are equal

Return value. Returns 0 on success, -1 on failure.

2.3.24.10 NdbInterpretedCode::branch_col_eq_null()

Description. This method tests the value of a table column and jumps to the indicated program label if the column value is NULL.

Signature.

```
int branch_col_eq_null
    (
      Uint32 attrId,
      Uint32 Label
    )
```

Parameters. This method requires the following two parameters:

- The attribute ID of the table column
- The program label to jump to if the column value is NULL

Return value. Returns 0 on success, -1 on failure.

2.3.24.11 NdbInterpretedCode::branch_col_ge()

Description. This method compares a table column value with an arbitrary constant and jumps to the specified program label if the constant is greater than or equal to the column value.

Signature.

```
int branch_col_ge
    (
      const void* val,
      Uint32 len,
      Uint32 attrId,
      Uint32 Label
    )
```

Parameters. This method takes the four parameters listed here:

- A constant value (*val*)
- The length of the value (in bytes)
- The attribute ID of the table column whose value is to be compared with *val*
- A *Label* (previously defined using def_label()) to jump to if the constant value is greater than or equal to the column value

Return value. Returns 0 on success, -1 on failure.

2.3.24.12 NdbInterpretedCode::branch_col_gt()

Description. This method compares a table column value with an arbitrary constant and jumps to the specified program label if the constant is greater than the column value.

Signature.

```
int branch_col_gt
    (
      const void* val,
      Uint32 len,
      Uint32 attrId,
      Uint32 Label
    )
```

Parameters. This method takes the following four parameters:

- A constant value (*val*)
- The length of the value (in bytes)
- The attribute ID of the table column whose value is to be compared with *val*
- A *Label* (previously defined using `def_label()`) to jump to if the constant value is greater than the column value

Return value. Returns `0` on success, `-1` on failure.

2.3.24.13 NdbInterpretedCode::branch_col_le()

Description. This method compares a table column value with an arbitrary constant and jumps to the specified program label if the constant is less than or equal to the column value.

Signature.

```
int branch_col_le
    (
      const void* val,
      Uint32 len,
      Uint32 attrId,
      Uint32 Label
    )
```

Parameters. This method takes the four parameters listed here:

- A constant value (*val*)
- The length of the value (in bytes)
- The attribute ID of the table column whose value is to be compared with *val*
- A *Label* (previously defined using `def_label()`) to jump to if the constant value is less than or equal to the column value

Return value. Returns `0` on success, `-1` on failure.

2.3.24.14 NdbInterpretedCode::branch_col_like()

Description. This method tests a table column value against a regular expression pattern and jumps to the indicated program label if they match.

Signature.

```
int branch_col_like
    (
      const void* val,
      Uint32 len,
      Uint32 attrId,
      Uint32 Label
    )
```

Parameters. This method takes four parameters, which are listed here:

- A regular expression pattern (*`val`*); see Pattern-Based NdbInterpretedCode Branch Operations, for the syntax supported
- Length of the pattern (in bytes)
- The attribute ID for the table column being tested
- The program label to jump to if the table column value matches the pattern

Return value. `0` on success, `-1` on failure

2.3.24.15 NdbInterpretedCode::branch_col_lt()

Description. This method compares a table column value with an arbitrary constant and jumps to the specified program label if the constant is less than the column value.

Signature.

```
int branch_col_lt
    (
      const void* val,
      Uint32 len,
      Uint32 attrId,
      Uint32 Label
    )
```

Parameters. This method takes the following four parameters:

- A constant value (*`val`*)
- The length of the value (in bytes)
- The attribute ID of the table column whose value is to be compared with *`val`*
- A *`Label`* (previously defined using `def_label()`) to jump to if the constant value is less than the column value

Return value. `0` on success, `-1` on failure.

2.3.24.16 NdbInterpretedCode::branch_col_ne()

Description. This method compares a table column value with an arbitrary constant and jumps to the specified program label if the two values are not equal.

Signature.

```
int branch_col_ne
    (
      const void* val,
      Uint32 len,
      Uint32 attrId,
      Uint32 Label
    )
```

Parameters. This method takes the four parameters listed here:

- A constant value (*`val`*)
- The length of the value (in bytes)
- The attribute ID of the table column whose value is to be compared with *`val`*
- A *`Label`* (previously defined using `def_label()`) to jump to if the compared values are unequal

Return value. Returns `0` on success, `-1` on failure.

2.3.24.17 NdbInterpretedCode::branch_col_ne_null()

Description. This method tests the value of a table column and jumps to the indicated program label if the column value is not `NULL`.

Signature.

```
int branch_col_ne_null
    (
      Uint32 attrId,
      Uint32 Label
    )
```

Parameters. This method requires the following two parameters:

- The attribute ID of the table column
- The program label to jump to if the column value is not `NULL`

Return value. Returns `0` on success, `-1` on failure.

2.3.24.18 NdbInterpretedCode::branch_col_notlike()

Description. This method is similar to `branch_col_like()` in that it tests a table column value against a regular expression pattern; however it jumps to the indicated program label only if the pattern and the column value do *not* match.

Signature.

```
int branch_col_notlike
    (
      const void* val,
      Uint32 len,
      Uint32 attrId,
      Uint32 Label
    )
```

Parameters. This method takes the following four parameters:

- A regular expression pattern (*`val`*); see Pattern-Based NdbInterpretedCode Branch Operations, for the syntax supported
- Length of the pattern (in bytes)
- The attribute ID for the table column being tested
- The program label to jump to if the table column value does not match the pattern

Return value. Returns `0` on success, `-1` on failure

2.3.24.19 NdbInterpretedCode::branch_eq()

Description. This method compares two register values; if they equal, then the interpreted program jumps to the specified label.

Signature.

```
int branch_eq
    (
      Uint32 RegLvalue,
      Uint32 RegRvalue,
```

```
      Uint32 Label
    )
```

Parameters. This method takes three parameters, the registers whose values are to be compared—*RegLvalue* and *RegRvalue*—and the program *Label* to jump to if they are equal. *Label* must have been defined previously using `def_label()` (see Section 2.3.24.30, "NdbInterpretedCode::def_label()").

Return value. `0` on success, `-1` on failure.

2.3.24.20 NdbInterpretedCode::branch_eq_null()

Description. This method compares a register value with `NULL`; if the register value is null, then the interpreted program jumps to the specified label.

Signature.

```
int branch_eq_null
    (
      Uint32 RegLvalue,
      Uint32 Label
    )
```

Parameters. This method takes two parameters, the register whose value is to be compared with `NULL` (*RegLvalue*) and the program *Label* to jump to if *RegLvalue* is null. *Label* must have been defined previously using `def_label()` (see Section 2.3.24.30, "NdbInterpretedCode::def_label()").

Return value. `0` on success, `-1` on failure.

2.3.24.21 NdbInterpretedCode::branch_ge()

Description. This method compares two register values; if the first is greater than or equal to the second, the interpreted program jumps to the specified label.

Signature.

```
int branch_ge
    (
      Uint32 RegLvalue,
      Uint32 RegRvalue,
      Uint32 Label
    )
```

Parameters. This method takes three parameters, the registers whose values are to be compared —*RegLvalue* and *RegRvalue*—and the program *Label* to jump to if *RegLvalue* is greater than or equal to *RegRvalue*. *Label* must have been defined previously using `def_label()` (see Section 2.3.24.30, "NdbInterpretedCode::def_label()").

Return value. `0` on success, `-1` on failure.

2.3.24.22 NdbInterpretedCode::branch_gt()

Description. This method compares two register values; if the first is greater than the second, the interpreted program jumps to the specified label.

Signature.

```
int branch_gt
    (
      Uint32 RegLvalue,
      Uint32 RegRvalue,
      Uint32 Label
    )
```

Parameters. This method takes three parameters, the registers whose values are to be compared—*RegLvalue* and *RegRvalue*—and the program *Label* to jump to if *RegLvalue* is greater than *RegRvalue*. *Label* must have been defined previously using `def_label()` (see Section 2.3.24.30, "NdbInterpretedCode::def_label()").

Return value. `0` on success, `-1` on failure.

2.3.24.23 NdbInterpretedCode::branch_label()

Description. This method performs an unconditional jump to an interpreted program label (see Section 2.3.24.30, "NdbInterpretedCode::def_label()").

Signature.

```
int branch_label
    (
      Uint32 Label
    )
```

Parameters. This method takes a single parameter, an interpreted program *Label* defined using `def_label()`.

Return value. `0` on success, `-1` on failure.

2.3.24.24 NdbInterpretedCode::branch_le()

Description. This method compares two register values; if the first is less than or equal to the second, the interpreted program jumps to the specified label.

Signature.

```
int branch_le
    (
      Uint32 RegLvalue,
      Uint32 RegRvalue,
      Uint32 Label
    )
```

Parameters. This method takes three parameters, the registers whose values are to be compared—*RegLvalue* and *RegRvalue*—and the program *Label* to jump to if *RegLvalue* is less than or equal to *RegRvalue*. *Label* must have been defined previously using `def_label()` (see Section 2.3.24.30, "NdbInterpretedCode::def_label()").

Return value. `0` on success, `-1` on failure.

2.3.24.25 NdbInterpretedCode::branch_lt()

Description. This method compares two register values; if the first is less than the second, the interpreted program jumps to the specified label.

Signature.

```
int branch_lt
    (
      Uint32 RegLvalue,
      Uint32 RegRvalue,
      Uint32 Label
    )
```

Parameters. This method takes three parameters, the registers whose values are to be compared—*RegLvalue* and *RegRvalue*—and the program *Label* to jump to if *RegLvalue* is less than *RegRvalue*. *Label* must have been defined previously using `def_label()` (see Section 2.3.24.30, "NdbInterpretedCode::def_label()").

Return value. `0` on success, `-1` on failure.

2.3.24.26 NdbInterpretedCode::branch_ne()

Description. This method compares two register values; if they are not equal, then the interpreted program jumps to the specified label.

Signature.

```
int branch_ne
    (
      Uint32 RegLvalue,
      Uint32 RegRvalue,
      Uint32 Label
    )
```

Parameters. This method takes three parameters, the registers whose values are to be compared—*`RegLvalue`* and *`RegRvalue`*—and the program *`Label`* to jump they are not equal. *`Label`* must have been defined previously using `def_label()` (see Section 2.3.24.30, “NdbInterpretedCode::def_label()”).

Return value. `0` on success, `-1` on failure.

2.3.24.27 NdbInterpretedCode::branch_ne_null()

Description. This method compares a register value with `NULL`; if the value is not null, then the interpreted program jumps to the specified label.

Signature.

```
int branch_ne_null
    (
      Uint32 RegLvalue,
      Uint32 Label
    )
```

Parameters. This method takes two parameters, the register whose value is to be compared with `NULL` (*`RegLvalue`*) and the program *`Label`* to jump to if *`RegLvalue`* is not null. *`Label`* must have been defined previously using `def_label()` (see Section 2.3.24.30, “NdbInterpretedCode::def_label()”).

Return value. `0` on success, `-1` on failure.

2.3.24.28 NdbInterpretedCode::call_sub()

Description. This method is used to call a subroutine.

Signature.

```
int call_sub
    (
      Uint32 SubroutineNumber
    )
```

Parameters. This method takes a single parameter, the number identifying the subroutine to be called.

Return value. Returns `0` on success, `-1` on failure.

2.3.24.29 NdbInterpretedCode::copy()

Description. Makes a deep copy of an `NdbInterpretedCode` object.

Signature.

```
int copy
    (
      const NdbInterpretedCode& src
    )
```

Parameters. A reference to the copy.

Return value. 0 on success, or an error code.

2.3.24.30 NdbInterpretedCode::def_label()

Description. This method defines a label to be used as the target of one or more jumps in an interpreted program.

`def_label()` uses a 2-word buffer and requires no space for request messages.

Signature.

```
int def_label
    (
      int LabelNum
    )
```

Parameters. This method takes a single parameter *`LabelNum`*, whose value must be unique among all values used for labels within the interpreted program.

Return value. `0` on success; `-1` on failure.

2.3.24.31 NdbInterpretedCode::def_sub()

Description. This method is used to mark the start of a subroutine. See Using Subroutines with NdbInterpretedCode, for more information.

Signature.

```
int def_sub
    (
      Uint32 SubroutineNumber
    )
```

Parameters. A single parameter, a number used to identify the subroutine.

Return value. Returns `0` on success, `-1` otherwise.

2.3.24.32 NdbInterpretedCode::finalise()

Description. This method prepares an interpreted program, including any subroutines it might have, by resolving all branching instructions and calls to subroutines. It must be called before using the program, and can be invoked only once for any given `NdbInterpretedCode` object.

If no instructions have been defined, this method attempts to insert a single `interpret_exit_ok()` method call prior to finalization.

Signature.

```
int finalise
    (
      void
    )
```

Parameters. *None*.

Return value. Returns `0` on success, `-1` otherwise.

2.3.24.33 NdbInterpretedCode::getNdbError()

Description. This method returns the most recent error associated with this `NdbInterpretedCode` object.

Signature.

```
const class NdbError& getNdbError
    (
      void
    ) const
```

Parameters. *None*.

Return value. A reference to an `NdbError` object.

2.3.24.34 NdbInterpretedCode::getTable()

Description. This method can be used to obtain a reference to the table for which the `NdbInterpretedCode` object was defined.

Signature.

```
const NdbDictionary::Table* getTable
    (
      void
    ) const
```

Parameters. *None*.

Return value. A pointer to a `Table` object. Returns `NULL` if no table object was supplied when the `NdbInterpretedCode` was instantiated.

2.3.24.35 NdbInterpretedCode::getWordsUsed()

Description. This method returns the number of words from the buffer that have been used, whether the buffer is one that is user-supplied or the internally-provided buffer.

Signature.

```
Uint32 getWordsUsed
    (
      void
    ) const
```

Parameters. *None*.

Return value. The 32-bit number of words used from the buffer.

2.3.24.36 NdbInterpretedCode::interpret_exit_last_row()

Description. For a scanning operation, invoking this method indicates that this row should be returned as part of the scan, and that no more rows in this fragment should be scanned. For other types of operations, the method causes the operation to be aborted.

Signature.

```
int interpret_exit_last_row
    (
      void
    )
```

Parameters. *None*.

Return value. Returns `0` if successful, `-1` otherwise.

2.3.24.37 NdbInterpretedCode::interpret_exit_nok()

Description. For scanning operations, this method is used to indicate that the current row should not be returned as part of the scan, and to cause the program should move on to the next row. It causes other types of operations to be aborted.

Signature.

```
int interpret_exit_nok
    (
      Uint32 ErrorCode = 626 //  HA_ERR_KEY_NOT_FOUND
    )
```

Parameters. This method takes a single (optional) parameter *`ErrorCode`* which . For a complete listing of NDB error codes, see Section 2.4.2, “NDB Error Codes: by Type”. If not supplied, defaults to 626 (**`HA_ERR_KEY_NOT_FOUND`**/`Tuple did not exist`. Applications should use error code 626 or another code in the range 6000 to 6999 inclusive.

Prior to NDB 7.2.1, *`ErrorCode`* defaulted to `NDB` error code `899` (`Rowid already allocated`). This value is still supported for backward compatibility. (Bug #16176006) For any values other than those mentioned here, the behavior of this method is undefined, and is subject to change at any time without prior notice.

Return value. Returns `0` on success, `-1` on failure.

2.3.24.38 NdbInterpretedCode::interpret_exit_ok()

Description. For a scanning operation, this method indicates that the current row should be returned as part of the results of the scan and that the program should move on to the next row. For other operations, calling this method causes the interpreted program to exit.

Signature.

```
int interpret_exit_ok
    (
      void
    )
```

Parameters. *None*.

Return value. Returns `0` on success, `-1` on failure.

2.3.24.39 NdbInterpretedCode::load_const_null()

Description. This method is used to load a `NULL` value into a register.

Signature.

```
int load_const_null
    (
      Uint32 RegDest
    )
```

Parameters. This method takes a single parameter, the register into which to place the `NULL`.

Return value. Returns `0` on success, `-1` otherwise.

2.3.24.40 NdbInterpretedCode::load_const_u16()

Description. This method loads a 16-bit value into the specified interpreter register.

Signature.

```
int load_const_u16
    (
      Uint32 RegDest,
      Uint32 Constant
    )
```

Parameters. This method takes the following two parameters:

- *`RegDest`*: The register into which the value should be loaded.
- A *`Constant`* value to be loaded

Return value. Returns `0` on success, `-1` otherwise.

2.3.24.41 NdbInterpretedCode::load_const_u32()

Description. This method loads a 32-bit value into the specified interpreter register.

Signature.

```
int load_const_u32
    (
      Uint32 RegDest,
      Uint32 Constant
    )
```

Parameters. This method takes the following two parameters:

- *`RegDest`*: The register into which the value should be loaded.
- A *`Constant`* value to be loaded

Return value. Returns `0` on success, `-1` otherwise.

2.3.24.42 NdbInterpretedCode::load_const_u64()

Description. This method loads a 64-bit value into the specified interpreter register.

Signature.

```
int load_const_u64
    (
      Uint32 RegDest,
      Uint64 Constant
    )
```

Parameters. This method takes the following two parameters:

- *`RegDest`*: The register into which the value should be loaded.
- A *`Constant`* value to be loaded

Return value. Returns `0` on success, `-1` otherwise.

2.3.24.43 NdbInterpretedCode::read_attr()

Description. The `read_attr()` method is used to read a table column value into a program register. The column may be specified either by using its attribute ID or as a pointer to a `Column` object.

Signature. This method can be called in either of two ways. The first of these is by referencing the column by its attribute ID, as shown here:

```
int read_attr
    (
      Uint32 RegDest,
      Uint32 attrId
    )
```

Alternatively, you can reference the column as a `Column` object, as shown here:

```
int read_attr
    (
      Uint32 RegDest,
      const NdbDictionary::Column* column
    )
```

Parameters. This method takes two parameters, as described here:

- The register to which the column value is to be copied (*`RegDest`*).
- Either of the following references to the table column whose value is to be copied:
  - The table column's attribute ID (*`attrId`*)
  - A pointer to a *`column`*—that is, a pointer to an `Column` object referencing the table column

Return value. Returns `0` on success, and `-1` on failure.

2.3.24.44 NdbInterpretedCode::ret_sub()

Description. This method marks the end of the current subroutine.

Signature.

```
int ret_sub
    (
      void
    )
```

Parameters. *None*.

Return value. Returns `0` on success, `-1` otherwise.

2.3.24.45 NdbInterpretedCode::sub_reg()

Description. This method gets the difference between the values stored in any two given registers and stores the result in a third register.

Signature.

```
int sub_reg
    (
      Uint32 RegDest,
      Uint32 RegSource1,
      Uint32 RegSource2
    )
```

Parameters. This method takes three parameters. The first of these is the register in which the result is to be stored (*`RegDest`*). The second and third parameters (*`RegSource1`*and *`RegSource2`*) are the registers whose values are to be subtracted. In other words, the value of register *`RegDest`* is calculated as the value of the expression shown here:

```
(value in register RegSource1) - (value in register RegSource2)
```

Note

It is possible to re-use one of the registers whose values are subtracted for storing the result; that is, *RegDest* can be the same as *RegSource1* or *RegSource2*.

Return value. `0` on success; `-1` on failure.

2.3.24.46 NdbInterpretedCode::sub_val()

Description. This method subtracts a specified value from the value of a given table column, and places the original and modified column values in registers 6 and 7. It is equivalent to the following series of `NdbInterpretedCode` method calls, where *attrId* is the table column' attribute ID and *aValue* is the value to be subtracted:

```
read_attr(6, attrId);
load_const_u32(7, aValue);
sub_reg(7,6,7);
write_attr(attrId, 7);
```

aValue can be a 32-bit or 64-bit integer.

Signature. This method can be invoked in either of two ways, depending on whether *aValue* is 32-bit or 64-bit.

32-bit *aValue*:

```
int sub_val
    (
      Uint32 attrId,
      Uint32 aValue
    )
```

64-bit *aValue*:

```
int sub_val
    (
      Uint32 attrId,
      Uint64 aValue
    )
```

Parameters. A table column attribute ID and a 32-bit or 64-bit integer value to be subtracted from this column value.

Return value. Returns `0` on success, `-1` on failure.

2.3.24.47 NdbInterpretedCode::write_attr()

Description. This method is used to copy a register value to a table column. The column may be specified either by using its attribute ID or as a pointer to a `Column` object.

Signature. This method can be invoked in either of two ways. The first of these is requires referencing the column by its attribute ID, as shown here:

```
int read_attr
    (
      Uint32 attrId,
      Uint32 RegSource
```

```
    )
```

You can also reference the column as a `Column` object instead, like this:

```
int read_attr
    (
      const NdbDictionary::Column* column,
      Uint32 RegSource
    )
```

Parameters. This method takes two parameters as follows:

- A reference to the table column to which the register value is to be copied. This can be either of the following:
  - The table column's attribute ID (*`attrId`*)
  - A pointer to a *`column`*—that is, a pointer to an `Column` object referencing the table column
- The register whose value is to be copied (*`RegSource`*).

Return value. Returns `0` on success; `-1` on failure.

2.3.25 The NdbOperation Class

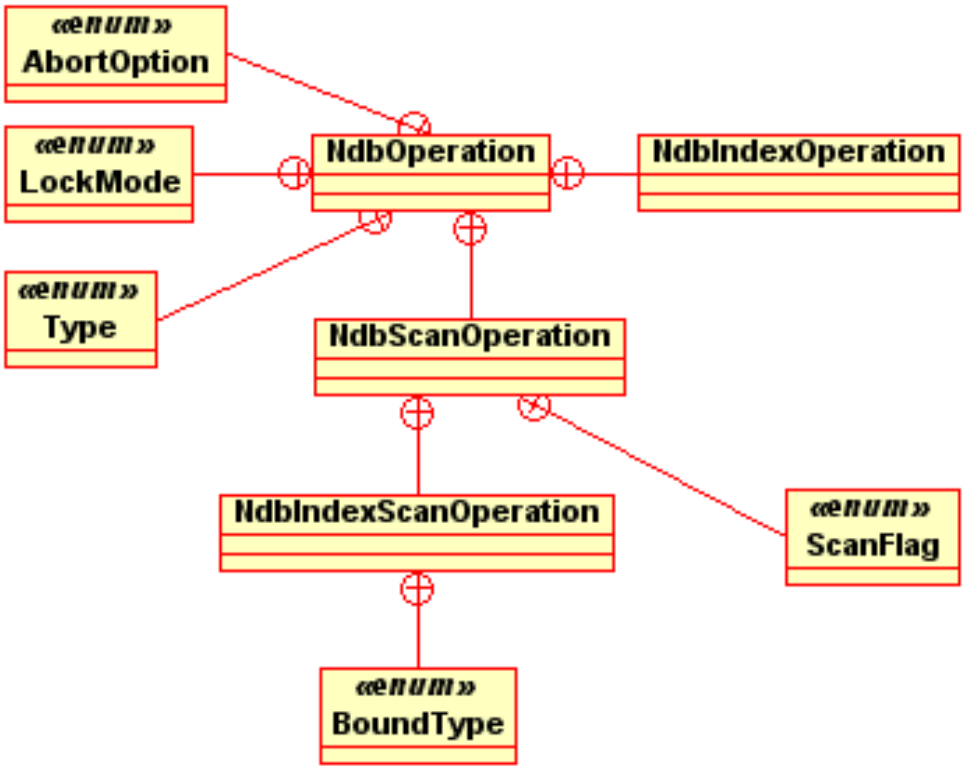
This section discusses the `NdbOperation` class.

Parent class. *None*

Child classes. `NdbIndexOperation`, `NdbScanOperation`

NdbOperation Subclasses. The following diagram shows the relationships of `NdbOperation`, its subclasses, and their public types:

Figure 2.20 NdbOperation subclasses



Description. `NdbOperation` represents a “generic” data operation. Its subclasses represent more specific types of operations. See Section 2.3.25.18, “NdbOperation::Type” for a listing of operation types and their corresponding `NdbOperation` subclasses.

Methods. The following table lists the public methods of this class and the purpose or use of each method:

Method	Purpose / Use
`deleteTuple()`	Removes a tuple from a table
`equal()`	Defines a search condition using equality
`getBlobHandle()`	Used to access blob attributes
`getLockHandle()`	Gets a lock handle for the operation
`getLockMode()`	Gets the operation's lock mode
`getNdbError()`	Gets the latest error
`getNdbErrorLine()`	Gets the number of the method where the latest error occurred
`getTableName()`	Gets the name of the table used for this operation
`getTable()`	Gets the table object used for this operation
`getNdbTransaction()`	Gets the `NdbTransaction` object for this operation
`getType()`	Gets the type of operation
`getValue()`	Allocates an attribute value holder for later access
`insertTuple()`	Adds a new tuple to a table
`readTuple()`	Reads a tuple from a table
`setValue()`	Defines an attribute to set or update
`updateTuple()`	Updates an existing tuple in a table
`writeTuple()`	Inserts or updates a tuple

Note

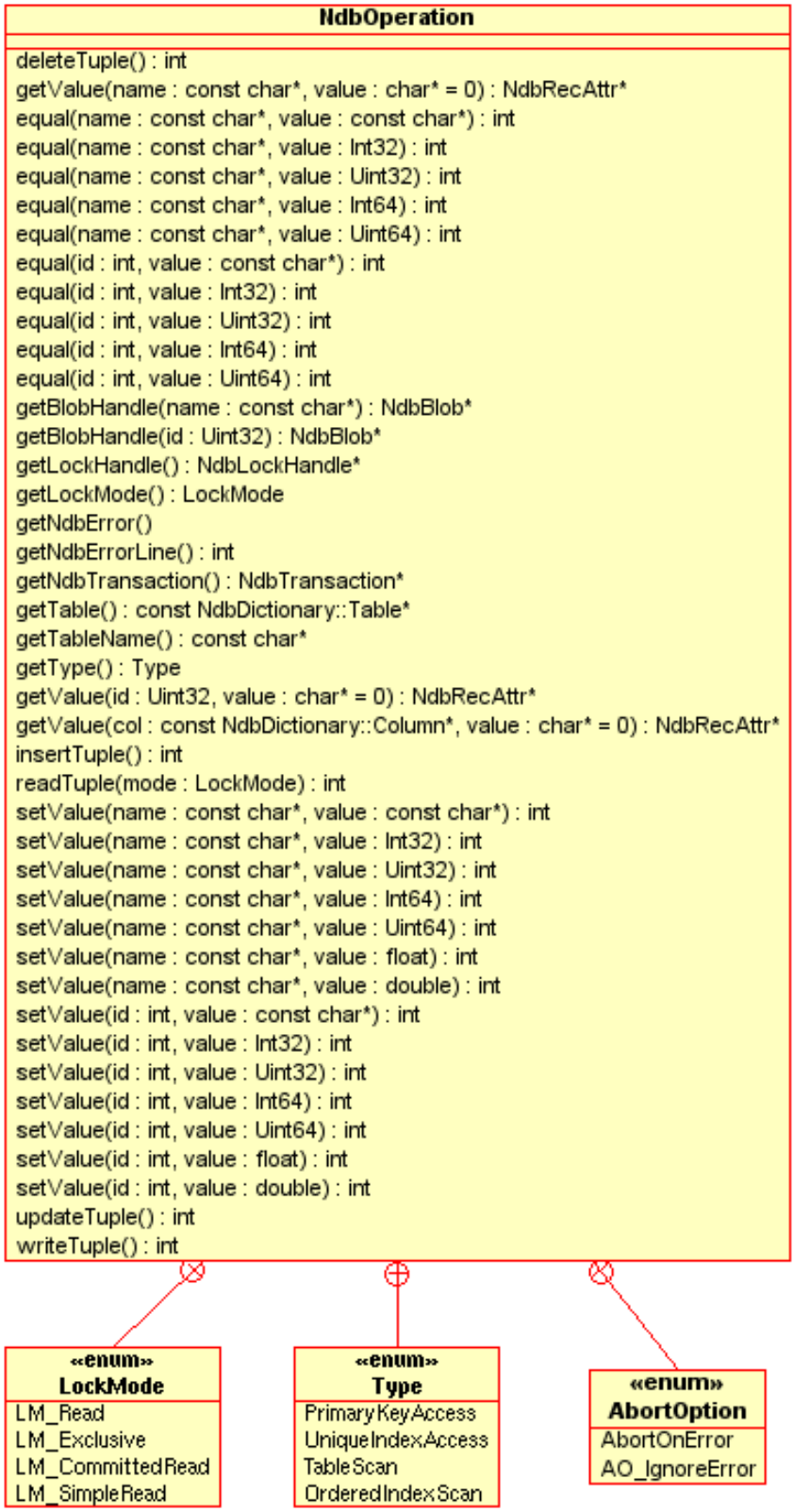
This class has no public constructor. To create an instance of `NdbOperation`, you must use `NdbTransaction::getNdbOperation()`.

Types. The `NdbOperation` class defines three public types, shown in the following table:

Type	Purpose / Use
`AbortOption()`	Determines whether a failed operation causes failure of the transaction of which it is part
`LockMode()`	The type of lock used when performing a read operation
`Type()`	Operation access types

Class diagram. The following diagram shows all the available methods and enumerated types of the `NdbOperation` class:

Figure 2.21 NdbOperation



Note

For more information about the use of `NdbOperation`, see Single-row operations.

2.3.25.1 NdbOperation::AbortOption

Description. This type is used to determine whether failed operations should force a transaction to be aborted. It is used as an argument to the `execute()` method—see Section 2.3.30.6, "NdbTransaction::execute()", for more information.

Enumeration values. Possible values are shown, along with descriptions, in the following table:

Value	Description
`AbortOnError`	A failed operation causes the transaction to abort.
`AO_IgnoreOnError`	Failed operations are ignored; the transaction continues to execute.
`DefaultAbortOption`	The `AbortOption` value is set according to the operation type: • Read operations: `AO_IgnoreOnError` • Scan takeover or DML operations: `AbortOnError`

See Section 2.3.30.6, "NdbTransaction::execute()", for more information.

2.3.25.2 NdbOperation::deleteTuple()

Description. This method defines the `NdbOperation` as a `DELETE` operation. When the `NdbTransaction::execute()` method is invoked, the operation deletes a tuple from the table.

Signature.

```
virtual int deleteTuple
    (
      void
    )
```

Parameters. *None*.

Return value. Returns `0` on success, `-1` on failure.

2.3.25.3 NdbOperation::equal()

Description. This method defines a search condition with an equality. The condition is true if the attribute has the given value. To set search conditions on multiple attributes, use several calls to `equal()`; in such cases all of them must be satisfied for the tuple to be selected.

Important

If the attribute is of a fixed size, its value must include all bytes. In particular a `Char` value must be native-space padded. If the attribute is of variable size, its value must start with 1 or 2 little-endian length bytes (2 if its type is `Long*`).

Note

When using `insertTuple()`, you may also define the search key with `setValue()`. See Section 2.3.25.17, "NdbOperation::setValue()".

Signature. There are 10 versions of `equal()`, each having slightly different parameters. All of these are listed here:

```
int  equal
```

```
    (
      const char* name,
      const char* value
    )

int  equal
    (
      const char* name,
      Int32       value
    )

int  equal
    (
      const char* name,
      Uint32      value
    )

int  equal
    (
      const char* name,
      Int64       value
    )

int  equal
    (
      const char* name,
      Uint64      value
    )

int  equal
    (
      Uint32      id,
      const char* value
    )

int  equal
    (
      Uint32 id,
      Int32  value
    )

int  equal
    (
      Uint32 id,
      Uint32 value
    )

int  equal
    (
      Uint32 id,
      Int64  value
    )

int  equal
    (
      Uint32 id,
      Uint64 value
    )
```

Parameters. This method requires two parameters:

- The first parameter can be either of the following:
  1. The *name* of the attribute (a string)
  2. The *id* of the attribute (an unsigned 32-bit integer)
- The second parameter is the attribute *value* to be tested. This value can be any one of the following 5 types:
  1. String

2. 32-bit integer
3. Unsigned 32-bit integer
4. 64-bit integer
5. Unsigned 64-bit integer

Return value. Returns `-1` in the event of an error.

2.3.25.4 NdbOperation::getBlobHandle()

Description. This method is used in place of `getValue()` or `setValue()` for blob attributes. It creates a blob handle (`NdbBlob` object). A second call with the same argument returns the previously created handle. The handle is linked to the operation and is maintained automatically.

Signature. This method has two forms, depending on whether it is called with the name or the ID of the blob attribute:

```
virtual NdbBlob* getBlobHandle
    (
      const char* name
    )
```

or

```
virtual NdbBlob* getBlobHandle
    (
      Uint32 id
    )
```

Parameters. This method takes a single parameter, which can be either one of the following:

- The *`name`* of the attribute
- The *`id`* of the attribute

Return value. Regardless of parameter type used, this method return a pointer to an instance of `NdbBlob`.

2.3.25.5 NdbOperation::getLockHandle

Description. Returns a pointer to the current operation's lock handle. When used with `NdbRecord`, the lock handle must first be requested with the `OO_LOCKHANDLE` operation option. For other operations, this method can be used alone. In any case, the `NdbLockHandle` object returned by this method cannot be used until the operation has been executed.

Using lock handle methods. Shared or exclusive locks taken by read operations in a transaction are normally held until the transaction commits or aborts. Such locks can be released before a transaction commits or aborts by requesting a lock handle when defining the read operation. Once the read operation has been executed, an `NdbLockHandle` can be used to create a new unlock operation (with `NdbTransaction::unlock()`). When the unlock operation is executed, the row lock placed by the read operation is released.

The steps required to release these locks are listed here:

- Define the primary key read operation in the normal way with lock mode `LM_Read` or `LM_Exclusive`.
- Call `NdbOperation::getLockHandle()` during operation definition, or, for `Ndbrecord`, set the `OO_LOCKHANDLE` operation option when calling `NdbTransaction::readTuple()`.

- Call `NdbTransaction::execute()`; the row is now locked from this point on, as normal.
- (Use data, possibly making calls to `NdbTransaction::execute()`.)
- Call `NdbTransaction::unlock()`, passing in the `const NdbLockHandle` obtained previously to create an unlock operation.
- Call `NdbTransaction::execute()`; this unlocks the row.

Notes:

- As with other operation types, unlock operations can be batched.
- Each `NdbLockHandle` object refers to a lock placed on a row by a single primary key read operation. A single row in the database may have concurrent multiple lock holders (mode `LM_Read`) and may have multiple lock holders pending (`LM_Exclusive`), so releasing the claim of one lock holder may not result in a change to the observable lock status of the row.
- Lock handles are supported for scan lock takeover operations; the lock handle must be requested before the lock takeover is executed.
- Lock handles and unlock operations are not supported for unique index read operations.

Signature.

```
const NdbLockHandle* getLockHandle
    (
      void
    ) const
```

(or)

```
const NdbLockHandle* getLockHandle
    (
      void
    )
```

Parameters. *None*.

Return value. Pointer to an `NdbLockHandle` that can be used by the `NdbTransaction` methods `unlock()` and `releaseLockHandle()`.

2.3.25.6 NdbOperation::getLockMode()

Description. This method gets the operation's lock mode.

Signature.

```
LockMode getLockMode
    (
      void
    ) const
```

Parameters. *None*.

Return value. A `LockMode` value. See Section 2.3.25.15, “NdbOperation::LockMode”.

2.3.25.7 NdbOperation::getNdbError()

Description. This method gets the most recent error (an `NdbError` object).

Signature.

```
const NdbError& getNdbError
    (
      void
    ) const
```

Parameters. *None*.

Return value. An `NdbError` object.

2.3.25.8 NdbOperation::getNdbErrorLine()

Description. This method retrieves the method number in which the latest error occurred.

Signature. This method can and should be used as shown here:

```
int getNdbErrorLine
    (
      void
    ) const
```

Parameters. *None*.

Return value. The method number (an integer).

2.3.25.9 NdbOperation::getTable()

Description. This method is used to retrieve the table object associated with the operation.

Signature.

```
const NdbDictionary::Table* getTable
    (
      void
    ) const
```

Parameters. *None*.

Return value. A pointer to an instance of `Table`.

2.3.25.10 NdbOperation::getTableName()

Description. This method retrieves the name of the table used for the operation.

Signature.

```
const char* getTableName
    (
      void
    ) const
```

Parameters. *None*.

Return value. The name of the table.

2.3.25.11 NdbOperation::getNdbTransaction()

Description. Gets the `NdbTransaction` object for this operation.

Signature.

```
virtual NdbTransaction* getNdbTransaction
    (
      void
    ) const
```

Parameters. *None*.

Return value. A pointer to an `NdbTransaction` object.

2.3.25.12 NdbOperation::getType()

Description. This method is used to retrieve the access type for this operation.

Signature.

```
Type getType
    (
      void
    ) const
```

Parameters. *None*.

Return value. A `Type` value.

2.3.25.13 NdbOperation::getValue()

Description. This method prepares for the retrieval of an attribute value. The NDB API allocates memory for an `NdbRecAttr` object that is later used to obtain the attribute value. This can be done by using one of the many `NdbRecAttr` accessor methods, the exact method to be used depending on the attribute's data type. (This includes the generic `NdbRecAttr::aRef()` method, which retrieves the data as `char*`, regardless of its actual type. However, this is not type-safe, and requires a cast from the user.)

Important

This method does *not* fetch the attribute value from the database; the `NdbRecAttr` object returned by this method is not readable or printable before calling `NdbTransaction::execute()`.

If a specific attribute has not changed, the corresponding `NdbRecAttr` has the state `UNDEFINED`. This can be checked by using `NdbRecAttr::isNULL()`, which in such cases returns `-1`.

See Section 2.3.30.6, “NdbTransaction::execute()”, and Section 2.3.26.13, “NdbRecAttr::isNULL()”.

Signature. There are three versions of this method, each having different parameters:

```
NdbRecAttr* getValue
    (
      const char* name,
      char*       value = 0
    )

NdbRecAttr* getValue
    (
      Uint32 id,
      char*  value = 0
    )

NdbRecAttr* getValue
    (
      const NdbDictionary::Column* col,
      char*                        value = 0
    )
```

Parameters. All three forms of this method have two parameters, the second parameter being optional (defaults to `0`). They differ only with regard to the type of the first parameter, which can be any one of the following:

- The attribute *name*
- The attribute *id*
- The table *column* on which the attribute is defined

In all three cases, the second parameter is a character buffer in which a non-NULL attribute value is returned. In the event that the attribute is NULL, is it stored only in the NdbRecAttr object returned by this method.

If no *value* is specified in the getValue() method call, or if 0 is passed as the value, then the NdbRecAttr object provides memory management for storing the received data. If the maximum size of the received data is above a small fixed size, malloc() is used to store it: For small sizes, a small, fixed internal buffer (32 bytes in extent) is provided. This storage is managed by the NdbRecAttr instance; it is freed when the operation is released, such as at transaction close time; any data written here that you wish to preserve should be copied elsewhere before this freeing of memory takes place.

If you pass a non-zero pointer for *value*, then it is assumed that this points to an portion of memory which is large enough to hold the maximum value of the column; any returned data is written to that location. The pointer should be at least 32-bit aligned.

Note

Index columns cannot be used in place of table columns with this method. In cases where a table column is not available, you can use the attribute name, obtained with getName(), for this purpose instead.

Return value. A pointer to an NdbRecAttr object to hold the value of the attribute, or a NULL pointer, indicating an error.

Retrieving integers. Integer values can be retrieved from both the *value* buffer passed as this method's second parameter, and from the NdbRecAttr object itself. On the other hand, character data is available from NdbRecAttr if no buffer has been passed in to getValue() (see Section 2.3.26.2, “NdbRecAttr::aRef()”). However, character data is written to the buffer only if one is provided, in which case it cannot be retrieved from the NdbRecAttr object that was returned. In the latter case, NdbRecAttr::aRef() returns a buffer pointing to an empty string.

Accessing bit values. The following example shows how to check a given bit from the *value* buffer. Here, op is an operation (NdbOperation object), name is the name of the column from which to get the bit value, and trans is an NdbTransaction object:

```
Uint32 buf[];

op->getValue(name, buf); /* bit column */

trans->execute();

if(buf[X/32] & 1 << (X & 31)) /* check bit X */
{
  /* bit X set */
}
```

2.3.25.14 NdbOperation::insertTuple()

Description. This method defines the NdbOperation to be an INSERT operation. When the NdbTransaction::execute() method is called, this operation adds a new tuple to the table.

Signature.

```
virtual int insertTuple
    (
```

```
      void
    )
```

Parameters. *None*.

Return value. Returns `0` on success, `-1` on failure.

2.3.25.15 NdbOperation::LockMode

Description. This type describes the lock mode used when performing a read operation.

Enumeration values. Possible values for this type are shown, along with descriptions, in the following table:

Value	Description
`LM_Read`	Read with shared lock
`LM_Exclusive`	Read with exclusive lock
`LM_CommittedRead`	Ignore locks; read last committed
`LM_SimpleRead`	Read with shared lock, but release lock directly

Note

There is also support for dirty reads (`LM_Dirty`), but this is normally for internal purposes only, and should not be used for applications deployed in a production setting.

2.3.25.16 NdbOperation::readTuple()

Description. This method defines the `NdbOperation` as a `READ` operation. When the `NdbTransaction::execute()` method is invoked, the operation reads a tuple.

Signature.

```
virtual int readTuple
    (
      LockMode mode
    )
```

Parameters. *`mode`* specifies the locking mode used by the read operation. See Section 2.3.25.15, “NdbOperation::LockMode”, for possible values.

Return value. Returns `0` on success, `-1` on failure.

2.3.25.17 NdbOperation::setValue()

Description. This method defines an attribute to be set or updated.

There are a number of `NdbOperation::setValue()` methods that take a certain type as input (pass by value rather than passing a pointer). It is the responsibility of the application programmer to use the correct types.

However, the NDB API does check that the application sends a correct length to the interface as given in the length parameter. A `char*` value can contain any data type or any type of array. If the length is not provided, or if it is set to zero, then the API assumes that the pointer is correct, and does not check it.

To set a `NULL` value, use the following construct:

```
setValue("ATTR_NAME", (char*)NULL);
```

When you use `insertTuple()`, the NDB API automatically detects that it is supposed to use `equal()` instead.

In addition, it is not necessary when using `insertTuple()` to use `setValue()` on key attributes before other attributes.

Signature. There are 14 versions of `NdbOperation::setValue()`, each with slightly different parameters, as listed here (and summarized in the *Parameters* section following):

```
int setValue
    (
      const char* name,
      const char* value
)

int setValue
    (
      const char* name,
      Int32       value
    )

int setValue
    (
      const char* name,
      Uint32      value
    )

int setValue
    (
      const char* name,
      Int64       value
    )

int setValue
    (
      const char* name,
      Uint64      value
    )

int setValue
    (
      const char* name,
      float       value
    )

int setValue
    (
      const char* name,
      double      value
    )

int setValue
    (
      Uint32      id,
      const char* value
    )

int setValue
    (
      Uint32 id,
      Int32  value
    )

int setValue
    (
      Uint32 id,
      Uint32 value
    )

int setValue
```

```
    (
      Uint32 id,
      Int64  value
    )

int setValue
    (
      Uint32 id,
      Uint64 value
    )

int setValue
    (
      Uint32 id,
      float  value
    )

int setValue
    (
      Uint32 id,
      double value
    )
```

Parameters. This method requires the following two parameters:

- The first parameter identified the attribute to be set, and may be either one of the following:
  1. The attribute *`name`* (a string)
  2. The attribute *`id`* (an unsigned 32-bit integer)
- The second parameter is the *`value`* to which the attribute is to be set; its type may be any one of the following 7 types:
  1. String (`const char*`)
  2. 32-bit integer
  3. Unsigned 32-bit integer
  4. 64-bit integer
  5. Unsigned 64-bit integer
  6. Double
  7. Float

See Section 2.3.25.3, “NdbOperation::equal()”, for important information regarding the value's format and length.

Return value. Returns `-1` in the event of failure.

2.3.25.18 NdbOperation::Type

Description. `Type` is used to describe the operation access type. Each access type is supported by `NdbOperation` or one of its subclasses, as shown in the following table:

Enumeration values. Possible values are shown, along with descriptions, in the following table:

Value	Description	Class
`PrimaryKeyAccess`	A read, insert, update, or delete operation using the table's primary key	`NdbOperation`

Value	Description	Class
`UniqueIndexAccess`	A read, update, or delete operation using a unique index	`NdbIndexOperation`
`TableScan`	A full table scan	`NdbScanOperation`
`OrderedIndexScan`	An ordered index scan	`NdbIndexScanOperation`

2.3.25.19 NdbOperation::writeTuple()

Description. This method defines the `NdbOperation` as a `WRITE` operation. When the `NdbTransaction::execute()` method is invoked, the operation writes a tuple to the table. If the tuple already exists, it is updated; otherwise an insert takes place.

Signature.

```
virtual int writeTuple
    (
      void
    )
```

Parameters. *None*.

Return value. Returns `0` on success, `-1` on failure.

2.3.25.20 NdbOperation::updateTuple()

Description. This method defines the `NdbOperation` as an `UPDATE` operation. When the `NdbTransaction::execute()` method is invoked, the operation updates a tuple found in the table.

Signature.

```
virtual int updateTuple
    (
      void
    )
```

Parameters. *None*.

Return value. Returns `0` on success, `-1` on failure.

2.3.26 The NdbRecAttr Class

The section describes the `NdbRecAttr` class and its public methods.

Parent class. *None*

Child classes. *None*

Description. `NdbRecAttr` contains the value of an attribute. An `NdbRecAttr` object is used to store an attribute value after it has been retrieved using the `NdbOperation::getValue()` method. This object is allocated by the NDB API. A brief example is shown here:

```
MyRecAttr = MyOperation->getValue("ATTR2", NULL);

if(MyRecAttr == NULL)
  goto error;

if(MyTransaction->execute(Commit) == -1)
  goto error;
```

```
ndbout << MyRecAttr->u_32_value();
```

For additional examples, see Section 2.5.1, “NDB API Example Using Synchronous Transactions”.

Note

An `NdbRecAttr` object is instantiated with its value only when `NdbTransaction::execute()` is invoked. Prior to this, the value is undefined. (Use `NdbRecAttr::isNULL()` to check whether the value is defined.) This means that an `NdbRecAttr` object has valid information only between the times that `NdbTransaction::execute()` and `Ndb::closeTransaction()` are called. The value of the `NULL` indicator is `-1` until the `NdbTransaction::execute()` method is invoked.

Methods. `NdbRecAttr` has a number of methods for retrieving values of various simple types directly from an instance of this class.

Note

It is also possible to obtain a reference to the value regardless of its actual type, by using `NdbRecAttr::aRef()`; however, you should be aware that this is not type-safe, and requires a cast from the user.

The following table lists all of the public methods of this class and the purpose or use of each method:

Method	Purpose / Use
`~NdbRecAttr()`	Destructor method
`aRef()`	Gets a pointer to the attribute value
`char_value()`	Retrieves a `Char` attribute value
`clone()`	Makes a deep copy of the `RecAttr` object
`double_value()`	Retrieves a `Double` attribute value, as a double (8 bytes)
`float_value()`	Retrieves a `Float` attribute value, as a float (4 bytes)
`get_size_in_bytes()`	Gets the size of the attribute, in bytes
`getColumn()`	Gets the column to which the attribute belongs
`getType()`	Gets the attribute's type (`Column::Type`)
`isNULL()`	Tests whether the attribute is `NULL`
`int8_value()`	Retrieves a `Tinyint` attribute value, as an 8-bit integer
`int32_value()`	Retrieves an `Int` attribute value, as a 32-bit integer
`int64_value()`	Retrieves a `Bigint` attribute value, as a 64-bit integer
`medium_value()`	Retrieves a `Mediumint` attribute value, as a 32-bit integer
`short_value()`	Retrieves a `Smallint` attribute value, as a 16-bit integer
`u_8_value()`	Retrieves a `Tinyunsigned` attribute value, as an unsigned 8-bit integer
`u_32_value()`	Retrieves an `Unsigned` attribute value, as an unsigned 32-bit integer
`u_64_value()`	Retrieves a `Bigunsigned` attribute value, as an unsigned 64-bit integer
`u_char_value()`	Retrieves a `Char` attribute value, as an unsigned `char`
`u_medium_value()`	Retrieves a `Mediumunsigned` attribute value, as an unsigned 32-bit integer
`u_short_value()`	Retrieves a `Smallunsigned` attribute value, as an unsigned 16-bit integer

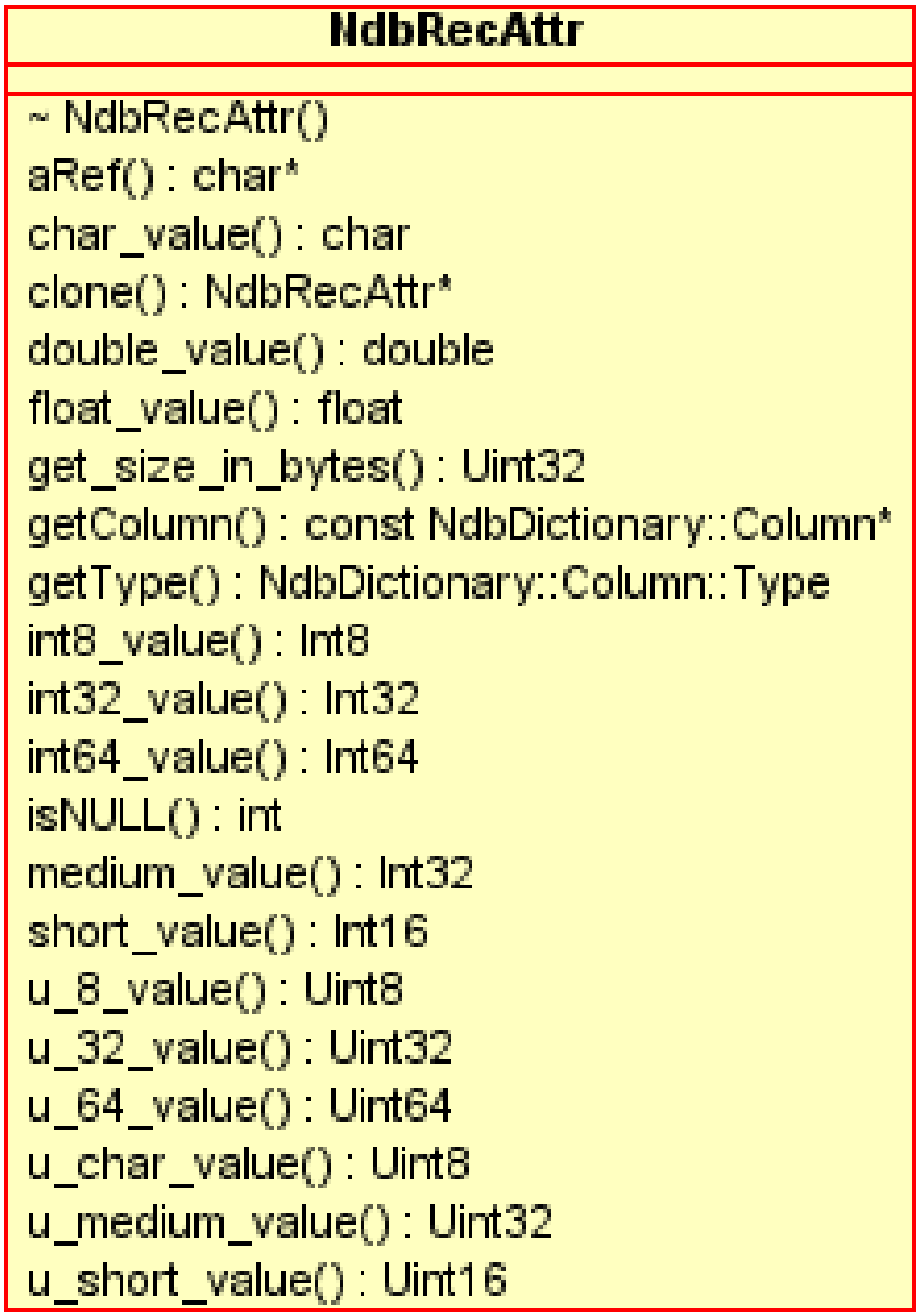
Note

The `NdbRecAttr` class has no public constructor; an instance of this object is created using `NdbTransaction::execute()`. For information about the destructor, which is public, see Section 2.3.26.1, "~NdbRecAttr()".

Types. The `NdbRecAttr` class defines no public types.

Class diagram. This diagram shows all the available methods of the `NdbRecAttr` class:

Figure 2.22 NdbRecAttr



2.3.26.1 ~NdbRecAttr()

Description. The `NdbRecAttr` class destructor method.

Important

You should delete only copies of `NdbRecAttr` objects that were created in your application using the `clone()` method.

Signature.

```
~NdbRecAttr
    (
      void
    )
```

Parameters. *None*.

Return value. *None*.

2.3.26.2 NdbRecAttr::aRef()

Description. This method is used to obtain a reference to an attribute value, as a `char` pointer. This pointer is aligned appropriately for the data type. The memory is released by the NDB API when `NdbTransaction::close()` is executed on the transaction which read the value.

Signature.

```
char* aRef
    (
      void
    ) const
```

Parameters. A pointer to the attribute value. Because this pointer is constant, this method can be called anytime after `NdbOperation::getValue()` has been called.

Return value. *None*.

2.3.26.3 NdbRecAttr::char_value()

Description. This method gets a `Char` value stored in an `NdbRecAttr` object, and returns it as a `char`.

Signature.

```
char char_value
    (
      void
    ) const
```

Parameters. *None*.

Return value. A `char` value.

2.3.26.4 NdbRecAttr::clone()

Description. This method creates a deep copy of an `NdbRecAttr` object.

Note

The copy created by this method should be deleted by the application when no longer needed.

Signature.

```
NdbRecAttr* clone
    (
      void
    ) const
```

Parameters. *None*.

Return value. An `NdbRecAttr` object. This is a complete copy of the original, including all data.

2.3.26.5 NdbRecAttr::double_value()

Description. This method gets a `Double` value stored in an `NdbRecAttr` object, and returns it as a double.

Signature.

```
double double_value
    (
      void
    ) const
```

Parameters. *None*.

Return value. A double (8 bytes).

2.3.26.6 NdbRecAttr::float_value()

Description. This method gets a `Float` value stored in an `NdbRecAttr` object, and returns it as a float.

Signature.

```
float float_value
    (
      void
    ) const
```

Parameters. *None*.

Return value. A float (4 bytes).

2.3.26.7 NdbRecAttr::get_size_in_bytes()

Description. You can use this method to obtain the size of an attribute (element).

Signature.

```
Uint32 get_size_in_bytes
    (
      void
    ) const
```

Parameters. *None*.

Return value. The attribute size in bytes, as an unsigned 32-bit integer.

2.3.26.8 NdbRecAttr::getColumn()

Description. This method is used to obtain the column to which the attribute belongs.

Signature.

```
const NdbDictionary::Column* getColumn
    (
      void
    ) const
```

Parameters. *None*.

Return value. A pointer to a `Column` object.

2.3.26.9 NdbRecAttr::getType()

Description. This method is used to obtain the column's data type.

Signature.

```
NdbDictionary::Column::Type getType
    (
      void
    ) const
```

Parameters. *None*.

Return value. An `Column::Type` value.

2.3.26.10 NdbRecAttr::int8_value()

Description. This method gets a `Small` value stored in an `NdbRecAttr` object, and returns it as an 8-bit signed integer.

Signature.

```
Int8 int8_value
    (
      void
    ) const
```

Parameters. *None*.

Return value. An 8-bit signed integer.

2.3.26.11 NdbRecAttr::int32_value()

Description. This method gets an `Int` value stored in an `NdbRecAttr` object, and returns it as a 32-bit signed integer.

Signature.

```
Int32 int32_value
    (
      void
    ) const
```

Parameters. *None*.

Return value. A 32-bit signed integer.

2.3.26.12 NdbRecAttr::int64_value()

Description. This method gets a `Bigint` value stored in an `NdbRecAttr` object, and returns it as a 64-bit signed integer.

Signature.

```
Int64 int64_value
    (
      void
    ) const
```

Parameters. *None*.

Return value. A 64-bit signed integer.

2.3.26.13 NdbRecAttr::isNULL()

Description. This method checks whether an attribute value is `NULL`.

Signature.

```
int isNULL
    (
      void
    ) const
```

Parameters. *None*.

Return value. One of the following three values:

- `-1`: The attribute value is not defined due to an error.
- `0`: The attribute value is defined, but is not `NULL`.
- `1`: The attribute value is defined and is `NULL`.

Important

In the event that `NdbTransaction::execute()` has not yet been called, the value returned by `isNULL()` is not determined.

2.3.26.14 NdbRecAttr::medium_value()

Description. Gets the value of a `Mediumint` value stored in an `NdbRecAttr` object, and returns it as a 32-bit signed integer.

Signature.

```
Int32 medium_value
    (
      void
    ) const
```

Parameters. *None*.

Return value. A 32-bit signed integer.

2.3.26.15 NdbRecAttr::short_value()

Description. This method gets a `Smallint` value stored in an `NdbRecAttr` object, and returns it as a 16-bit signed integer (short).

Signature.

```
short short_value
    (
      void
    ) const
```

Parameters. *None*.

Return value. A 16-bit signed integer.

2.3.26.16 NdbRecAttr::u_8_value()

Description. This method gets a `Smallunsigned` value stored in an `NdbRecAttr` object, and returns it as an 8-bit unsigned integer.

Signature.

```
Uint8 u_8_value
    (
      void
    ) const
```

Parameters. *None*.

Return value. An 8-bit unsigned integer.

2.3.26.17 NdbRecAttr::u_32_value()

Description. This method gets an `Unsigned` value stored in an `NdbRecAttr` object, and returns it as a 32-bit unsigned integer.

Signature.

```
Uint32 u_32_value
    (
      void
    ) const
```

Parameters. *None*.

Return value. A 32-bit unsigned integer.

2.3.26.18 NdbRecAttr::u_64_value()

Description. This method gets a `Bigunsigned` value stored in an `NdbRecAttr` object, and returns it as a 64-bit unsigned integer.

Signature.

```
Uint64 u_64_value
    (
      void
```

```
    ) const
```

Parameters. *None*.

Return value. A 64-bit unsigned integer.

2.3.26.19 NdbRecAttr::u_char_value()

Description. This method gets a `Char` value stored in an `NdbRecAttr` object, and returns it as an unsigned `char`.

Signature.

```
Uint8 u_char_value
    (
      void
    ) const
```

Parameters. *None*.

Return value. An 8-bit unsigned `char` value.

2.3.26.20 NdbRecAttr::u_medium_value()

Description. This method gets an `Mediumunsigned` value stored in an `NdbRecAttr` object, and returns it as a 32-bit unsigned integer.

Signature.

```
Uint32 u_medium_value
    (
      void
    ) const
```

Parameters. *None*.

Return value. A 32-bit unsigned integer.

2.3.26.21 NdbRecAttr::u_short_value()

Description. This method gets a `Smallunsigned` value stored in an `NdbRecAttr` object, and returns it as a 16-bit (short) unsigned integer.

Signature.

```
Uint16 u_short_value
    (
      void
    ) const
```

Parameters. *None*.

Return value. A short (16-bit) unsigned integer.

2.3.27 The NdbRecord Interface

`NdbRecord` is an interface which provides a mapping to a full or a partial record stored in `NDB`. In the latter case, it can be used in conjunction with a bitmap to assist in access.

`NdbRecord` has no API methods of its own; rather it acts as a handle that can be passed between various method calls for use in many different sorts of operations, including the following operation types:

- Unique key reads and primary key reads
- Table scans and index scans
- DML operations involving unique keys or primary keys
- Operations involving index bounds

The same `NdbRecord` can be used simultaneously in multiple operations, transactions, and threads.

An `NdbRecord` can be created in NDB API programs by calling the `createRecord()` method of the `Dictionary` class. In addition, a number of NDB API methods have additional declarations that enable the programmer to leverage `NdbRecord`:

- `NdbScanOperation::nextResult()`
- `NdbScanOperation::lockCurrentTuple()`
- `NdbScanOperation::updateCurrentTuple()`
- `NdbScanOperation::deleteCurrentTuple()`
- `Dictionary::createRecord()`
- `Dictionary::releaseRecord()`
- `NdbTransaction::readTuple()`
- `NdbTransaction::insertTuple()`
- `NdbTransaction::updateTuple()`
- `NdbTransaction::writeTuple()`
- `NdbTransaction::deleteTuple()`
- `NdbTransaction::scanTable()`
- `NdbTransaction::scanIndex()`

The following members of `NdbIndexScanOperation` and `NdbDictionary` can also be used with `NdbRecord` scans:

- `IndexBound` is a structure used to describe index scan bounds.
- `RecordSpecification` is a structure used to specify columns and range offsets.

You can also use `NdbRecord` in conjunction with the new `PartitionSpec` structure to perform scans that take advantage of partition pruning, by means of a variant of `NdbIndexScanOperation::setBound()` that was added in the same NDB Cluster releases.

2.3.28 The NdbScanFilter Class

This section discusses the `NdbScanFilter` class and its public members.

Parent class. *None*

Child classes. *None*

Description. `NdbScanFilter` provides an alternative means of specifying filters for scan operations.

Important

Prior to MySQL 5.1.14, the comparison methods of this class did not work with `BIT` values (see Bug #24503).

Development of this interface continues; the characteristics of the `NdbScanFilter` class are likely to change further in future releases.

Methods. The following table lists the public methods of this class and the purpose or use of each method:

Method	Purpose / Use
`NdbScanFilter()`	Constructor method
`~NdbScanFilter()`	Destructor method
`begin()`	Begins a compound (set of conditions)
`cmp()`	Compares a column value with an arbitrary value
`end()`	Ends a compound
`eq()`	Tests for equality
`ge()`	Tests for a greater-than-or-equal condition
`getNdbError()`	Provides access to error information
`getNdbOperation()`	Gets the associated `NdbOperation`
`gt()`	Tests for a greater-than condition
`isfalse()`	Defines a term in a compound as `FALSE`
`isnotnull()`	Tests whether a column value is not `NULL`
`isnull()`	Tests whether a column value is `NULL`
`istrue()`	Defines a term in a compound as `TRUE`
`le()`	Tests for a less-than-or-equal condition
`lt()`	Tests for a less-than condition
`ne()`	Tests for inequality

NdbScanFilter Integer Comparison Methods. `NdbScanFilter` provides several convenience methods which can be used in lieu of the `cmp()` method when the arbitrary value to be compared is an integer: `eq()`, `ge()`, `gt()`, `le()`, `lt()`, and `ne()`.

Each of these methods is essentially a wrapper for `cmp()` that includes an appropriate value of `BinaryCondition` for that method's *`condition`* parameter; for example, `NdbScanFilter::eq()` is defined like this:

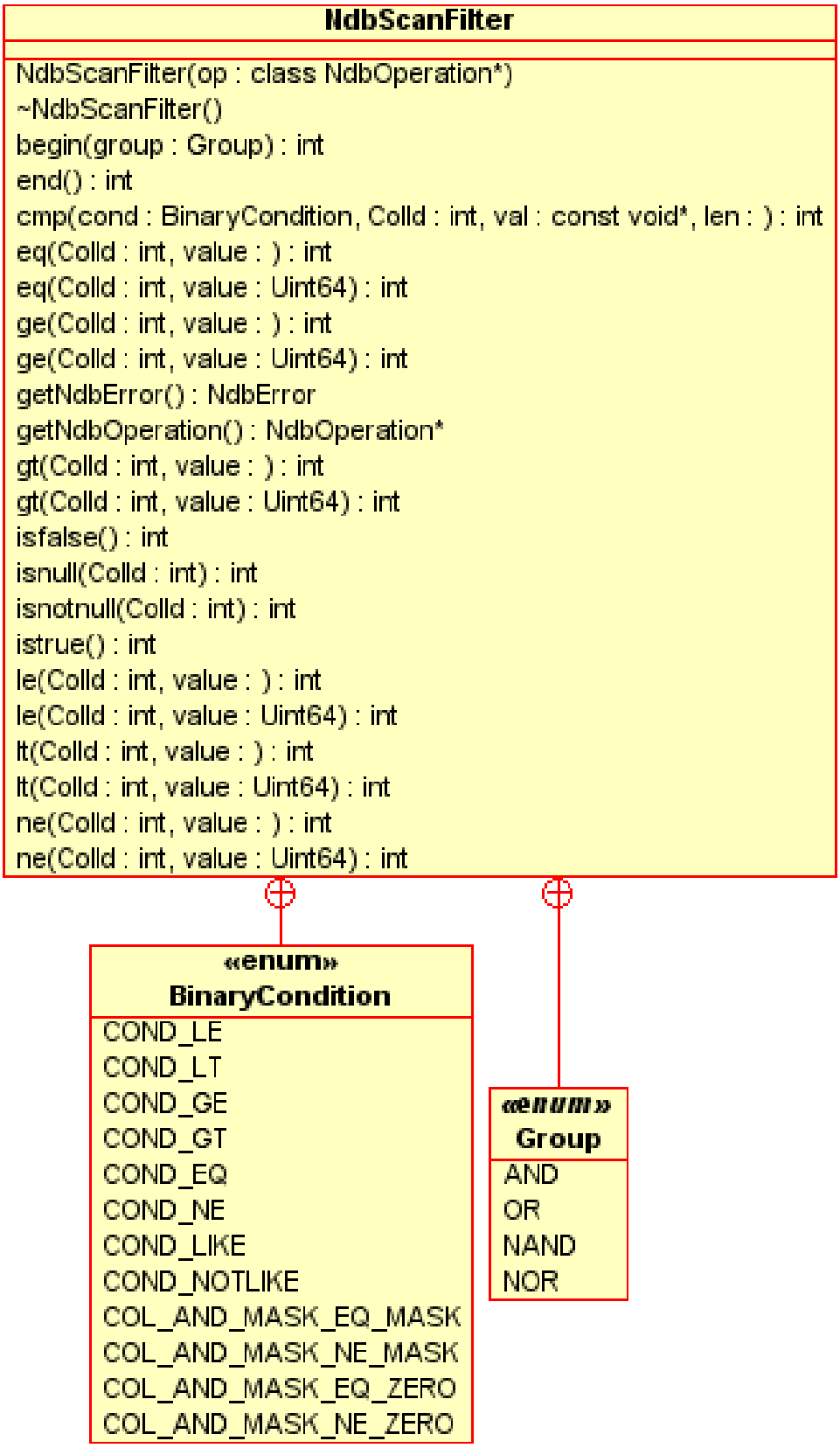
```
int eq(int columnId, Uint32 value)
{
  return cmp(BinaryCondition::COND_EQ, columnId, &value, 4);
}
```

Types. The `NdbScanFilter` class defines two public types:

- `BinaryCondition`: The type of condition, such as lower bound or upper bound.
- `Group`: A logical grouping operator, such as `AND` or `OR`.

Class diagram. This diagram shows all the public members of the `NdbScanFilter` class:

Figure 2.23 NdbScanFilter



2.3.28.1 NdbScanFilter::begin()

Description. This method is used to start a compound, and specifies the logical operator used to group the conditions making up the compound. The default is `AND`.

Signature.

```
int begin
    (
      Group group = AND
    )
```

Parameters. A `Group` value: one of `AND`, `OR`, `NAND`, or `NOR`. See Section 2.3.28.14, "NdbScanFilter::Group", for additional information.

Return value. `0` on success, `-1` on failure.

2.3.28.2 NdbScanFilter::BinaryCondition

Description. This type represents a condition based on the comparison of a column value with some arbitrary value—that is, a bound condition. A value of this type is used as the first argument to `NdbScanFilter::cmp()`.

Enumeration values. Possible values are shown, along with descriptions, in the following table:

Enumeration value	Description	Type of column values compared
`COND_EQ`	Equality (=)	any
`COND_NE`	Inequality (<> or !=)	any
`COND_LE`	Lower bound (<=)	any
`COND_LT`	Strict lower bound (<)	any
`COND_GE`	Upper bound (>=)	any
`COND_GT`	Strict upper bound (>)	any
`COND_LIKE`	`LIKE` condition	string or binary
`COND_NOTLIKE`	`NOT LIKE` condition	string or binary
`COL_AND_MASK_EQ_MASK`	Column value `AND`ed with bitmask is equal to bitmask	`BIT`
`COL_AND_MASK_NE_MASK`	Column value `AND`ed with bitmask is not equal to bitmask	`BIT`
`COL_AND_MASK_EQ_ZERO`	Column value `AND`ed with bitmask is equal to zero	`BIT`
`COL_AND_MASK_NE_ZERO`	Column value `AND`ed with bitmask is not equal to zero	`BIT`

When used in comparisons with `COND_EQ`, `COND_NE`, `COND_LT`, `COND_LE`, `COND_GT`, or `COND_GE`, fixed-length character and binary column values must be prefixed with the column size, and must be padded to length. This is not necessary for such values when used in `COND_LIKE`, `COND_NOTLIKE`, `COL_AND_MASK_EQ_MASK`, `COL_AND_MASK_NE_MASK`, `COL_AND_MASK_EQ_ZERO`, or `COL_AND_MASK_NE_ZERO` comparisons.

String comparisons. Strings compared using `COND_LIKE` and `COND_NOTLIKE` can use the pattern metacharacters `%` and `_`. See Section 2.3.28.3, "NdbScanFilter::cmp()", for more information.

BIT comparisons. The `BIT` comparison operators are `COL_AND_MASK_EQ_MASK`, `COL_AND_MASK_NE_MASK`, `COL_AND_MASK_EQ_ZERO`, and `COL_AND_MASK_NE_ZERO`. Corresponding methods are available for `NdbInterpretedCode` and `NdbOperation`; for more information about these methods, see NdbInterpretedCode Bitwise Comparison Operations.

2.3.28.3 NdbScanFilter::cmp()

Description. This method is used to define a comparison between a given value and the value of a column. (This method does not actually execute the comparison, which is done later when performing the scan for which this `NdbScanFilter` is defined.)

Note

In many cases, where the value to be compared is an integer, you can instead use one of several convenience methods provided by `NdbScanFilter` for this purpose. See NdbScanFilter Integer Comparison Methods.

Signature.

```
int cmp
    (
      BinaryCondition condition,
      int columnId,
      const void* value,
      Uint32 length = 0
    )
```

Parameters. This method takes the following parameters:

- *`condition`*: This represents the condition to be tested which compares the value of the column having the column ID *`columnID`* with some arbitrary value. The *`condition`* is a `BinaryCondition` value; for permitted values and the relations that they represent, see Section 2.3.28.2, “NdbScanFilter::BinaryCondition”.

  The *`condition`* values `COND_LIKE` or `COND_NOTLIKE` are used to compare a column value with a string pattern.

- *`columnId`*: This is the column's identifier, which can be obtained using the `Column::getColumnNo()` method.

- *`value`*: The value to be compared, repesented as a pointer to `void`.

  When using a `COND_LIKE` or `COND_NOTLIKE` comparison condition, the *`value`* is treated as a string pattern. This string must not be padded or use a prefix. The string *`value`* can include the pattern metacharacters or “wildcard” characters `%` and `_`, which have the meanings shown here:

Metacharacter	Description
`%`	Matches zero or more characters
`_`	Matches exactly one character

  To match against a literal “%” or “_” character, use the backslash (`\`) as an escape character. To match a literal “\” character, use `\\`.

  Note

  These are the same wildcard characters that are supported by the SQL `LIKE` and `NOT LIKE` operators, and are interpreted in the same way. See String Comparison Functions, for more information.

- *`length`*: The length of the value to be compared. The default value is `0`. Using `0` for the *`length`* has the same effect as comparing to `NULL`, that is using the `isnull()` method.

Return value. This method returns an integer: `0` on success, and `-1` on failure.

2.3.28.4 NdbScanFilter Constructor

Description. This is the constructor method for `NdbScanFilter`, and creates a new instance of the class.

Signature.

```
NdbScanFilter
    (
      class NdbOperation* op
    )
```

Parameters. This method takes a single parameter, a pointer to the `NdbOperation` to which the filter applies.

Return value. A new instance of `NdbScanFilter`.

Destructor. The destructor takes no arguments and does not return a value. It should be called to remove the `NdbScanFilter` object when it is no longer needed.

2.3.28.5 NdbScanFilter::end()

Description. This method completes a compound, signalling that there are no more conditions to be added to it.

Signature.

```
int end
    (
      void
    )
```

Parameters. *None*.

Return value. Returns `0` on success, or `-1` on failure.

2.3.28.6 NdbScanFilter::eq()

Description. This method is used to perform an equality test on a column value and an integer.

Signature.

```
int eq
    (
      int    ColId,
      Uint32 value
    )
```

or

```
int eq
    (
      int    ColId,
      Uint64 value
    )
```

Parameters. This method takes two parameters, listed here:

- The ID (*`ColId`*) of the column whose value is to be tested
- An integer with which to compare the column value; this integer may be either 32-bit or 64-bit, and is unsigned in either case.

Return value. Returns `0` on success, or `-1` on failure.

2.3.28.7 NdbScanFilter::isfalse()

Description. Defines a term of the current group as `FALSE`.

Signature.

```
int isfalse
    (
      void
    )
```

Parameters. *None*.

Return value. `0` on success, or `-1` on failure.

2.3.28.8 NdbScanFilter::isnotnull()

Description. This method is used to check whether a column value is not `NULL`.

Signature.

```
int isnotnull
    (
      int ColId
    )
```

Parameters. The ID of the column whose value is to be tested.

Return value. Returns `0`, if the column value is not `NULL`.

2.3.28.9 NdbScanFilter::isnull()

Description. This method is used to check whether a column value is `NULL`.

Signature.

```
int isnull
    (
      int ColId
    )
```

Parameters. The ID of the column whose value is to be tested.

Return value. Returns `0`, if the column value is `NULL`.

2.3.28.10 NdbScanFilter::istrue()

Description. Defines a term of the current group as `TRUE`.

Signature.

```
int istrue
    (
      void
    )
```

Parameters. *None*.

Return value. Returns `0` on success, `-1` on failure.

2.3.28.11 NdbScanFilter::ge()

Description. This method is used to perform a greater-than-or-equal test on a column value and an integer.

Signature. This method accepts both 32-bit and 64-bit values, as shown here:

```
int ge
    (
      int    ColId,
```

```
      Uint32 value
    )


int ge
    (
      int    ColId,
      Uint64 value
    )
```

Parameters. Like `eq()`, `lt()`, `le()`, and the other `NdbScanFilter` methods of this type, this method takes two parameters:

- The ID (*`ColId`*) of the column whose value is to be tested
- An integer with which to compare the column value; this integer may be either 32-bit or 64-bit, and is unsigned in either case.

Return value. `0` on success; `-1` on failure.

2.3.28.12 NdbScanFilter::getNdbError()

Description. Because errors encountered when building an `NdbScanFilter` do not propagate to any involved `NdbOperation` object, it is necessary to use this method to access error information.

Signature.

```
const NdbError& getNdbError
    (
      void
    )
```

Parameters. *None*.

Return value. A reference to an `NdbError`.

2.3.28.13 NdbScanFilter::getNdbOperation()

Description. If the `NdbScanFilter` was constructed with an `NdbOperation`, this method can be used to obtain a pointer to that `NdbOperation` object.

Signature.

```
NdbOperation* getNdbOperation
    (
      void
    )
```

Parameters. *None*.

Return value. A pointer to the `NdbOperation` associated with this `NdbScanFilter`, if there is one. Otherwise, `NULL`.

2.3.28.14 NdbScanFilter::Group

Description. This type is used to describe logical (grouping) operators, and is used with the `begin()` method. (See Section 2.3.28.1, “NdbScanFilter::begin()”.)

Enumeration values. Possible values are shown, along with descriptions, in the following table:

Value	Description
`AND`	Logical `AND`: *`A AND B AND C`*
`OR`	Logical `OR`: *`A OR B OR C`*
`NAND`	Logical `NOT AND`: `NOT (`*`A AND B AND C`*`)`

Value	Description
NOR	Logical NOT OR: NOT (*A* OR *B* OR *C*)

2.3.28.15 NdbScanFilter::gt()

Description. This method is used to perform a greater-than (strict upper bound) test on a column value and an integer.

Signature. This method accommodates both 32-bit and 64-bit values:

```
int gt
    (
      int    ColId,
      Uint32 value
    )


int gt
    (
      int    ColId,
      Uint64 value
    )
```

Parameters. Like the other `NdbScanFilter` methods of this type, this method takes two parameters:

- The ID (*`ColId`*) of the column whose value is to be tested
- An integer with which to compare the column value; this integer may be either 32-bit or 64-bit, and is unsigned in either case.

Return value. `0` on success; `-1` on failure.

2.3.28.16 NdbScanFilter::le()

Description. This method is used to perform a less-than-or-equal test on a column value and an integer.

Signature. This method has two variants, to accommodate 32-bit and 64-bit values:

```
int le
    (
      int    ColId,
      Uint32 value
    )


int le
    (
      int    ColId,
      Uint64 value
    )
```

Parameters. Like the other `NdbScanFilter` methods of this type, this method takes two parameters:

- The ID (*`ColId`*) of the column whose value is to be tested
- An integer with which to compare the column value; this integer may be either 32-bit or 64-bit, and is unsigned in either case.

Return value. Returns `0` on success, or `-1` on failure.

2.3.28.17 NdbScanFilter::lt()

Description. This method is used to perform a less-than (strict lower bound) test on a column value and an integer.

Signature. This method has 32-bit and 64-bit variants, as shown here:

```
int lt
    (
      int    ColId,
      Uint32 value
    )

int lt
    (
      int    ColId,
      Uint64 value
    )
```

Parameters. Like `eq()`, `ne()`, and the other `NdbScanFilter` methods of this type, this method takes two parameters, listed here:

- The ID (*`ColId`*) of the column whose value is to be tested
- An integer with which to compare the column value; this integer may be either 32-bit or 64-bit, and is unsigned in either case.

Return value. Retrturns `0` on success, or `-1` on failure.

2.3.28.18 NdbScanFilter::ne()

Description. This method is used to perform an inequality test on a column value and an integer.

Signature. This method has 32-bit and 64-bit variants, as shown here:

```
int ne
    (
      int    ColId,
      Uint32 value
    )

int ne
    (
      int    ColId,
      Uint64 value
    )
```

Parameters. Like `eq()` and the other `NdbScanFilter` methods of this type, this method takes two parameters:

- The ID (*`ColId`*) of the column whose value is to be tested
- An integer with which to compare the column value; this integer may be either 32-bit or 64-bit, and is unsigned in either case.

Return value. Returns `0` on success, or `-1` on failure.

2.3.29 The NdbScanOperation Class

This section describes the `NdbScanOperation` class and its class members.

Parent class. `NdbOperation`

Child classes. `NdbIndexScanOperation`

Description. The `NdbScanOperation` class represents a scanning operation used in a transaction. This class inherits from `NdbOperation`.

Methods. The following table lists the public methods of this class and the purpose or use of each method:

Method	Purpose / Use
`close()`	Closes the scan
`deleteCurrentTuple()`	Deletes the current tuple
`lockCurrentTuple()`	Locks the current tuple
`nextResult()`	Gets the next tuple
`getNdbTransaction()`	Gets the `NdbTransaction` object for this scan
`getPruned()`	Used to find out whether this scan is pruned to a single partition
`readTuples()`	Reads tuples
`restart()`	Restarts the scan
`updateCurrentTuple()`	Updates the current tuple

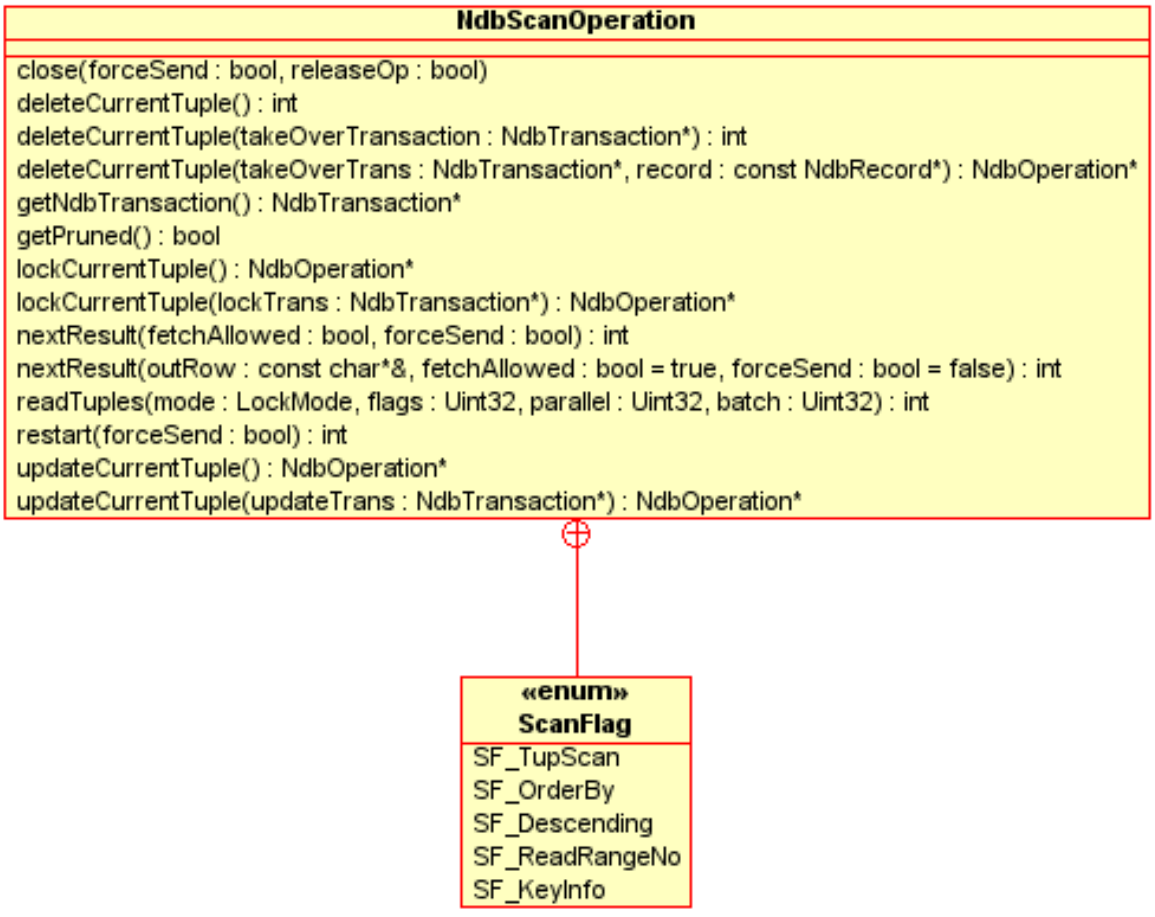
Note

This class has no public constructor. To create an instance of `NdbScanOperation`, it is necessary to use the `NdbTransaction::getNdbScanOperation()` method.

Types. This class defines a single public type `ScanFlag`.

Class diagram. This diagram shows all the available members of the `NdbScanOperation` class:

Figure 2.24 NdbScanOperation



For more information about the use of `NdbScanOperation`, see Scan Operations, and Using Scans to Update or Delete Rows.

2.3.29.1 NdbScanOperation::close()

Description. Calling this method closes a scan. Rows returned by this scan are no longer available after the scan has been closed using this method.

Note

See Scans with exclusive locks, for information about multiple threads attempting to perform the same scan with an exclusive lock and how this can affect closing the scans.

Signature.

```
void close
    (
      bool forceSend = false,
      bool releaseOp = false
    )
```

Parameters. This method takes the two parameters listed here:

- *`forceSend`* defaults to `false`; call `close()` with this parameter set to `true` in order to force transactions to be sent.
- *`releaseOp`* also defaults to `false`; set this to `true` in order to release the operation.

  Prior to NDB 7.2.19 and NDB 7.3.8, the buffer allocated by an `NdbScanOperation` for receiving the scanned rows was not released until the `NdbTransaction` owning the scan operation was closed (Bug #75128, Bug #20166585). In these and subsequent versions of NDB Cluster, the buffer is released whenever the cursor navigating the result set is closed using the `close()` method, regardless of the value of the *`releaseOp`* argument.

Return value. *None*.

2.3.29.2 NdbScanOperation::deleteCurrentTuple()

Description. This method is used to delete the current tuple.

Signature.

```
const NdbOperation* deleteCurrentTuple
    (
      NdbTransaction* takeOverTrans,
      const NdbRecord* record,
      char* row = 0,
      const unsigned char* mask = 0,
      const NdbOperation::OperationOptions* opts = 0,
      Uint32 sizeOfOpts = 0
    )
```

For more information, see Section 2.3.27, “The NdbRecord Interface”.

Parameters. When used with the `NdbRecord` interface, this method takes the parameters listed here:

- The transaction (*`takeOverTrans`*) that should perform the lock; when using `NdbRecord` with scans, this parameter is not optional.
- The `NdbRecord` referenced by the scan. This *`record`* value is required, even if no records are being read.

- The *row* from which to read. Set this to NULL if no read is to occur.
- The *mask* pointer is optional. If it is present, then only columns for which the corresponding bit in the mask is set are retrieved by the scan.
- OperationOptions (*opts*) can be used to provide more finely-grained control of operation definitions. An OperationOptions structure is passed with flags indicating which operation definition options are present. Not all operation types support all operation options; the options supported for each type of operation are shown in the following table:

Operation type (Method)	OperationOptions Flags Supported
readTuple()	OO_ABORTOPTION, OO_GETVALUE, OO_PARTITION_ID, OO_INTERPRETED
insertTuple()	OO_ABORTOPTION, OO_SETVALUE, OO_PARTITION_ID, OO_ANYVALUE
updateTuple()	OO_ABORTOPTION, OO_SETVALUE, OO_PARTITION_ID, OO_INTERPRETED, OO_ANYVALUE
writeTuple()	OO_ABORTOPTION, OO_SETVALUE, OO_PARTITION_ID, OO_ANYVALUE
deleteTuple()	OO_ABORTOPTION, OO_GETVALUE, OO_PARTITION_ID, OO_INTERPRETED, OO_ANYVALUE

- The optional *sizeOfOptions* parameter is used to preserve backward compatibility of this interface with previous definitions of the OperationOptions structure. If an unusual size is detected by the interface implementation, it can use this to determine how to interpret the passed OperationOptions structure. To enable this functionality, the caller should pass sizeof(NdbOperation::OperationOptions) for the value of this argument.
- If options are specified, their length (*sizeOfOpts*) must be specified as well.

Return value. Returns 0 on success, or -1 on failure.

2.3.29.3 NdbScanOperation::getNdbTransaction()

Description. Gets the NdbTransaction object for this scan.

Signature.

```
NdbTransaction* getNdbTransaction
    (
      void
    ) const
```

Parameters. *None*.

Return value. A pointer to an NdbTransaction object.

2.3.29.4 NdbScanOperation::getPruned()

Description. This method is used to determine whether or not a given scan operation has been pruned to a single partition. For scans defined using NdbRecord, this method can be called before or after the scan is executed. For scans not defined using NdbRecord, getPruned() is valid only after the scan has been executed.

Signature.

```
bool getPruned
    (
      void
    ) const
```

Parameters. *None*.

Return value. Returns `true`, if the scan is pruned to a single table partition.

2.3.29.5 NdbScanOperation::lockCurrentTuple()

Description. This method locks the current tuple.

Signature. In MySQL 5.1 and later, this method can be called with an optional single parameter, in either of the two ways shown here:

```
NdbOperation* lockCurrentTuple
    (
      void
    )

NdbOperation* lockCurrentTuple
    (
      NdbTransaction* lockTrans
    )
```

The following signature is also supported for this method, when using `NdbRecord`:

```
NdbOperation *lockCurrentTuple
    (
      NdbTransaction* takeOverTrans,
      const NdbRecord* record,
      char* row = 0,
      const unsigned char* mask = 0
    )
```

This method also supports specifying one or more `OperationOptions` (also when using `NdbRecord`):

```
NdbOperation *lockCurrentTuple
    (
      NdbTransaction* takeOverTrans,
      const NdbRecord* record,
      char* row = 0,
      const unsigned char* mask = 0,
      const NdbOperation::OperationOptions* opts = 0,
      Uint32 sizeOfOptions = 0
    )
```

Parameters (old style). This method takes a single, optional parameter—the transaction that should perform the lock. If this is omitted, the transaction is the current one.

Parameters (when using NdbRecord). When using the `NdbRecord` interface, this method takes these parameters, as described in the following list:

- The transaction (*`takeOverTrans`*) that should perform the lock; when using `NdbRecord` with scans, this parameter is not optional.
- The `NdbRecord` referenced by the scan. This is required, even if no records are being read.
- The *`row`* from which to read. Set this to `NULL` if no read is to occur.
- The *`mask`* pointer is optional. If it is present, then only columns for which the corresponding bit in the mask is set are retrieved by the scan.
- The *`opts`* argument can take on any of the following `OperationOptions` values: `OO_ABORTOPTION`, `OO_GETVALUE`, and `OO_ANYVALUE`.
- If options are specified, their length (*`sizeOfOptions`*) must be specified as well.

Important

Calling an `NdbRecord` scan lock takeover on an `NdbRecAttr`-style scan is not valid, nor is calling an `NdbRecAttr`-style scan lock takeover on an `NdbRecord`-style scan.

Return value. This method returns a pointer to an `NdbOperation` object, or `NULL`.

2.3.29.6 NdbScanOperation::nextResult()

Description. This method is used to fetch the next tuple in a scan transaction. Following each call to `nextResult()`, the buffers and `NdbRecAttr` objects defined in `NdbOperation::getValue()` are updated with values from the scanned tuple.

Prior to NDB 7.2.7: When an NDB API application called this method again after the previous call had returned end-of-file (return code 1), a transaction object was leaked, and NDB returned -1 (undefined). (Bug #11748194) *Later versions*: When `nextResult()` is executed following end-of-file, `NDB` returns error code 4210 (`Ndb sent more info than length specified`) and the extra transaction object is freed by returning it to the idle list for the right TC node.

Signature. This method can be invoked in one of two ways. The first of these, shown here, is available beginning in MySQL 5.1:

```
int nextResult
    (
      bool fetchAllowed = true,
      bool forceSend = false
    )
```

It is also possible to use this method as shown here:

```
int nextResult
    (
      const char*& outRow,
      bool fetchAllowed = true,
      bool forceSend = false
    )
```

Parameters (2-parameter version). This method takes the following two parameters:

- Normally, the NDB API contacts the `NDB` kernel for more tuples whenever it is necessary; setting *`fetchAllowed`* to `false` keeps this from happening.

  Disabling *`fetchAllowed`* by setting it to `false` forces `NDB` to process any records it already has in its caches. When there are no more cached records it returns `2`. You must then call `nextResult()` with *`fetchAllowed`* equal to `true` in order to contact `NDB` for more records.

  While `nextResult(false)` returns `0`, you should transfer the record to another transaction using `execute(NdbTransaction::NoCommit)`. When `nextResult(false)` returns `2`, you should normally execute and commit the other transaction. This causes any locks to be transferred to the other transaction, updates or deletes to be made, and then, the locks to be released. Following this, you can call `nextResult(true)` to have more records fetched and cached in the NDB API.

  Note

  If you do not transfer the records to another transaction, the locks on those records will be released the next time that the `NDB` Kernel is contacted for more records.

  Disabling *`fetchAllowed`* can be useful when you want to update or delete all of the records obtained in a given transaction, as doing so saves time and speeds up updates or deletes of scanned records.

- *forceSend* defaults to false, and can normally be omitted. However, setting this parameter to true means that transactions are sent immediately. See Section 1.3.4, “The Adaptive Send Algorithm”, for more information.

Parameters (3-parameter version). This method can also be called with the following three parameters:

- Calling nextResult() sets a pointer to the next row in *outRow* (if returning 0). This pointer is valid (only) until the next call to nextResult() when *fetchAllowed* is true. The NdbRecord object defining the row format must be specified beforehand using NdbTransaction::scanTable() (or NdbTransaction::scanIndex().
- When false, *fetchAllowed* forces NDB to process any records it already has in its caches. See the description for this parameter in the previous *Parameters* subsection for more details.
- Setting *forceSend* to true means that transactions are sent immediately, as described in the previous *Parameters* subsection, as well as in Section 1.3.4, “The Adaptive Send Algorithm”.

Return value. This method returns one of the following 4 integer values, interpreted as shown in the following list:

- -1: Indicates that an error has occurred.
- 0: Another tuple has been received.
- 1: There are no more tuples to scan.
- 2: There are no more cached records (invoke nextResult(true) to fetch more records).

Example. See Section 2.5.4, “NDB API Basic Scanning Example”.

2.3.29.7 NdbScanOperation::readTuples()

Description. This method is used to perform a scan.

Signature.

```
virtual int readTuples
    (
      LockMode mode = LM_Read,
      Uint32   flags = 0,
      Uint32   parallel = 0,
      Uint32   batch = 0
    )
```

Parameters. This method takes the four parameters listed here:

- The lock *mode*; this is a LockMode value.

  Scans with exclusive locks. When scanning with an exclusive lock, extra care must be taken due to the fact that, if two threads perform this scan simultaneously over the same range, then there is a significant probability of causing a deadlock. The likelihood of a deadlock is increased if the scan is also ordered (that is, using SF_OrderBy or SF_Descending).

  The NdbScanOperation::close() method is also affected by this deadlock, since all outstanding requests are serviced before the scan is actually closed.

- One or more ScanFlag values. Multiple values are OR'ed together
- The number of fragments to scan in *parallel*; use 0 to require that the maximum possible number be used.

- The *batch* parameter specifies how many records will be returned to the client from the server by the next `NdbScanOperation::nextResult(true)` method call. Use `0` to specify the maximum automatically.

Note

This parameter was ignored prior to MySQL 5.1.12, and the maximum was used (see Bug #20252).

Return value. Returns `0` on success, `-1` on failure.

2.3.29.8 NdbScanOperation::restart()

Description. Use this method to restart a scan without changing any of its `getValue()` calls or search conditions.

Signature.

```
int restart
    (
      bool forceSend = false
    )
```

Parameters. Call this method with *forceSend* set to `true` in order to force the transaction to be sent.

Return value. `0` on success; `-1` on failure.

2.3.29.9 NdbScanOperation::ScanFlag

Description. Values of this type are the scan flags used with the `readTuples()` method. More than one may be used, in which case, they are `OR`'ed together as the second argument to that method. See Section 2.3.29.7, "NdbScanOperation::readTuples()", for more information.

Enumeration values. Possible values are shown, along with descriptions, in the following table:

Value	Description
`SF_TupScan`	Scan in TUP order (that is, in the order of the rows in memory). Applies to table scans only.
`SF_DiskScan`	Scan in disk order (order of rows on disk). Applies to table scans only.
`SF_OrderBy`	Ordered index scan (ascending); rows returned from an index scan are sorted, and ordered on the index key. Scans in either ascending or descending order are affected by this flag, which causes the API to perform a merge-sort among the ordered scans of each fragment to obtain a single sorted result set. *Notes*: • Ordered indexes are distributed, with one ordered index for each fragment of a table. • Range scans are often parallel across all index fragments. Occasionally, they can be pruned to one index fragment. • Each index fragment range scan can return results in either ascending or descending order. Ascending is the default; to choose descending order, set the `SF_Descending` flag.

Value	Description
	• When multiple index fragments are scanned in parallel, the results are sent back to NDB where they can optionally be merge-sorted before being returned to the user. This merge sorting is controlled using the `SF_OrderBy` and `SF_OrderByFull` flags. • If `SF_OrderBy` or `SF_OrderByFull` is not used, the results from each index fragment are in order (either ascending or descending), but results from different fragments may be interleaved. • When using `SF_OrderBy` or `SF_OrderByFull`, some extra constraints are imposed internally; these are listed here: 1. If the range scan is not pruned to one index fragment then all index fragments must be scanned in parallel. (Unordered scans can be executed with less than full parallelism.) 2. Results from every index fragment must be available before returning any rows, to ensure a correct merge sort. This serialises the “scrolling” of the scan, potentially resulting in lower row throughput. 3. Unordered scans can return rows to the API client before all index fragments have returned any batches, and can overlap next-batch requests with row processing.
`SF_OrderByFull`	This is the same as `SF_OrderBy`, except that all key columns are added automatically to the read bitmask.
`SF_Descending`	Causes an ordered index scan to be performed in descending order.
`SF_ReadRangeNo`	For index scans, when this flag is set, `NdbIndexScanOperation::get_range_no()` can be called to read back the `range_no` defined in `NdbIndexScanOperation::setBound()`. In addition, when this flag is set, and `SF_OrderBy` or `SF_OrderByFull` is also set, results from ranges are returned in their entirety before any results are returned from subsequent ranges.
`SF_MultiRange`	Indicates that this scan is part of a multirange scan; each range is scanned separately.
`SF_KeyInfo`	Requests `KeyInfo` to be sent back to the caller. This enables the option to take over the row lock taken by the scan, using `lockCurrentTuple()`, by making sure that the kernel sends back the information needed to identify the row and the lock. This flag is enabled by default for scans using `LM_Exclusive`, but must be explicitly specified to enable the taking over of `LM_Read` locks. (See the `LockMode` documentation for more information.)

2.3.29.10 NdbScanOperation::updateCurrentTuple()

Description. This method is used to update the current tuple.

Signature. Originally, this method could be called with a single. optional parameter, in either of the ways shown here:

```
NdbOperation* updateCurrentTuple
    (
```

```
      void
    )


NdbOperation* updateCurrentTuple
    (
      NdbTransaction* updateTrans
    )
```

It is also possible to employ this method, when using `NdbRecord` with scans, as shown here:

```
NdbOperation* updateCurrentTuple
    (
      NdbTransaction*      takeOverTrans,
      const NdbRecord*     record,
      const char*          row,
      const unsigned char* mask = 0
    )
```

See Section 2.3.27, “The NdbRecord Interface”, for more information.

Parameters (original). This method takes a single, optional parameter—the transaction that should perform the lock. If this is omitted, the transaction is the current one.

Parameters (when using NdbRecord). When using the `NdbRecord` interface, this method takes the following parameters, as described in the following list:

- The takeover transaction (*`takeOverTrans`*).
- The *`record`* (`NdbRecord` object) referencing the column used for the scan.
- The *`row`* to read from. If no attributes are to be read, set this equal to `NULL`.
- The *`mask`* pointer is optional. If it is present, then only columns for which the corresponding bit in the mask is set are retrieved by the scan.

Return value. This method returns an `NdbOperation` object or `NULL`.

2.3.30 The NdbTransaction Class

This section describes the `NdbTransaction` class and its public members.

Parent class. *None*

Child classes. *None*

Description. A transaction is represented in the NDB API by an `NdbTransaction` object, which belongs to an `Ndb` object and is created using `Ndb::startTransaction()`. A transaction consists of a list of operations represented by the `NdbOperation` class, or by one of its subclasses —`NdbScanOperation`, `NdbIndexOperation`, or `NdbIndexScanOperation`. Each operation access exactly one table.

Using Transactions. After obtaining an `NdbTransaction` object, it is employed as follows:

1. An operation is allocated to the transaction using any one of the following methods:
   - `getNdbOperation()`
   - `getNdbScanOperation()`
   - `getNdbIndexOperation()`
   - `getNdbIndexScanOperation()`

Calling one of these methods defines the operation. Several operations can be defined on the same `NdbTransaction` object, in which case they are executed in parallel. When all operations are defined, the `execute()` method sends them to the `NDB` kernel for execution.

2. The `execute()` method returns when the `NDB` kernel has completed execution of all operations previously defined.

Important

All allocated operations should be properly defined *before* calling the `execute()` method.

3. `execute()` operates in one of the three modes listed here:

   - `NdbTransaction::NoCommit`: Executes operations without committing them.
   - `NdbTransaction::Commit`: Executes any remaining operation and then commits the complete transaction.
   - `NdbTransaction::Rollback`: Rolls back the entire transaction.

   `execute()` is also equipped with an extra error handling parameter, which provides the two alternatives listed here:

   - `NdbOperation::AbortOnError`: Any error causes the transaction to be aborted. This is the default behavior.
   - `NdbOperation::AO_IgnoreError`: The transaction continues to be executed even if one or more of the operations defined for that transaction fails.

Note

In MySQL 5.1.15 and earlier, these values were `NdbTransaction::AbortOnError` and `NdbTransaction::AO_IgnoreError`.

Methods. The following table lists the public methods of this class and the purpose or use of each method:

Method	Purpose / Use
`close()`	Closes a transaction
`commitStatus()`	Gets the transaction's commit status
`deleteTuple()`	Delete a tuple using `NdbRecord`
`execute()`	Executes a transaction
`executePendingBlobOps()`	Executes a transaction in `NoCommit` mode if it includes any blob part operations of the specified types that are not yet executed.
`getGCI()`	Gets a transaction's global checkpoint ID (GCI)
`getMaxPendingBlobReadBytes()`	Get the current `BLOB` read batch size
`getMaxPendingBlobWriteBytes()`	Get the current `BLOB` write batch size
`getNdbError()`	Gets the most recent error
`getNdbErrorLine()`	Gets the line number where the most recent error occurred
`getNdbErrorOperation()`	Gets the most recent operation which caused an error
`getNextCompletedOperation()`	Gets operations that have been executed; used for finding errors

Method	Purpose / Use
getNdbOperation()	Gets an NdbOperation
getNdbScanOperation()	Gets an NdbScanOperation
getNdbIndexOperation()	Gets an NdbIndexOperation
getNdbIndexScanOperation()	Gets an NdbIndexScanOperation
getTransactionId()	Gets the transaction ID
insertTuple()	Insert a tuple using NdbRecord
readTuple()	Read a tuple using NdbRecord
refresh()	Keeps a transaction from timing out
releaseLockHandle()	Release an NdbLockHandle object once it is no longer needed
scanIndex()	Perform an index scan using NdbRecord
scanTable()	Perform a table scan using NdbRecord
setMaxPendingBlobReadBytes()	Set the BLOB read batch size
setMaxPendingBlobWriteBytes()	Set the BLOB write batch size
setSchemaObjectOwnerChecks()	Enable or disable schema object ownership checks
unlock()	Create an unlock operation on the current transaction
updateTuple()	Update a tuple using NdbRecord
writeTuple()	Write a tuple using NdbRecord

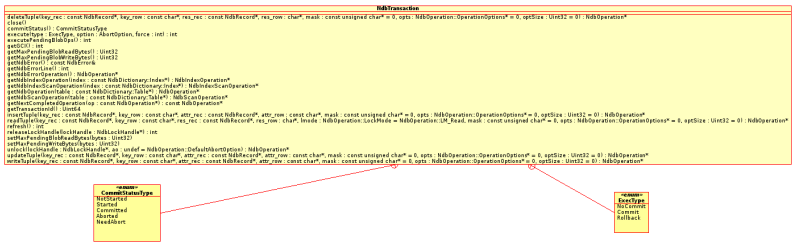
The methods readTuple(), insertTuple(), updateTuple(), writeTuple(), deleteTuple(), scanTable(), and scanIndex() require the use of NdbRecord.

Types. NdbTransaction defines 2 public types as shown in the following table:

Type	Purpose / Use
CommitStatusType()	Describes the transaction's commit status
ExecType()	Determines whether the transaction should be committed or rolled back

Class diagram. This diagram shows all the available methods and enumerated types of the NdbTransaction class:

Figure 2.25 NdbTransaction



2.3.30.1 NdbTransaction::close()

Description. This method closes a transaction. It is equivalent to calling Ndb::closeTransaction().

Important

If the transaction has not yet been committed, it is aborted when this method is called. See Section 2.3.16.34, “Ndb::startTransaction()”.

Signature.

```
void close
    (
      void
    )
```

Parameters. *None*.

Return value. *None*.

2.3.30.2 NdbTransaction::commitStatus()

Description. This method gets the transaction's commit status.

Signature.

```
CommitStatusType commitStatus
    (
      void
    )
```

Parameters. *None*.

Return value. The commit status of the transaction, a `CommitStatusType` value. See Section 2.3.30.3, “NdbTransaction::CommitStatusType”.

2.3.30.3 NdbTransaction::CommitStatusType

Description. This type is used to describe a transaction's commit status.

Enumeration values. Possible values are shown, along with descriptions, in the following table:

Value	Description
`NotStarted`	The transaction has not yet been started.
`Started`	The transaction has started, but is not yet committed.
`Committed`	The transaction has completed, and has been committed.
`Aborted`	The transaction was aborted.
`NeedAbort`	The transaction has encountered an error, but has not yet been aborted.

A transaction's commit status ca be read using the `commitStatus()` method. See Section 2.3.30.2, “NdbTransaction::commitStatus()”.

2.3.30.4 NdbTransaction::deleteTuple()

Description. Deletes a tuple using `NdbRecord`.

Signature.

```
const NdbOperation* deleteTuple
    (
      const NdbRecord* key_rec,
      const char* key_row,
      const NdbRecord* result_rec,
      char* result_row,
      const unsigned char* result_mask = 0,
      const NdbOperation::OperationOptions* opts = 0,
      Uint32 sizeOfOptions = 0
    )
```

Parameters. This method takes the following parameters:

- *key_rec* is a pointer to an NdbRecord for either a table or an index. If on a table, then the delete operation uses a primary key; if on an index, then the operation uses a unique key. In either case, the *key_rec* must include all columns of the key.
- The *key_row* passed to this method defines the primary or unique key of the tuple to be deleted, and must remain valid until execute() is called.
- The *result_rec* is the NdbRecord to be used.
- The *result_row* can be NULL if no attributes are to be returned.
- The *result_mask*, if not NULL, defines a subset of attributes to be read and returned to the client. The mask is copied, and so does not need to remain valid after the call to this method returns.
- OperationOptions (*opts*) can be used to provide more finely-grained control of operation definitions. An OperationOptions structure is passed with flags indicating which operation definition options are present. Not all operation types support all operation options; for the options supported by each type of operation, see Section 2.3.30.21, "NdbTransaction::readTuple()".
- The optional *sizeOfOptions* parameter provides backward compatibility of this interface with previous definitions of the OperationOptions structure. If an unusual size is detected by the interface implementation, it can use this to determine how to interpret the passed OperationOptions structure. To enable this functionality, the caller should pass sizeof(NdbOperation::OperationOptions) for the value of this argument.

Return value. A const pointer to the NdbOperation representing this write operation. The operation can be checked for errors if necessary.

2.3.30.5 NdbTransaction::ExecType

Description. This type sets the transaction's execution type; that is, whether it should execute, execute and commit, or abort. It is used as a parameter to the execute() method. (See Section 2.3.30.6, "NdbTransaction::execute()".)

Enumeration values. Possible values are shown, along with descriptions, in the following table:

Value	Description
NoCommit	The transaction should execute, but not commit.
Commit	The transaction should execute and be committed.
Rollback	The transaction should be rolled back.

2.3.30.6 NdbTransaction::execute()

Description. This method is used to execute a transaction.

Signature.

```
int execute
    (
      ExecType execType,
      NdbOperation::AbortOption abortOption = NdbOperation::DefaultAbortOption,
      int force = 0
    )
```

Parameters. The execute method takes the three parameters listed here:

- The execution type (ExecType value); see Section 2.3.30.5, "NdbTransaction::ExecType", for more information and possible values.
- An abort option (NdbOperation::AbortOption value).

Errors arising from this method are found with `NdbOperation::getNdbError()` rather than `NdbTransaction::getNdbError()`. information.

- A *`force`* parameter, which determines when operations should be sent to the `NDB` Kernel. It takes ones of the values listed here:
  - `0`: Nonforced; detected by the adaptive send algorithm.
  - `1`: Forced; detected by the adaptive send algorithm.
  - `2`: Nonforced; not detected by the adaptive send algorithm.

  See Section 1.3.4, "The Adaptive Send Algorithm", for more information.

Return value. Returns `0` on success, or `-1` on failure. The fact that the transaction did not abort does not necessarily mean that each operation was successful; you must check each operation individually for errors.

In MySQL 5.1.15 and earlier versions, this method returned `-1` for some errors even when the trasnsaction itself was not aborted; beginning with MySQL 5.1.16, this method reports a failure *if and only if* the transaction was aborted. (This change was made due to the fact it had been possible to construct cases where there was no way to determine whether or not a transaction was actually aborted.) However, the transaction's error information is still set in such cases to reflect the actual error code and category.

This means, in the case where a **`NoDataFound`** error is a possibility, you must now check for it explicitly, as shown in this example:

```
Ndb_cluster_connection myConnection;

if( myConnection.connect(4, 5, 1) )
{
  cout << "Unable to connect to cluster within 30 secs." << endl;
  exit(-1);
}

Ndb myNdb(&myConnection, "test");

//  define operations...

myTransaction = myNdb->startTransaction();

if(myTransaction->getNdbError().classification == NdbError:NoDataFound)
{
  cout << "No records found." << endl;
  //  ...
}

myNdb->closeTransaction(myTransaction);
```

2.3.30.7 NdbTransaction::executePendingBlobOps()

Description. This method executes the transaction with `ExecType` equal to `NoCommit` if there remain any blob part operations of the given types which have not yet been executed.

Signature.

```
int executePendingBlobOps
  (
    Uint8 flags = 0xFF
  )
```

Parameters. The *`flags`* argument is the result of a bitwise `OR`, equal to `1 << optype`, where *`optype`* is an `NdbOperation::Type`. The default corresponds to `PrimaryKeyAccess`.

Return value. Returns `0` on success, or `-1` on failure. The fact that the transaction did not abort does not necessarily mean that each operation was successful; you must check each operation individually for errors.

2.3.30.8 NdbTransaction::getGCI()

Description. This method retrieves the transaction's global checkpoint ID (GCI).

Each committed transaction belongs to a GCI. The log for the committed transaction is saved on disk when a global checkpoint occurs.

By comparing the GCI of a transaction with the value of the latest GCI restored in a restarted NDB Cluster, you can determine whether or not the transaction was restored.

Note

Whether or not the global checkpoint with this GCI has been saved on disk cannot be determined by this method.

Important

The GCI for a scan transaction is undefined, since no updates are performed in scan transactions.

Signature.

```
int getGCI
    (
      void
    )
```

Parameters. *None*.

Return value. The transaction's GCI, or `-1` if none is available.

Note

No GCI is available until `execute()` has been called with `ExecType::Commit`.

2.3.30.9 NdbTransaction::getMaxPendingBlobReadBytes()

Description. Gets the current batch size in bytes for `BLOB` read operations. When the volume of `BLOB` data to be read within a given transaction exceeds this amount, all of the transaction's pending `BLOB` read operations are executed.

Signature.

```
Uint32 getMaxPendingBlobReadBytes
    (
      void
    ) const
```

Parameters. *None*.

Return value. The current `BLOB` read batch size, in bytes. See Section 2.3.30.26, "NdbTransaction::setMaxPendingBlobReadBytes()", for more information.

2.3.30.10 NdbTransaction::getMaxPendingBlobWriteBytes()

Description. Gets the current batch size in bytes for `BLOB` write operations. When the volume of `BLOB` data to be written within a given transaction exceeds this amount, all of the transaction's pending `BLOB` write operations are executed.

Signature.

```
Uint32 getMaxPendingBlobWriteBytes
    (
      void
    ) const
```

Parameters. *None*.

Return value. The current `BLOB` write batch size, in bytes. See Section 2.3.30.27, "NdbTransaction::setMaxPendingBlobWriteBytes()", for more information.

2.3.30.11 NdbTransaction::getNdbError()

Description. This method is used to obtain the most recent error (`NdbError`).

Signature.

```
const NdbError& getNdbError
    (
      void
    ) const
```

Parameters. *None*.

Return value. A reference to an `NdbError` object.

Note

For additional information about handling errors in transactions, see Error Handling.

2.3.30.12 NdbTransaction::getNdbErrorLine()

Description. This method return the line number where the most recent error occurred.

Signature.

```
int getNdbErrorLine
    (
      void
    )
```

Parameters. *None*.

Return value. The line number of the most recent error.

Note

For additional information about handling errors in transactions, see Error Handling.

2.3.30.13 NdbTransaction::getNdbErrorOperation()

Description. This method retrieves the operation that caused an error.

Tip

To obtain more information about the actual error, use the `NdbOperation::getNdbError()` method of the `NdbOperation` object returned by `getNdbErrorOperation()`.

Signature.

```
NdbOperation* getNdbErrorOperation
    (
      void
    )
```

Parameters. *None*.

Return value. A pointer to an `NdbOperation`.

Note

For additional information about handling errors in transactions, see Error Handling.

2.3.30.14 NdbTransaction::getNdbIndexOperation()

Description. This method is used to create an `NdbIndexOperation` associated with a given table.

Note

All index operations within the same transaction must be initialised with this method. Operations must be defined before they are executed.

Signature.

```
NdbIndexOperation* getNdbIndexOperation
    (
      const NdbDictionary::Index* index
    )
```

Parameters. The `Index` object on which the operation is to be performed.

Return value. A pointer to the new `NdbIndexOperation`.

2.3.30.15 NdbTransaction::getNdbIndexScanOperation()

Description. This method is used to create an `NdbIndexScanOperation` associated with a given table.

Note

All index scan operations within the same transaction must be initialised with this method. Operations must be defined before they are executed.

Signature.

```
NdbIndexScanOperation* getNdbIndexScanOperation
    (
      const NdbDictionary::Index* index
    )
```

Parameters. The `Index` object on which the operation is to be performed.

Return value. A pointer to the new `NdbIndexScanOperation`.

2.3.30.16 NdbTransaction::getNdbOperation()

Description. This method is used to create an `NdbOperation` associated with a given table.

Note

All operations within the same transaction must be initialised with this method. Operations must be defined before they are executed.

Signature.

```
NdbOperation* getNdbOperation
    (
      const NdbDictionary::Table* table
    )
```

Parameters. The `Table` object on which the operation is to be performed.

Return value. A pointer to the new `NdbOperation`.

2.3.30.17 NdbTransaction::getNdbScanOperation()

Description. This method is used to create an `NdbScanOperation` associated with a given table.

Note

All scan operations within the same transaction must be initialised with this method. Operations must be defined before they are executed.

Signature.

```
NdbScanOperation* getNdbScanOperation
    (
      const NdbDictionary::Table* table
    )
```

Parameters. The `Table` object on which the operation is to be performed.

Return value. A pointer to the new `NdbScanOperation`.

2.3.30.18 NdbTransaction::getNextCompletedOperation()

Description. This method is used to retrieve a transaction's completed operations. It is typically used to fetch all operations belonging to a given transaction to check for errors.

`NdbTransaction::getNextCompletedOperation(NULL)` returns the transaction's first `NdbOperation` object; `NdbTransaction::getNextCompletedOperation(myOp)` returns the `NdbOperation` object defined after `NdbOperation` *`myOp`*.

Important

This method should only be used after the transaction has been executed, but before the transaction has been closed.

Signature.

```
const NdbOperation* getNextCompletedOperation
    (
      const NdbOperation* op
    ) const
```

Parameters. This method requires a single parameter *`op`*, which is an operation (`NdbOperation` object), or `NULL`.

Return value. The operation following *`op`*, or the first operation defined for the transaction if `getNextCompletedOperation()` was called using `NULL`.

2.3.30.19 NdbTransaction::getTransactionId()

Description. This method is used to obtain the transaction ID.

Signature.

```
Uint64 getTransactionId
    (
      void
    )
```

Parameters. *None*.

Return value. The transaction ID, as an unsigned 64-bit integer.

2.3.30.20 NdbTransaction::insertTuple()

Description. Inserts a tuple using `NdbRecord`.

Signature.

```
const NdbOperation* insertTuple
    (
      const NdbRecord* key_rec,
      const char* key_row,
      const NdbRecord* attr_rec,
      const char* attr_row,
      const unsigned char* mask = 0,
      const NdbOperation::OperationOptions* opts = 0,
      Uint32 sizeOfOptions = 0
    )
```

```
const NdbOperation* insertTuple
    (
      const NdbRecord* combined_rec,
      const char* combined_row,
      const unsigned char* mask = 0,
      const NdbOperation::OperationOptions* opts = 0,
      Uint32 sizeOfOptions = 0
    )
```

Parameters. `insertTuple()` takes the following parameters:

- A pointer to an `NdbRecord` indicating the record (*`key_rec`*) to be inserted.
- A row (*`key_row`*) of data to be inserted.
- A pointer to an `NdbRecord` indicating an attribute (*`attr_rec`*) to be inserted.
- A row (*`attr_row`*) of data to be inserted as the attribute.
- A *`mask`* which can be used to filter the columns to be inserted.
- `OperationOptions` (*`opts`*) can be used to provide more finely-grained control of operation definitions. An `OperationOptions` structure is passed with flags indicating which operation definition options are present. Not all operation types support all operation options; for the options supported by each type of operation, see Section 2.3.30.21, “NdbTransaction::readTuple()”.
- The optional *`sizeOfOptions`* parameter is used to preserve backward compatibility of this interface with previous definitions of the `OperationOptions` structure. If an unusual size is detected by the interface implementation, it can use this to determine how to interpret the passed `OperationOptions` structure. To enable this functionality, the caller should pass `sizeof(NdbOperation::OperationOptions)` for the value of this argument.

This method can also be called using a single `NdbRecord` pointer and single `char` pointer (*`combined_rec`*, *`combined_row`*) where the single `NdbRecord` represents record and attribute and data.

Return value. A `const` pointer to the `NdbOperation` representing this insert operation.

2.3.30.21 NdbTransaction::readTuple()

Description. This method reads a tuple using `NdbRecord` objects.

Signature.

```
const NdbOperation* readTuple
    (
      const NdbRecord* key_rec,
      const char* key_row,
      const NdbRecord* result_rec,
      char* result_row,
      NdbOperation::LockMode lock_mode = NdbOperation::LM_Read,
      const unsigned char* result_mask = 0,
      const NdbOperation::OperationOptions* opts = 0,
      Uint32 sizeOfOptions = 0
    )
```

Parameters. This method takes the following parameters:

- *`key_rec`* is a pointer to an `NdbRecord` for either a table or an index. If on a table, then the operation uses a primary key; if on an index, then the operation uses a unique key. In either case, the *`key_rec`* must include all columns of the key.
- The *`key_row`* passed to this method defines the primary or unique key of the affected tuple, and must remain valid until `execute()` is called.

  The mask, if not `NULL`, defines a subset of attributes to read, update, or insert. Only if `(mask[`*`attrId`*` >> 3] & (1<<(`*`attrId`*` & 7)))` is set is the column affected. The mask is copied by the methods, so need not remain valid after the call returns.
- *`result_rec`* is a pointer to an `NdbRecord` used to hold the result
- *`result_row`* defines a buffer for the result data.
- *`lock_mode`* specifies the lock mode in effect for the operation. See Section 2.3.25.15, “NdbOperation::LockMode”, for permitted values and other information.
- *`result_mask`* defines a subset of attributes to read. Only if `mask[attrId >> 3] & (1<<(attrId & 7))` is set is the column affected. The mask is copied, and so need not remain valid after the method call returns.
- `OperationOptions` (*`opts`*) can be used to provide more finely-grained control of operation definitions. An `OperationOptions` structure is passed with flags indicating which operation definition options are present. Not all operation types support all operation options; the options supported for each type of operation are shown in the following table:

Operation type (Method)	`OperationOptions` Flags Supported
`readTuple()`	`OO_ABORTOPTION`, `OO_GETVALUE`, `OO_PARTITION_ID`, `OO_INTERPRETED`
`insertTuple()`	`OO_ABORTOPTION`, `OO_SETVALUE`, `OO_PARTITION_ID`, `OO_ANYVALUE`
`updateTuple()`	`OO_ABORTOPTION`, `OO_SETVALUE`, `OO_PARTITION_ID`, `OO_INTERPRETED`, `OO_ANYVALUE`
`writeTuple()`	`OO_ABORTOPTION`, `OO_SETVALUE`, `OO_PARTITION_ID`, `OO_ANYVALUE`
`deleteTuple()`	`OO_ABORTOPTION`, `OO_GETVALUE`, `OO_PARTITION_ID`, `OO_INTERPRETED`, `OO_ANYVALUE`

- The optional *`sizeOfOptions`* parameter is used to preserve backward compatibility of this interface with previous definitions of the `OperationOptions` structure. If an unusual size is detected by the interface implementation, it can use this to determine how to interpret the passed `OperationOptions` structure. To enable this functionality, the caller should pass `sizeof(NdbOperation::OperationOptions)` for the value of this argument.

Return value. A pointer to the `NdbOperation` representing this read operation (this can be used to check for errors).

2.3.30.22 NdbTransaction::refresh()

Description. This method updates the transaction's timeout counter, and thus avoids aborting due to transaction timeout.

Note

It is not advisable to take a lock on a record and maintain it for a extended time since this can impact other transactions.

Signature.

```
int refresh
    (
      void
    )
```

Parameters. *None*.

Return value. Returns `0` on success, `-1` on failure.

2.3.30.23 NdbTransaction::releaseLockHandle()

Description. This method is used to release a lock handle (see Section 2.3.25.5, "NdbOperation::getLockHandle") when it is no longer required. For `NdbRecord` primary key read operations, this cannot be called until the associated read operation has been executed.

Note

All lock handles associated with a given transaction are released when that transaction is closed.

Signature.

```
int releaseLockHandle
    (
      const NdbLockHandle* lockHandle
    )
```

Parameters. The `NdbLockHandle` object to be released.

Return value. 0 on success.

2.3.30.24 NdbTransaction::scanIndex()

Description. Perform an index range scan of a table, with optional ordering.

Signature.

```
NdbIndexScanOperation* scanIndex
    (
      const NdbRecord* key_record,
      const NdbRecord* result_record,
      NdbOperation::LockMode lock_mode = NdbOperation::LM_Read,
```

```
      const unsigned char* result_mask = 0,
      const NdbIndexScanOperation::IndexBound* bound = 0,
      const NdbScanOperation::ScanOptions* options = 0,
      Uint32 sizeOfOptions = 0
    )
```

Parameters. The *key_record* describes the index to be scanned. It must be a key record for the index; that is, it must specify, at a minimum, all of the key columns of the index. The *key_record* must be created from the index to be scanned (and not from the underlying table).

The *result_record* describes the rows to be returned from the scan. For an ordered index scan, *result_record* must be a key record for the index to be scanned; that is, it must include (at a minimum) all of the columns in the index (the full index key is needed by the NDB API for merge-sorting the ordered rows returned from each fragment).

Like the *key_record*, the result_record must be created from the underlying table, and not from the index to be scanned. Both the *key_record* and *result_record* `NdbRecord` structures must stay in place until the scan operation is closed.

A single `IndexBound` can be specified either in this call or in a separate call to `NdbIndexScanOperation::setBound()`. To perform a multi-range read, the *scan_flags* in the `ScanOptions` structure must include `SF_MULTIRANGE`. Additional bounds can be added using successive calls to `NdbIndexScanOperation::setBound()`.

To specify an equals bound, use the same row pointer for the *low_key* and *high_key* with the low and high inclusive bits set.

To specify additional options, pass a `ScanOptions` structure.

The *sizeOfOptions* exists To enable backward compatability for this interface. This parameter indicates the size of the `ScanOptions` structure at the time the client was compiled, and enables detection of the use of an old-style `ScanOptions` structure. If this functionality is not required, this argument can be left set to 0.

Note

For multi-range scans, the *low_key* and *high_key* pointers must be unique. In other words, it is not permissible to reuse the same row buffer for several different range bounds within a single scan. However, it is permissible to use the same row pointer as *low_key* and *high_key* in order to specify an equals bound; it is also permissible to reuse the rows after the `scanIndex()` method returns—that is, they need not remain valid until `execute()` time (unlike the `NdbRecord` pointers).

Return value. The current `NdbIndexScanOperation`, which can be used for error checking.

2.3.30.25 NdbTransaction::scanTable()

Description. This method performs a table scan, using an `NdbRecord` object to read out column data.

Signature.

```
NdbScanOperation* scanTable
    (
      const NdbRecord* result_record,
      NdbOperation::LockMode lock_mode = NdbOperation::LM_Read,
      const unsigned char* result_mask = 0,
      Uint32 scan_flags = 0,
      Uint32 parallel = 0,
      Uint32 batch = 0
    )
```

Parameters. The `scanTable()` method takes the following parameters:

- A pointer to an `NdbRecord` for storing the result. This *`result_record`* must remain valid until after the `execute()` call has been made.
- The *`lock_mode`* in effect for the operation. See Section 2.3.25.15, “NdbOperation::LockMode”, for permitted values and other information.
- The *`result_mask`* pointer is optional. If it is present, only columns for which the corresponding bit (by attribute ID order) in *`result_mask`* is set will be retrieved in the scan. The *`result_mask`* is copied internally, so in contrast to *`result_record`* need not be valid when `execute()` is invoked.
- *`scan_flags`* can be used to impose ordering and sorting conditions for scans. See Section 2.3.29.9, “NdbScanOperation::ScanFlag”, for a list of permitted values.
- The *`parallel`* argument is the desired parallelism, or `0` for maximum parallelism (receiving rows from all fragments in parallel), which is the default.
- *`batch`* determines whether batching is employed. The default is 0 (off).

Return value. A pointer to the `NdbScanOperation` representing this scan. The operation can be checked for errors if necessary.

2.3.30.26 NdbTransaction::setMaxPendingBlobReadBytes()

Description. Sets the batch size in bytes for `BLOB` read operations. When the volume of `BLOB` data to be read within a given transaction exceeds this amount, all of the transaction's pending `BLOB` read operations are executed.

Signature.

```
void setMaxPendingBlobReadBytes
    (
      Uint32 bytes
    )
```

Parameters. The batch size, as the number of *`bytes`*. Using 0 causes `BLOB` read batching to be disabled, which is the default behavior (for backward compatibility).

Return value. *None*.

Note

`BLOB` read batching can also be controlled in the mysql client and other MySQL client application using the MySQL Server's `--ndb-blob-read-batch-bytes` option and its associated MySQL Server system variables.

2.3.30.27 NdbTransaction::setMaxPendingBlobWriteBytes()

Description. Sets the batch size in bytes for `BLOB` write operations. When the volume of `BLOB` data to be written within a given transaction exceeds this amount, all of the transaction's pending `BLOB` write operations are executed.

Signature.

```
void setMaxPendingBlobWriteBytes
    (
      Uint32 bytes
    )
```

Parameters. The batch size, as the number of *`bytes`*. Using 0 causes `BLOB` write batching to be disabled, which is the default behavior (for backward compatibility).

Return value. *None*.

Note

`BLOB` write batching can also be controlled in the mysql client and other MySQL client application using the MySQL Server's `--ndb-blob-write-batch-bytes` option and its associated MySQL Server system variables.

2.3.30.28 NdbTransaction::setSchemaObjectOwnerChecks()

Description. Enables or disables a schema object ownership check when multiple `Ndb_cluster_connection` objects are in use. When this check is enabled, objects used by this transaction are checked to make sure that they belong to the `NdbDictionary` owned by this connection. This is done by acquiring the schema objects of the same names from the connection and comparing these with the schema objects passed to the transaction. If they do not match, an error is returned.

This method is available for debugging purposes beginning with NDB 7.2.0, NDB 7.3.9, and NDB 7.4.4. (Bug #19875977) You should be aware that enabling this check carries a performance penalty and for this reason you should avoid doing so in a production setting.

Signature.

```
void setSchemaObjOwnerChecks
    (
      bool runChecks
    )
```

Parameters. A single parameter *`runChecks`*. Use `true` to enable ownership checks, `false` to disable them.

Return value. *None*.

2.3.30.29 NdbTransaction::unlock()

Description. This method creates an unlock operation on the current transaction; when executed, the unlock operation removes the lock referenced by the `NdbLockHandle` (see Section 2.3.25.5, “NdbOperation::getLockHandle”) passed to the method.

Signature.

```
const NdbOperation* unlock
    (
      const NdbLockHandle* lockHandle,
      NdbOperation::AbortOption ao = NdbOperation::DefaultAbortOption
    )
```

Parameters. A pointer to a lock handle; in addition, optionally, an `AbortOption` value *`ao`*.

In the event that the unlock operation fails—for example, due to the row already being unlocked—the `AbortOption` specifies how this is handled, the default being that errors cause transactions to abort.

Return value. A pointer to an `NdbOperation` (the unlock operation created).

2.3.30.30 NdbTransaction::updateTuple()

Description. Updates a tuple using an `NdbRecord` object.

Signature.

```
const NdbOperation* updateTuple
    (
      const NdbRecord* key_rec,
      const char* key_row,
      const NdbRecord* attr_rec,
```

```
      const char* attr_row,
      const unsigned char* mask = 0,
      const NdbOperation::OperationOptions* opts = 0,
      Uint32 sizeOfOptions = 0
    )
```

Parameters. `updateTuple()` takes the following parameters:

- *`key_rec`* is a pointer to an `NdbRecord` for either a table or an index. If on a table, then the operation uses a primary key; if on an index, then the operation uses a unique key. In either case, the *`key_rec`* must include all columns of the key.
- The *`key_row`* passed to this method defines the primary or unique key of the affected tuple, and must remain valid until `execute()` is called.
- *`attr_rec`* is an `NdbRecord` referencing the attribute to be updated.

Note

For unique index operations, the *`attr_rec`* must refer to the underlying table of the index, not to the index itself.

- *`attr_row`* is a buffer containing the new data for the update.
- The *`mask`*, if not `NULL`, defines a subset of attributes to be updated. The mask is copied, and so does not need to remain valid after the call to this method returns.
- `OperationOptions` (*`opts`*) can be used to provide more finely-grained control of operation definitions. An `OperationOptions` structure is passed with flags indicating which operation definition options are present. Not all operation types support all operation options; for the options supported by each type of operation, see Section 2.3.30.21, “NdbTransaction::readTuple()”.
- The optional *`sizeOfOptions`* parameter is used to preserve backward compatibility of this interface with previous definitions of the `OperationOptions` structure. If an unusual size is detected by the interface implementation, it can use this to determine how to interpret the passed `OperationOptions` structure. To enable this functionality, the caller should pass `sizeof(NdbOperation::OperationOptions)` for the value of this argument.

Return value. The `NdbOperation` representing this operation (can be used to check for errors).

2.3.30.31 NdbTransaction::writeTuple()

Description. This method is used with `NdbRecord` to write a tuple of data.

Signature.

```
const NdbOperation* writeTuple
    (
      const NdbRecord* key_rec,
      const char* key_row,
      const NdbRecord* attr_rec,
      const char* attr_row,
      const unsigned char* mask = 0,
      const NdbOperation::OperationOptions* opts = 0,
      Uint32 sizeOfOptions = 0
    )
```

Parameters. This method takes the following parameters:

- *`key_rec`* is a pointer to an `NdbRecord` for either a table or an index. If on a table, then the operation uses a primary key; if on an index, then the operation uses a unique key. In either case, the *`key_rec`* must include all columns of the key.
- The *`key_row`* passed to this method defines the primary or unique key of the tuple to be written, and must remain valid until `execute()` is called.

- *attr_rec* is an NdbRecord referencing the attribute to be written.

Note

For unique index operations, the *attr_rec* must refer to the underlying table of the index, not to the index itself.

- *attr_row* is a buffer containing the new data.
- The *mask*, if not NULL, defines a subset of attributes to be written. The mask is copied, and so does not need to remain valid after the call to this method returns.
- OperationOptions (*opts*) can be used to provide more finely-grained control of operation definitions. An OperationOptions structure is passed with flags indicating which operation definition options are present. Not all operation types support all operation options; for the options supported by each type of operation, see Section 2.3.30.21, “NdbTransaction::readTuple()”.
- The optional *sizeOfOptions* parameter is used to provide backward compatibility of this interface with previous definitions of the OperationOptions structure. If an unusual size is detected by the interface implementation, it can use this to determine how to interpret the passed OperationOptions structure. To enable this functionality, the caller should pass sizeof(NdbOperation::OperationOptions) for the value of this argument.

Return value. A const pointer to the NdbOperation representing this write operation. The operation can be checked for errors if and as necessary.

2.3.31 The Object Class

This class provides meta-information about database objects such as tables and indexes. Object subclasses model these and other database objects.

Parent class. NdbDictionary

Child classes. Datafile, Event, Index, LogfileGroup, Table, Tablespace, Undofile, HashMap, ForeignKey

Methods. The following table lists the public methods of the Object class and the purpose or use of each method:

Method	Purpose / Use
getObjectId()	Gets an object's ID
getObjectStatus()	Gets an object's status
getObjectVersion()	Gets the version of an object

Note

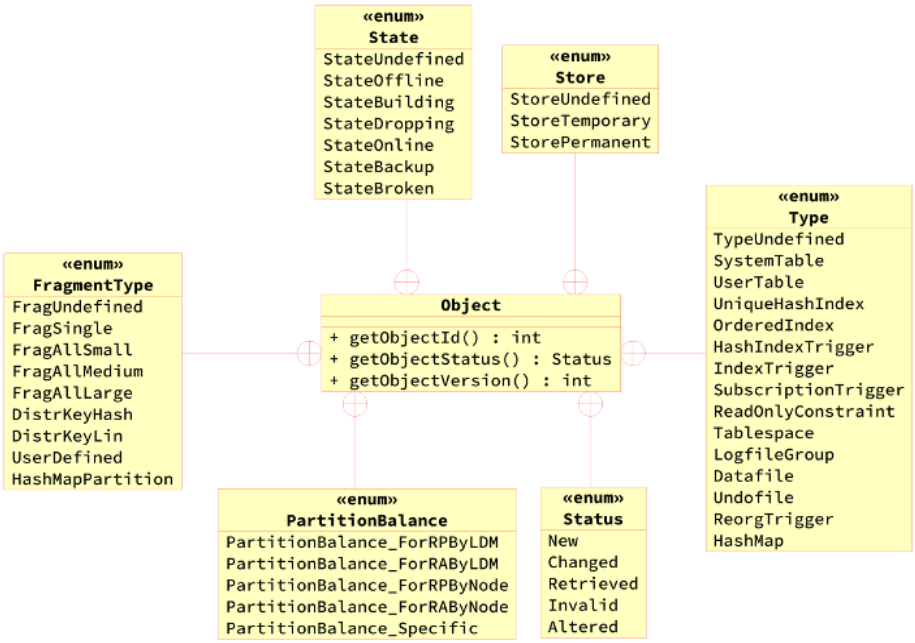
All 3 of these methods are pure virtual methods, and are reimplemented in the Table, Index, and Event subclasses where needed.

Types. These are the public types of the Object class:

Type	Purpose / Use
FragmentType	Fragmentation type used by the object (a table or index)
State	The object's state (whether it is usable)
Status	The object's state (whether it is available)
Store	Whether the object has been temporarily or permanently stored
Type	The object's type (what sort of table, index, or other database object the Object represents)

This diagram shows all public members of the `Object` class:

Figure 2.26 NdbDictionary::Object



For a visual representation of `Object`'s subclasses, see Section 2.3.19, “The NdbDictionary Class”.

2.3.31.1 Object::FragmentType

This type describes the `Object`'s fragmentation type.

Description. This parameter specifies how data in the table or index is distributed among the cluster's storage nodes, that is, the number of fragments per node. The larger the table, the larger the number of fragments that should be used. Note that all replicas count as a single fragment. For a table, the default is `FragAllMedium`. For a unique hash index, the default is taken from the underlying table and cannot currently be changed.

Enumeration values. Possible values for `FragmentType` are shown, along with descriptions, in the following table:

Value	Description
`FragUndefined`	The fragmentation type is undefined or the default
`FragAllMedium`	Two fragments per node
`FragAllLarge`	Four fragments per node
`DistrKeyHash`	Distributed hash key
`DistrKeyLin`	Distributed linear hash key
`UserDefined`	User defined
`HashMapPartition`	Hash map partition

2.3.31.2 Object::PartitionBalance

Description. This type enumerates provides partition balance settings (fragment count types) from which to choose when using `setPartitionBalance()`. This is also the type returned by `getPartitionBalance()`

Enumeration values. Possible values for `PartitionBalance` are shown, along with descriptions, in the following table:

Value	Description
`PartitionBalance_ForRPByLDM`	Use one fragment per LDM per node
`PartitionBalance_ForRAByLDM`	Use one fragment per LDM per node group
`PartitionBalance_ForRPByNode`	Use one fragment per node
`PartitionBalance_ForRAByNode`	Use one fragment per node group
`PartitionBalance_Specific`	Use setting determined by `setPartitionBalance()`

Prior to NDB 7.5.4, this was known as `FragmentCountType`, and could take one of the values `FragmentCount_OnePerLDMPerNode`, `FragmentCount_OnePerLDMPerNodeGroup`, `FragmentCount_OnePerNode`, `FragmentCount_OnePerNodeGroup`, or `FragmentCount_Specific`. These values correspond to those shown in the previous table, in the order shown.

2.3.31.3 Object::State

This type describes the state of the `Object`.

Description. This parameter provides us with the object's state. By *state*, we mean whether or not the object is defined and is in a usable condition.

Enumeration values. Possible values for `State` are shown, along with descriptions, in the following table:

Value	Description
`StateUndefined`	Undefined
`StateOffline`	Offline, not useable
`StateBuilding`	Building (e.g. restore?), not useable(?)
`StateDropping`	Going offline or being dropped; not usable
`StateOnline`	Online, usable
`StateBackup`	Online, being backed up, usable
`StateBroken`	Broken; should be dropped and re-created

2.3.31.4 Object::Status

This type describes the `Object`'s status.

Description. Reading an object's `Status` tells whether or not it is available in the `NDB` kernel.

Enumeration values. Possible values for `Status` are shown, along with descriptions, in the following table:

Value	Description
`New`	The object exists only in memory, and has not yet been created in the `NDB` kernel
`Changed`	The object has been modified in memory, and must be committed in the `NDB` Kernel for changes to take effect
`Retrieved`	The object exists, and has been read into main memory from the `NDB` Kernel
`Invalid`	The object has been invalidated, and should no longer be used
`Altered`	The table has been altered in the `NDB` kernel, but is still available for use

2.3.31.5 Object::Store

This type describes the `Object`'s persistence.

Description. Reading this value tells us is the object is temporary or permanent.

Enumeration values. Possible values for `Store` are shown, along with descriptions, in the following table:

Value	Description
`StoreUndefined`	The object is undefined
`StoreTemporary`	Temporary storage; the object or data will be deleted on system restart
`StorePermanent`	The object or data is permanent; it has been logged to disk

2.3.31.6 Object::Type

This type describes the type of the `Object`.

Description. The `Type` of the object can be one of several different sorts of index, trigger, tablespace, and so on.

Enumeration values. Possible values for `Type` are shown, along with descriptions, in the following table:

Value	Description
`TypeUndefined`	Undefined
`SystemTable`	System table
`UserTable`	User table (may be temporary)
`UniqueHashIndex`	Unique (but unordered) hash index
`OrderedIndex`	Ordered (but not unique) index
`HashIndexTrigger`	Index maintenance (*internal*)
`IndexTrigger`	Index maintenance (*internal*)
`SubscriptionTrigger`	Backup or replication (*internal*)
`ReadOnlyConstraint`	Trigger (*internal*)
`Tablespace`	Tablespace
`LogfileGroup`	Logfile group
`Datafile`	Datafile
`Undofile`	Undofile
`ReorgTrigger`	Trigger
`HashMap`	Hash map
`ForeignKey`	Foreign key
`FKParentTrigger`	Trigger on a foreign key's parent table
`FKChildTrigger`	Trigger on a foreign key's child table

`HashMap` was added in NDB 7.2.7.

`ForeignKey`, `FKParentTrigger`, and `FKChildTrigger` were added in NDB Cluster 7.3. See Section 2.3.8, “The ForeignKey Class”.

2.3.31.7 Object::getObjectId()

Description. This method retrieves the object's ID.

Signature.

```
virtual int getObjectId
    (
      void
    ) const
```

Parameters. *None*.

Return value. The object ID, an integer.

2.3.31.8 Object::getObjectStatus()

Description. This method retrieves the status of the object for which it is invoked.

Signature.

```
virtual Status getObjectStatus
    (
      void
    ) const
```

Parameters. *None*.

Return value. Returns the current `Status` of the `Object`.

2.3.31.9 Object::getObjectVersion()

Description. The method gets the current version of the object.

Signature.

```
virtual int getObjectVersion
    (
      void
    ) const
```

Parameters. *None*.

Return value. The object's version number, an integer.

2.3.32 The OperationOptions Structure

Parent class. `NdbOperation`

Description. These options are passed to the `NdbRecord`-based primary key and scan takeover operation methods defined in the `NdbTransaction` and `NdbScanOperation` classes.

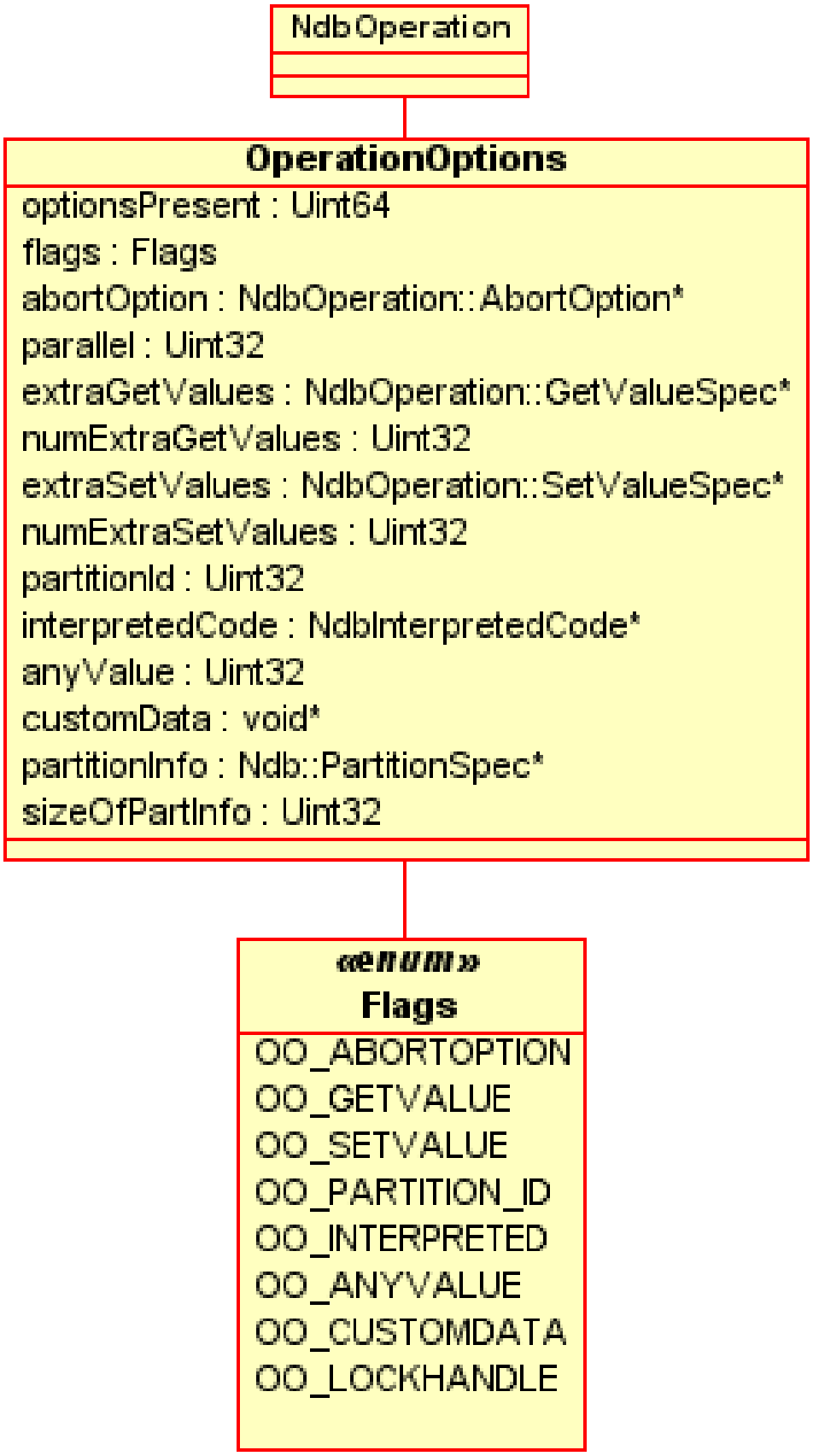
Note

Most `NdbTransaction::*Tuple()` methods (see Section 2.3.30, "The NdbTransaction Class") take a supplementary *`sizeOfOptions`* parameter. This is optional, and is intended to permit the interface implementation to remain backward compatible with older un-recompiled clients that may pass an older (smaller) version of the `OperationOptions` structure. This effect is achieved by passing `sizeof(OperationOptions)` into this parameter.

Each option type is marked as present by setting the corresponding bit in *`optionsPresent`*. (Only the option types marked in *`optionsPresent`* need have sensible data.) All data is copied out of the `OperationOptions` structure (and any subtended structures) at operation definition time. If no options are required, then `NULL` may be passed instead.

Members. The makeup of this structure is shown here:

Figure 2.27 NdbOperation::OperationOptions



The individual members are described in more detail in the following table:

Name	Type	Description
`optionsPresent`	`Uint64`	Which flags are present.
[...]	`Flags`: The accepted names and values are shown in the following list: • `OO_ABORTOPTION`: `0x01` • `OO_GETVALUE`: `0x02` • `OO_SETVALUE`: `0x04` • `OO_PARTITION_ID`: `0x08` • `OO_INTERPRETED`: `0x10` • `OO_ANYVALUE`: `0x20` • `OO_CUSTOMDATA`: `0x40` • `OO_LOCKHANDLE`: `0x80`	Type of flags.
`abortOption`	`AbortOption`	An operation-specific abort option; necessary only if the default abortoption behavior is not satisfactory.
`extraGetValues`	`GetValueSpec`	Extra column values to be read.
`numExtraGetValues`	`Uint32`	Number of extra column values to be read.
`extraSetValues`	`SetValueSpec`	Extra column values to be set.
`numExtraSetValues`	`Uint32`	Number of extra column values to be set.
`partitionId`	`Uint32`	Limit the scan to the partition having this ID; alternatively, you can supply an `PartitionSpec` here. For index scans, partitioning information can be supplied for each range.
`interpretedCode`	`NdbInterpretedCode`	Interpeted code to execute as part of the scan.
`anyValue`	`Uint32`	An `anyValue` to be used with this operation. This is used by NDB Cluster Replication to store the SQL node's server ID. By starting the SQL node with the `--server-id-bits` option (which causes only some of the `server_id`'s bits to be used for uniquely identifying it) set to less than 32, the remaining bits can be used to store user data.
`customData`	`void*`	Data pointer to associate with this operation.
`partitionInfo`	`PartitionSpec`	Partition information for bounding this scan.
`sizeOfPartInfo`	`Uint32`	Size of the bounding partition information.

For more information, see Section 2.3.27, “The NdbRecord Interface”.

2.3.33 The PartitionSpec Structure

This section describes the `PartitionSpec` structure.

Parent class. `Ndb`

Description. A `PartitionSpec` is used for describing a table partition in terms of any one of the following criteria:

- A specific partition ID for a table with user-defined partitioning.
- An array made up of a table's distribution key values for a table with native partitioning.
- A row in `NdbRecord` format containing a natively partitioned table's distribution key values.

Attributes. A `PartitionSpec` has two attributes, a `SpecType` and a `Spec` which is a data structure corresponding to that `SpecType`, as shown in the following table:

`SpecType` Enumeration	**`SpecType` Value (`Uint32`)**	**Data Structure**	**Description**
`PS_NONE`	`0`	*none*	No partitioning information is provided.
`PS_USER_DEFINED`	`1`	`UserDefined`	For a table having user-defined partitioning, a specific partition is identified by its partition ID.
`PS_DISTR_KEY_PART_PTR`	`2`	`KeyPartPtr`	For a table having native partitioning, an array containing the table's distribution key values is used to identify the partition.
`PS_DISTR_KEY_RECORD`	`3`	`KeyRecord`	The partition is identified using a natively partitioned table's distribution key values, as contained in a row given in `NdbRecord` format.

UserDefined structure. This structure is used when the `SpecType` is `PS_USER_DEFINED`.

Attribute	**Type**	**Description**
`partitionId`	`Uint32`	The partition ID for the desired table.

KeyPartPtr structure. This structure is used when the `SpecType` is `PS_DISTR_KEY_PART_PTR`.

Attribute	**Type**	**Description**
`tableKeyParts`	`Key_part_ptr`	Pointer to the distribution key values for a table having native partitioning.
`xfrmbuf`	`void*`	Pointer to a temporary buffer used for performing calculations.
`xfrmbuflen`	`Uint32`	Length of the temporary buffer.

KeyRecord structure. This structure is used when the `SpecType` is `PS_DISTR_KEY_RECORD`.

Attribute	Type	Description
keyRecord	NdbRecord	A row in NdbRecord format, containing a table's distribution keys.
keyRow	const char*	The distribution key data.
xfrmbuf	void*	Pointer to a temporary buffer used for performing calculations.
xfrmbuflen	Uint32	Length of the temporary buffer.

Definition from Ndb.hpp. Because this is a fairly complex structure, we here provide the original source-code definition of PartitionSpec, as given in storage/ndb/include/ndbapi/Ndb.hpp:

```
struct PartitionSpec
{
  enum SpecType
  {
    PS_NONE                = 0,
    PS_USER_DEFINED        = 1,
    PS_DISTR_KEY_PART_PTR  = 2,
    PS_DISTR_KEY_RECORD    = 3
  };

  Uint32 type;

  union
  {
    struct {
      Uint32 partitionId;
    } UserDefined;

    struct {
      const Key_part_ptr* tableKeyParts;
      void* xfrmbuf;
      Uint32 xfrmbuflen;
    } KeyPartPtr;

    struct {
      const NdbRecord* keyRecord;
      const char* keyRow;
      void* xfrmbuf;
      Uint32 xfrmbuflen;
    } KeyRecord;
  };
};
```

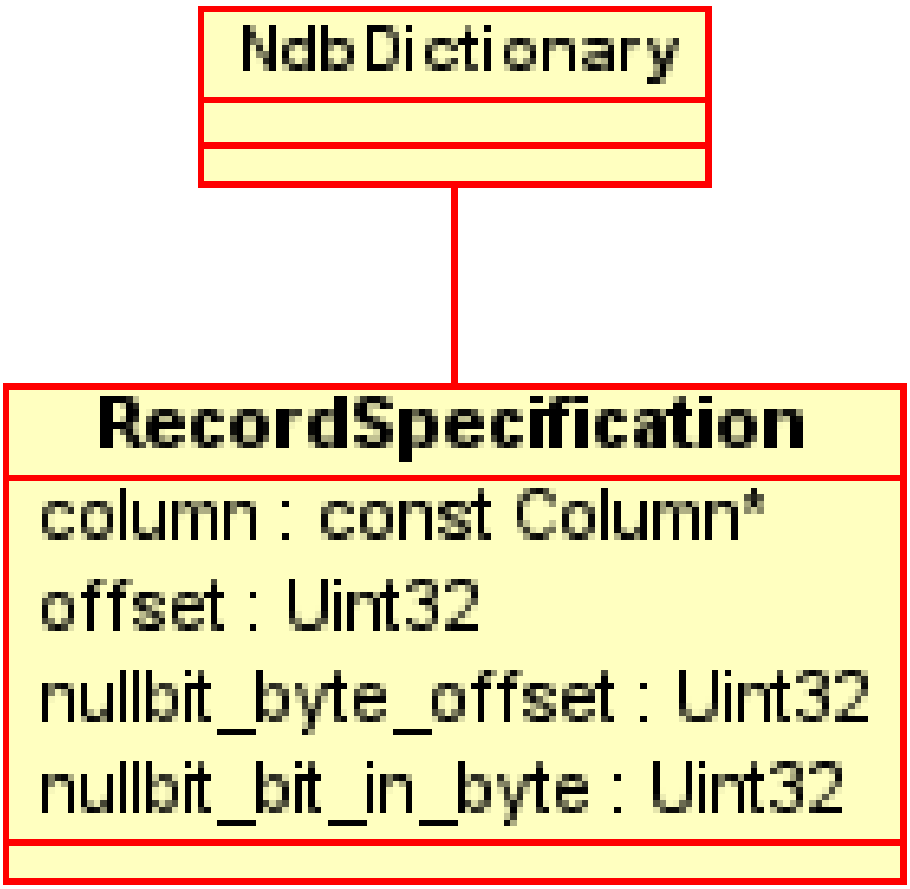
2.3.34 The RecordSpecification Structure

Parent class. NdbDictionary

Description. This structure is used to specify columns and range offsets when creating NdbRecord objects.

Members. The makeup of this structure is shown here:

Figure 2.28 NdbDictionary::RecordSpecification



The individual members are described in more detail in the following table:

Name	Type	Description
`column`	`Column`	The column described by this entry (the column's maximum size defines the field size for the row). Even when creating an `NdbRecord` for an index, this must point to a column obtained from the underlying table, and not from the index itself.
`offset`	`Uint32`	The offset of data from the beginning of a row. For reading blobs, the blob handle (`NdbBlob`), rather than the actual blob data, is written into the row. This means that there must be at least `sizeof(NdbBlob*)` must be available in the row.
`nullbit_byte_offset`	`Uint32`	The offset from the beginning of the row of the byte containing the `NULL` bit.
`nullbit_bit_in_byte`	`Uint32`	`NULL` bit (0-7).

Important

`nullbit_byte_offset` and `nullbit_bit_in_byte` are not used for non-`NULL`able columns.

For more information, see Section 2.3.27, “The NdbRecord Interface”.

2.3.35 The ScanOptions Structure

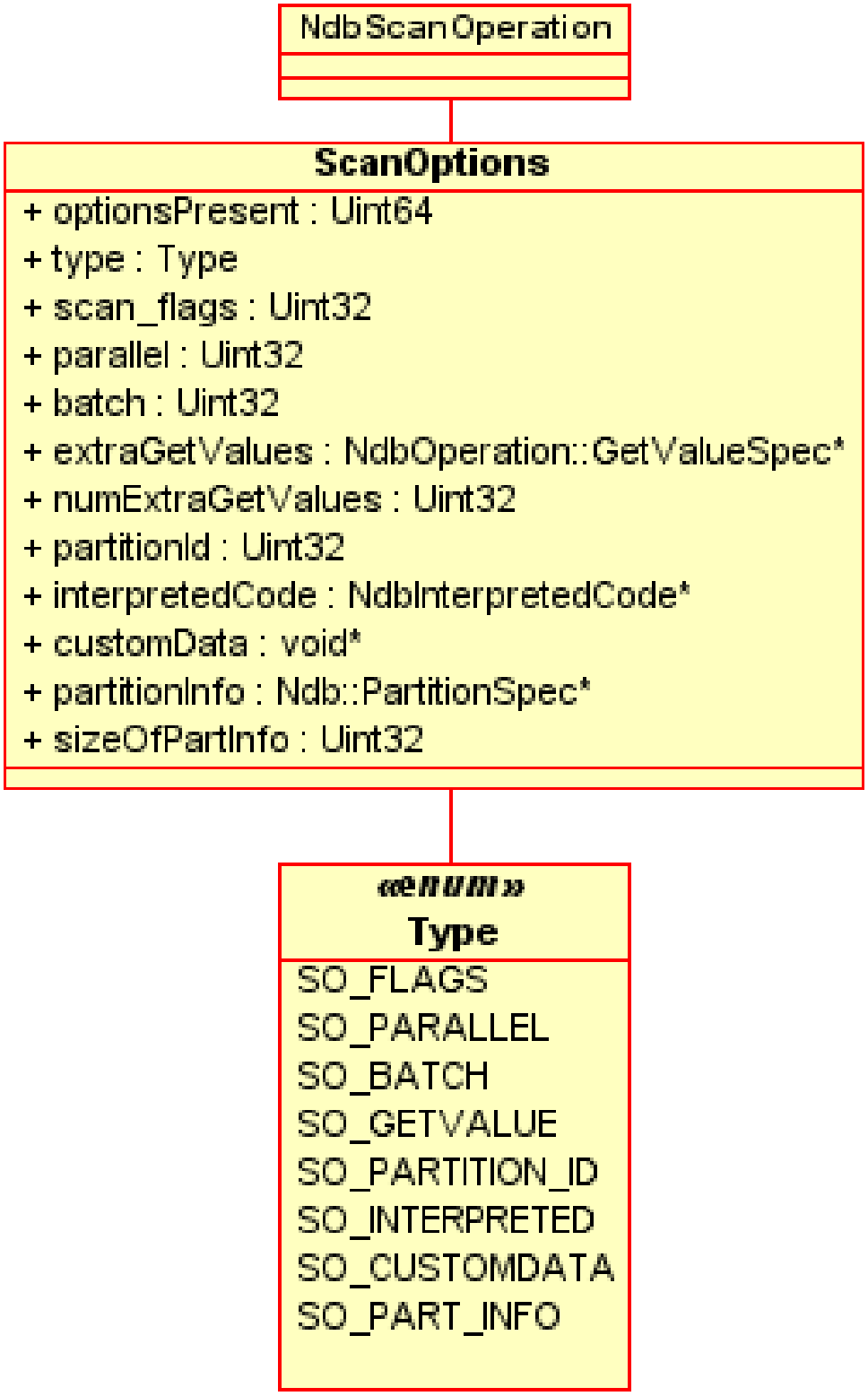
Parent class. `NdbScanOperation`

Description. This data structure is used to pass options to the `NdbRecord`-based `scanTable()` and `scanIndex()` methods of the `NdbTransaction` class. Each option type is marked as present by setting the corresponding bit in the `optionsPresent` field. Only the option types marked in the `optionsPresent` field need have sensible data.

All data is copied out of the `ScanOptions` structure (and any subtended structures) at operation definition time. If no options are required, then `NULL` may be passed as the `ScanOptions` pointer.

Members. The makeup of this structure is shown here:

Figure 2.29 NdbScanOperation::ScanOptions



The individual members are described in more detail in the following table:

Name	Type	Description
`optionsPresent`	`Uint64`	Which options are present.
[...]	`Type`:	Type of options.

Name	Type	Description
	• SO_SCANFLAGS: 0x01 • SO_PARALLEL: 0x02 • SO_BATCH: 0x04 • SO_GETVALUE: 0x08 • SO_PARTITION_ID: 0x10 • SO_INTERPRETED: 0x20 • SO_CUSTOMDATA: 0x40 • SO_PARTINFO: 0x80	
scan_flags	Uint32	Flags controlling scan behavior; see Section 2.3.29.9, "NdbScanOperation::ScanFlag", for more information.
parallel	Uint32	Scan parallelism; 0 (the default) sets maximum parallelism.
batch	Uint32	Batch size for transfers from data nodes to API nodes; 0 (the default) enables this to be selected automatically.
extraGetValues	GetValueSpec	Extra values to be read for each row matching the sdcan criteria.
numExtraGetValues	Uint32	Number of extra values to be read.
partitionId	Uint32	Limit the scan to the partition having this ID; alternatively, you can supply an PartitionSpec here. For index scans, partitioning information can be supplied for each range.
interpretedCode	NdbInterpretedCode	Interpeted code to execute as part of the scan.
customData	void*	Data pointer to associate with this scan operation.
partitionInfo	PartitionSpec	Partition information for bounding this scan.
sizeOfPartInfo	Uint32	Size of the bounding partition information.

For more information, see Section 2.3.27, "The NdbRecord Interface".

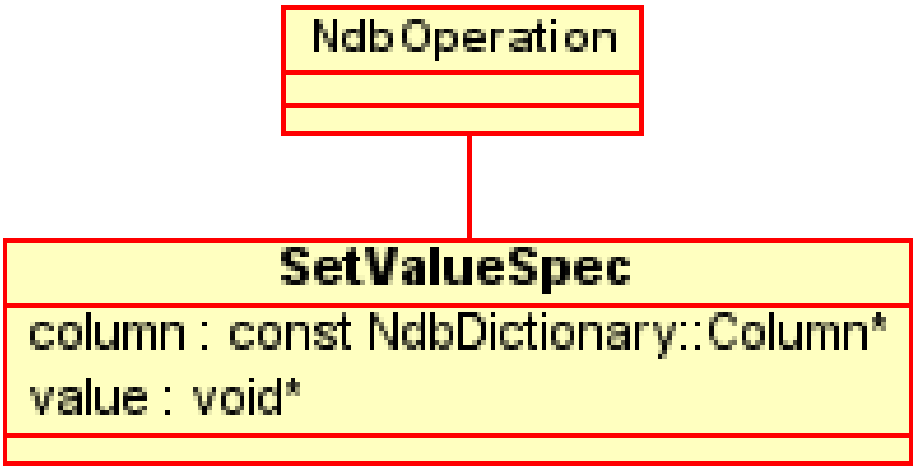
2.3.36 The SetValueSpec Structure

Parent class. NdbOperation

Description. This structure is used to specify an extra value to set as part of an NdbRecord operation.

Members. The makeup of this structure is shown here:

Figure 2.30 NdbOperation::SetValueSpec



The individual members are described in more detail in the following table:

Name	Type	Description
`column`	`Column`	To specify an extra value to read, the caller must provide this, as well as (optionally `NULL`) *`appStorage`* pointer.
`value`	`void*`	This must point to the value to be set, or to `NULL` if the attribute is to be set to `NULL`. The value pointed to is copied when the operation is defined, and need not remain in place until execution time.

Important

Currently, blob values cannot be set using `SetValueSpec`.

For more information, see Section 2.3.27, “The NdbRecord Interface”.

2.3.37 The Table Class

This section describes the `Table` class, which models a database table in the NDB API.

Parent class. `NdbDictionary`

Child classes. *None*

Description. The `Table` class represents a table in an NDB Cluster database. This class extends the `Object` class, which in turn is an inner class of the `NdbDictionary` class.

Important

It is possible using the NDB API to create tables independently of the MySQL server. However, it is usually not advisable to do so, since tables created in this fashion cannot be seen by the MySQL server. Similarly, it is possible using `Table` methods to modify existing tables, but these changes (except for renaming tables) are not visible to MySQL.

Calculating Table Sizes. When calculating the data storage one should add the size of all attributes (each attribute consuming a minimum of 4 bytes) and well as 12 bytes overhead. Variable size attributes have a size of 12 bytes plus the actual data storage parts, with an additional overhead based on the size of the variable part. For example, consider a table with 5 attributes: one 64-bit attribute,

one 32-bit attribute, two 16-bit attributes, and one array of 64 8-bit attributes. The amount of memory consumed per record by this table is the sum of the following:

- 8 bytes for the 64-bit attribute
- 4 bytes for the 32-bit attribute
- 8 bytes for the two 16-bit attributes, each of these taking up 4 bytes due to right-alignment
- 64 bytes for the array (64 * 1 byte per array element)
- 12 bytes overhead

This totals 96 bytes per record. In addition, you should assume an overhead of about 2% for the allocation of page headers and wasted space. Thus, 1 million records should consume 96 MB, and the additional page header and other overhead comes to approximately 2 MB. Rounding up yields 100 MB.

Methods. The following table lists the public methods of this class and the purpose or use of each method:

Method	Purpose / Use
`Table()`	Class constructor
`~Table()`	Destructor
`addColumn()`	Adds a column to the table
`aggregate()`	Computes aggregate data for the table
`equal()`	Compares the table with another table
`getColumn()`	Gets a column (by name) from the table
`getDefaultNoPartitionsFlag()`	Checks whether the default number of partitions is being used
`getFragmentCount()`	Gets the number of fragments for this table
`getFragmentData()`	Gets table fragment data (ID, state, and node group)
`getFragmentDataLen()`	Gets the length of the table fragment data
`getFragmentNodes()`	Gets IDs of data nodes on which fragments are located
`getFragmentType()`	Gets the table's `FragmentType`
`getFrmData()`	Gets the data from the table `.FRM` file
`getFrmLength()`	Gets the length of the table's `.FRM` file
`getHashMap()`	Gets the table's hash map.
`getKValue()`	Gets the table's `KValue`
`getLinearFlag()`	Gets the current setting for the table's linear hashing flag
`getLogging()`	Checks whether logging to disk is enabled for this table
`getMaxLoadFactor()`	Gets the table's maximum load factor
`getMaxRows()`	Gets the maximum number of rows that this table may contain
`getMinLoadFactor()`	Gets the table's minimum load factor
`getName()`	Gets the table's name
`getNoOfColumns()`	Gets the number of columns in the table
`getNoOfPrimaryKeys()`	Gets the number of columns in the table's primary key.
`getObjectId()`	Gets the table's object ID
`getObjectStatus()`	Gets the table's object status

Method	Purpose / Use
getObjectType()	Removed in NDB 7.5.0 (Bug #47960, Bug #11756088)
getObjectVersion()	Gets the table's object version
getPartitionBalance()	Gets partition balance (fragment count type) used for this table (NDB 7.5.4 and later)
getPartitionBalanceString()	Gets partition balance used for this table, as a string (NDB 7.5.4 and later)
getPartitionId()	Gets a partition ID from a hash value
getPrimaryKey()	Gets the name of the table's primary key
getRangeListData()	Gets a RANGE or LIST array
getRangeListDataLen()	Gets the length of the table RANGE or LIST array
getRowChecksumIndicator()	Checks whether the row checksum indicator has been set
getRowGCIIndicator()	Checks whether the row GCI indicator has been set
getTableId()	Gets the table's ID
getTablespace()	Gets the tablespace containing this table
getTablespaceData()	Gets the ID and version of the tablespace containing the table
getTablespaceDataLen()	Gets the length of the table's tablespace data
getTablespaceNames()	Gets the names of the tablespaces used in the table fragments
hasDefaultValues()	Determine whether table has any columns using default values
setDefaultNoPartitionsFlag()	Toggles whether the default number of partitions should be used for the table
getFragmentCount()	Gets the number of fragments for this table
setFragmentData()	Sets the fragment ID, node group ID, and fragment state
setFragmentType()	Sets the table's FragmentType
setFrm()	Sets the .FRM file to be used for this table
setHashMap()	Sets the table's hash map.
setKValue()	Set the KValue
setLinearFlag()	Sets the table's linear hashing flag
setLogging()	Toggle logging of the table to disk
setMaxLoadFactor()	Set the table's maximum load factor (MaxLoadFactor)
setMaxRows()	Sets the maximum number of rows in the table
setMinLoadFactor()	Set the table's minimum load factor (MinLoadFactor)
setPartitionBalance()	Sets the partition balance (fragment count type) for this table (NDB 7.5.4 and later)
setName()	Sets the table's name
setObjectType()	Removed in NDB 7.5.0 (Bug #47960, Bug #11756088)
setRangeListData()	Sets LIST and RANGE partition data
setRowChecksumIndicator()	Sets the row checksum indicator
setRowGCIIndicator()	Sets the row GCI indicator

Method	Purpose / Use
`setStatusInvalid()`	
`setTablespace()`	Set the tablespace to use for this table
`setTablespaceData()`	Sets the tablespace ID and version
`setTablespaceNames()`	Sets the tablespace names for fragments
`validate()`	Validates the definition for a new table prior to creating it

The assignment (=) operator is overloaded for this class, so that it always performs a deep copy.

Note

As with other database objects, `Table` object creation and attribute changes to existing tables done using the NDB API are not visible from MySQL. For example, if you add a new column to a table using `Table::addColumn()`, MySQL cannot see the new column. The only exception to this rule with regard to tables is that a change of name of an existing `NDB` table using `Table::setName()` is visible to MySQL.

Types. The `Table` class defines a single public type `SingleUserMode`.

Class diagram. This diagram shows all the available methods of the `Table` class:

```
getFrmData() : const void*
getFrmLength() : Uint32
getHashMap(id : Uint32* = 0, version : Uint32* = 0) : bool
getKValue() : Int
getLinearFlag() : bool
getLogging() : bool
getMaxLoadFactor() : Int
getMaxRows() : Uint64
getMinLoadFactor() : Int
getName() : const char*
getNoOfColumns() : Int
getNoOfPrimaryKeys() : Int
getObjectId() : Int
getObjectStatus() : Object::Status
getObjectType() : Object::Type
getObjectVersion() : Int
getPartitionBalance()
getPartitionId(hashvalue : Uint32) : Uint32
getPrimaryKey(no : Int) : const char*
getRangeListData() : const void*
getRangeListDataLen() : Uint32
getRowChecksumIndicator() : bool
getRowGCIIndicator() : bool
getTableId() : Int
getTablespace() : const char*
getTablespace(id : Uint32*, version : Uint32*) : bool
getTablespaceData() : const void*
getTablespaceDataLen() : Uint32
getTablespaceNames() : const void*
getTablespaceNamesLen() : Uint32
hasDefaultValues() : bool
setDefaultNoPartitionsFlag(indicator : Uint32)
setFragmentData(data : const void*, len : Uint32)
setFrm(data : const void*, len : Uint32)
setFragmentCount(count : Uint32)
setFragmentType(fragmentType : FragmentType)
setHashMap(hashmap : const class HashMap&) : Int
setKValue(kValue : Int)
setLinearFlag(flag : Uint32)
setLogging(enable : bool)
setName(name : const char*)
setMinLoadFactor(min : Int)
setMaxRows(maxRows : Uint64)
setTablespace(name : const char*)
setObjectType(type : Object::Type)
setPartitionBalance(scheme : Object::PartitionBalance)
setRangeListData(data : const void*, len : Uint32)
setRowChecksumIndicator(value : bool)
setRowGCIIndicator(value : bool)
setStatusInvalid()
setMaxLoadFactor(max : Int)
setTablespace(tablespace : class const Tablespace&)
setTablespaceData(data : const void*, len : Uint32)
setTablespaceNames(data : const void*, len : Uint32)
validate(error : struct NdbError&) : Int
operator =(table : const Table&) : Table&
```

2.3.37.1 Table::addColumn()

Description. Adds a column to a table.

Signature.

```
void addColumn
    (
      const Column& column
    )
```

Parameters. A reference to the column which is to be added to the table.

Return value. *None*; however, it does create a copy of the original `Column` object.

2.3.37.2 Table::aggregate()

Description. This method computes aggregate data for the table. It is required in order for aggregate methods such as `getNoOfPrimaryKeys()` to work properly before the table has been created and retrieved via `getTableId()`.

Note

This method was added in MySQL 5.1.12 (see Bug #21690).

Signature.

```
int aggregate
    (
      struct NdbError& error
    )
```

Parameters. A reference to an `NdbError` object.

Return value. An integer, whose value is `0` on success, and `-1` if the table is in an inconsistent state. In the latter case, the *`error`* is also set.

2.3.37.3 Table Constructor

Description. Creates a `Table` instance. There are two versions of the `Table` constructor, one for creating a new instance, and a copy constructor.

Important

Tables created in the NDB API using this method are not accessible from MySQL.

Signature. New instance:

```
Table
    (
      const char* name = ""
    )
```

Copy constructor:

```
Table
    (
      const Table& table
    )
```

Parameters. For a new instance, the name of the table to be created. For a copy, a reference to the table to be copied.

Return value. A `Table` object.

Destructor.

```
virtual ~Table()
```

2.3.37.4 Table::equal()

Description. This method is used to compare one instance of `Table` with another.

Signature.

```
bool equal
    (
      const Table& table
    ) const
```

Parameters. A reference to the `Table` object with which the current instance is to be compared.

Return value. `true` if the two tables are the same, otherwise `false`.

2.3.37.5 Table::getDefaultNoPartitionsFlag()

Description. This method is used to find out whether the default number of partitions is used for the table.

Signature.

```
Uint32 getDefaultNoPartitionsFlag
    (
      void
    ) const
```

Parameters. *None*.

Return value. A 32-bit unsigned integer.

2.3.37.6 Table::getFragmentCount()

Description. This method gets the number of fragments in the table.

Signature.

```
Uint32 getFragmentCount
    (
      void
    ) const
```

Parameters. *None*.

Return value. The number of table fragments, as a 32-bit unsigned integer.

2.3.37.7 Table::getFragmentData()

Description. This method gets the table's fragment data (ID, state, and node group).

Signature.

```
const void* getFragmentData
    (
      void
    ) const
```

Parameters. *None*.

Return value. A pointer to the data to be read.

2.3.37.8 Table::getFragmentDataLen()

Description. Gets the length of the table fragment data to be read, in bytes.

Signature.

```
Uint32 getFragmentDataLen
    (
      void
    ) const
```

Parameters. *None*.

Return value. The number of bytes to be read, as an unsigned 32-bit integer.

2.3.37.9 Table::getFrmData()

Description. The the data from the `.FRM` file associated with the table.

Signature.

```
const void* getFrmData
    (
      void
    ) const
```

Parameters. *None*.

Return value. A pointer to the `.FRM` data.

2.3.37.10 Table::getFrmLength()

Description. Gets the length of the table's `.FRM` file data, in bytes.

Signature.

```
Uint32 getFrmLength
    (
      void
    ) const
```

Parameters. *None*.

Return value. The length of the `.FRM` file data (an unsigned 32-bit integer).

2.3.37.11 Table::getColumn()

Description. This method is used to obtain a column definition, given either the index or the name of the column.

Signature. This method can be invoked using either the column ID or column name, as shown here:

```
Column* getColumn
    (
      const int AttributeId
    )

Column* getColumn
    (
      const char* name
    )
```

Parameters. Either of: the column's index in the table (as it would be returned by the column's `getColumnNo()` method), or the name of the column.

Return value. A pointer to the column with the specified index or name. If there is no such column, then this method returns `NULL`.

2.3.37.12 Table::getFragmentNodes()

Description. This method retrieves a list of nodes storing a given fragment.

Signature.

```
Uint32 getFragmentNodes
    (
      Uint32 fragmentId,
      Uint32* nodeIdArrayPtr,
      Uint32 arraySize
    ) const
```

Parameters. This method takes the following three parameters:

- *`fragmentId`*: The ID of the desired fragment.
- *`nodeIdArrayPtr`*: Pointer to an array of node IDs of the nodes containing this fragment.

  Note

  Normally, the primary fragment is entry 0 in this array.

- *`arraySize`*: The size of the array containing the node IDs. If this is less than the number of fragments, then only the first *`arraySize`* entries are written to this array.

Return value. A return value of `0` indicates an error; otherwise, this is the number of table fragments, as a 32-bit unsigned integer.

2.3.37.13 Table::getFragmentType()

Description. This method gets the table's fragmentation type.

Signature.

```
FragmentType getFragmentType
    (
      void
    ) const
```

Parameters. *None*.

Return value. A `FragmentType` value, as defined in Section 2.3.31.1, “Object::FragmentType”.

2.3.37.14 Table::getHashMap()

Description. Get the hash map used for this table. Introduced in NDB 7.2.7.

Signature.

```
bool getHashMap
    (
      Uint32* id = 0,
      Uint32* version = 0
    ) const
```

Parameters. The table ID and version.

Return value. True if the table has a hash map, otherwise false.

2.3.37.15 Table::getKValue()

Description. This method gets the KValue, a hashing parameter which is currently restricted to the value 6. In a future release, it may become feasible to set this parameter to other values.

Signature.

```
int getKValue
    (
      void
    ) const
```

Parameters. *None*.

Return value. An integer (currently always 6).

2.3.37.16 Table::getLinearFlag()

Description. This method retrieves the value of the table's linear hashing flag.

Signature.

```
bool getLinearFlag
    (
      void
    ) const
```

Parameters. *None*.

Return value. `true` if the flag is set, and `false` if it is not.

2.3.37.17 Table::getLogging()

Description. This class is used to check whether a table is logged to disk—that is, whether it is permanent or temporary.

Signature.

```
bool getLogging
    (
      void
    ) const
```

Parameters. *None*.

Return value. Returns a Boolean value. If this method returns `true`, then full checkpointing and logging are done on the table. If `false`, then the table is a temporary table and is not logged to disk;

in the event of a system restart the table still exists and retains its definition, but it will be empty. The default logging value is `true`.

2.3.37.18 Table::getMaxLoadFactor()

Description. This method returns the load factor (a hashing parameter) when splitting of the containers in the local hash tables begins.

Signature.

```
int getMaxLoadFactor
    (
      void
    ) const
```

Parameters. *None*.

Return value. An integer whose maximum value is 100. When the maximum value is returned, this means that memory usage is optimised. Smaller values indicate that less data is stored in each container, which means that keys are found more quickly; however, this also consumes more memory.

2.3.37.19 Table::getMaxRows()

Description. This method gets the maximum number of rows that the table can hold. This is used for calculating the number of partitions.

Signature.

```
Uint64 getMaxRows
    (
      void
    ) const
```

Parameters. *None*.

Return value. The maximum number of table rows, as a 64-bit unsigned integer.

2.3.37.20 Table::getMinLoadFactor()

Description. This method gets the value of the load factor when reduction of the hash table begins. This should always be less than the value returned by `getMaxLoadFactor()`.

Signature.

```
int getMinLoadFactor
    (
      void
    ) const
```

Parameters. *None*.

Return value. An integer (actually, a percentage expressed as an integer; see Section 2.3.37.18, “Table::getMaxLoadFactor()”).

2.3.37.21 Table:getName()

Description. Gets the name of a table.

Signature.

```
const char* getName
    (
      void
    ) const
```

Parameters. *None*.

Return value. The name of the table (a string).

2.3.37.22 Table::getNoOfCOlumns()

Description. This method is used to obtain the number of columns in a table.

Signature.

```
int getNoOfColumns
    (
      void
    ) const
```

Parameters. *None*.

Return value. An integer representing the number of columns in the table.

2.3.37.23 Table::getNoOfPrimaryKeys()

Description. This method finds the number of primary key columns in the table.

Signature.

```
int getNoOfPrimaryKeys
    (
      void
    ) const
```

Parameters. *None*.

Return value. An integer representing the number of primary key columns in the table.

2.3.37.24 Table::getObjectId()

Description. This method gets the table's object ID.

Signature.

```
virtual int getObjectId
    (
      void
    ) const
```

Parameters. *None*.

Return value. The object ID is returned as an integer.

2.3.37.25 Table::getObjectStatus()

Description. This method gets the table's status—that is, its `Object::Status`.

Signature.

```
virtual Object::Status getObjectStatus
```

```
    (
      void
    ) const
```

Parameters. *None*.

Return value. A `Status` value. For possible values, see Section 2.3.31.4, “Object::Status”.

2.3.37.26 Table::getObjectType()

Description. This method did not work as intended, and was removed in NDB 7.5.0 (Bug #47960, Bug #11756088).

Signature.

```
Object::Type getObjectType
    (
      void
    ) const
```

Parameters. *None*.

Return value. Returns a `Type` value. For possible values, see Section 2.3.31.6, “Object::Type”.

2.3.37.27 Table::getObjectVersion()

Description. This method gets the table's object version (see NDB Schema Object Versions).

Signature.

```
virtual int getObjectVersion
    (
      void
    ) const
```

Parameters. *None*.

Return value. The table's object version, as an integer.

2.3.37.28 Table::getPartitionBalance()

Description. This method gets the table' partition balance scheme (fragment count type).

Signature.

```
Object::PartitionBalance getPartitionBalance
    (
      void
    ) const
```

Parameters. *None*.

Return value. The partition balancing scheme, as a value of type `Object::PartitionBalance`.

Prior to NDB 7.5.4, this method was known as `getFragmentCountType()`.

2.3.37.29 Table::getPartitionBalanceString()

Description. This method gets the table' partition balance scheme (fragment count type), and returns it as a string.

Signature.

```
const char* getPartitionBalanceString
    (
      void
    ) const
```

Parameters. *None*.

Return value. The partition balancing scheme, as a string value.

Prior to NDB 7.5.4, this method was known as `getFragmentCountTypeString()`.

2.3.37.30 Table::getPartitionId()

Description. Gets a table partition ID given its hash value.

Signature.

```
Uint32 getPartitionId
    (
      Uint32 hashvalue
    ) const
```

Parameters. A *`hashvalue`*. Note that if the table has not actually been retrieved (using, for example, `getTableId()`), then the result is likely not to be accurate or useful.

Return value. The identifier of the partition corresponding to the *`hashvalue`*.

2.3.37.31 Table::getPrimaryKey()

Description. This method is used to obtain the name of the table's primary key.

Signature.

```
const char* getPrimaryKey
    (
      int no
    ) const
```

Parameters. *None*.

Return value. The name of the primary key, a string (character pointer).

2.3.37.32 Table::getRangeListData()

Description. This method gets the range or list data associated with the table.

Signature.

```
const void* getRangeListData
    (
      void
    ) const
```

Parameters. *None*.

Return value. A pointer to the data.

2.3.37.33 Table::getRangeListDataLen()

Description. This method gets the size of the table's range or list array.

Signature.

```
Uint32 getRangeListDataLen
    (
      void
    ) const
```

Parameters. *None*.

Return value. The length of the list or range array, as an integer.

2.3.37.34 Table::getRowChecksumIndicator()

Description. Check whether the row checksum indicator has been set.

Signature.

```
bool getRowChecksumIndicator
    (
      void
    ) const
```

Parameters. *None*.

Return value. A `true` or `false` value.

2.3.37.35 Table::getRowGCIIndicator()

Description. Checks whether the row GCI indicator has been set.

Signature.

```
bool getRowGCIIndicator
    (
      void
    ) const
```

Parameters. *None*.

Return value. A `true` or `false` value.

2.3.37.36 Table::getTableId()

Description. This method gets a table's ID.

Signature.

```
int getTableId
    (
      void
    ) const
```

Parameters. *None*.

Return value. An integer.

2.3.37.37 Table::getTablespace()

Description. This method is used in two ways: to obtain the name of the tablespace to which this table is assigned; to verify that a given tablespace is the one being used by this table.

Signatures. To obtain the name of the tablespace, invoke without any arguments:

```
const char* getTablespace
    (
      void
    ) const
```

To determine whether the tablespace is the one indicated by the given ID and version, supply these as arguments, as shown here:

```
bool getTablespace
    (
      Uint32* id      = 0,
      Uint32* version = 0
    ) const
```

Parameters. The number and types of parameters depend on how this method is being used:

A. When used to obtain the name of the tablespace in use by the table, it is called without any arguments.

B. When used to determine whether the given tablespace is the one being used by this table, then getTablespace() takes two parameters:

- The tablespace *`id`*, given as a pointer to a 32-bit unsigned integer
- The tablespace *`version`*, also given as a pointer to a 32-bit unsigned integer

The default value for both *`id`* and *`version`* is `0`.

Return value. The return type depends on how the method is called.

A. When `getTablespace()` is called without any arguments, it returns a `Tablespace` object instance.

B. When called with two arguments, it returns `true` if the tablespace is the same as the one having the ID and version indicated; otherwise, it returns `false`.

2.3.37.38 Table::getTablespaceData()

Description. This method gets the table's tablespace data (ID and version).

Signature.

```
const void* getTablespaceData
    (
      void
    ) const
```

Parameters. *None*.

Return value. A pointer to the data.

2.3.37.39 Table::getTablespaceDataLen()

Description. This method is used to get the length of the table's tablespace data.

Signature.

```
Uint32 getTablespaceDataLen
```

```
    (
      void
    ) const
```

Parameters. *None*.

Return value. The length of the data, as a 32-bit unsigned integer.

2.3.37.40 Table::getTablespaceNames()

Description. This method gets a pointer to the names of the tablespaces used in the table fragments.

Signature.

```
const void* getTablespaceNames
    (
      void
    )
```

Parameters. *None*.

Return value. Returns a pointer to the tablespace name data.

2.3.37.41 Table::getTablespaceNamesLen()

Description. This method gets the length of the tablespace name data returned by `getTablespaceNames()`. (See Section 2.3.37.40, “Table::getTablespaceNames()”.)

Signature.

```
Uint32 getTablespaceNamesLen
    (
      void
    ) const
```

Parameters. *None*.

Return value. Returns the length of the name data, in bytes, as a 32-but unsigned integer.

2.3.37.42 Table::hasDefaultValues()

Description. Used to determine whether the table has any columns that are defined with non-`NULL` default values.

To read and write default column values, use `Column::getDefaultValue()` and `Column::setDefaultValue()`.

Signature.

```
bool hasDefaultValues
    (
      void
    ) const
```

Parameters. *None*.

Return value. Returns `true` if the table has any non-`NULL` columns with default values, otherwise `false`.

2.3.37.43 Table::setDefaultNoPartitionsFlag()

Description. This method sets an indicator that determines whether the default number of partitions is used for the table.

Signature.

```
void setDefaultNoPartitionsFlag
    (
      Uint32 indicator
    ) const
```

Parameters. This method takes a single argument *indicator*, a 32-bit unsigned integer.

Return value. *None*.

2.3.37.44 Table::setFragmentCount()

Description. Sets the number of table fragments.

Signature.

```
void setFragmentCount
    (
      Uint32 count
    )
```

Parameters. *count* is the number of fragments to be used for the table.

Return value. *None*.

2.3.37.45 Table::setFragmentData()

Description. This method writes an array containing the following fragment information:

- Fragment ID
- Node group ID
- Fragment State

Signature.

```
void setFragmentData
    (
      const void* data,
      Uint32      len
    )
```

Parameters. This method takes the following two parameters:

- A pointer to the fragment *data* to be written
- The length (*len*) of this data, in bytes, as a 32-bit unsigned integer

Return value. *None*.

2.3.37.46 Table::setFragmentType()

Description. This method sets the table's fragmentation type.

Signature.

```
void setFragmentType
    (
      FragmentType fragmentType
    )
```

Parameters. This method takes one argument, a `FragmentType` value. See Section 2.3.31.1, “Object::FragmentType”, for more information.

Return value. *None*.

2.3.37.47 Table::setFrm()

Description. This method is used to write data to this table's `.FRM` file.

Signature.

```
void setFrm
    (
      const void* data,
      Uint32      len
    )
```

Parameters. This method takes the following two arguments:

- A pointer to the *`data`* to be written.
- The length (*`len`*) of the data.

Return value. *None*.

2.3.37.48 Table::setHashMap()

Description. Set a hash map for the table. Introduced in NDB 7.2.7.

Signature.

```
int setHashMap
    (
      const class HashMap &
    )
```

Parameters. A reference to the hash map.

Return value. Returns 0 on success; on failure, returns -1 and sets error.

2.3.37.49 Table::setKValue()

Description. This sets the `KValue`, a hashing parameter.

Signature.

```
void setKValue
    (
      int kValue
    )
```

Parameters. *`kValue`* is an integer. Currently the only permitted value is `6`. In a future version this may become a variable parameter.

Return value. *None*.

2.3.37.50 Table::setLinearFlag()

Description.

Signature.

```
void setLinearFlag
    (
      Uint32 flag
    )
```

Parameters. The *`flag`* is a 32-bit unsigned integer.

Return value. *None*.

2.3.37.51 Table::setLogging()

Description. Toggles the table's logging state. See Section 2.3.37.17, “Table::getLogging()”.

Signature.

```
void setLogging
    (
      bool enable
    )
```

Parameters. If *`enable`* is `true`, then logging for this table is enabled; if it is `false`, then logging is disabled.

Return value. *None*.

2.3.37.52 Table::setMaxLoadFactor()

Description. This method sets the maximum load factor when splitting the containers in the local hash tables.

Signature.

```
void setMaxLoadFactor
    (
      int max
    )
```

Parameters. This method takes a single parameter *`max`*, an integer representation of a percentage (for example, `45` represents 45 percent). For more information, see Section 2.3.37.18, “Table::getMaxLoadFactor()”.

Caution

This should never be greater than the minimum load factor.

Return value. *None*.

2.3.37.53 Table::setMaxRows()

Description. This method sets the maximum number of rows that can be held by the table.

Signature.

```
void setMaxRows
    (
      Uint64 maxRows
    )
```

Parameters. *maxRows* is a 64-bit unsigned integer that represents the maximum number of rows to be held in the table.

Return value. *None*.

2.3.37.54 Table::setMinLoadFactor()

Description. This method sets the minimum load factor when reduction of the hash table begins.

Signature.

```
void setMinLoadFactor
    (
      int min
    )
```

Parameters. This method takes a single parameter *min*, an integer representation of a percentage (for example, 45 represents 45 percent). For more information, see Section 2.3.37.20, "Table::getMinLoadFactor()".

Return value. *None*.

2.3.37.55 Table::setName()

Description. This method sets the name of the table.

Note

This is the only `set*()` method of `Table` whose effects are visible to MySQL.

Signature.

```
void setName
    (
      const char* name
    )
```

Parameters. *name* is the (new) name of the table.

Return value. *None*.

2.3.37.56 Table::setObjectType()

Description. This method did not work as intended, and was removed in NDB 7.5.0 (Bug #47960, Bug #11756088).

Signature.

```
void setObjectType
    (
      Object::Type type
    )
```

Parameters. The desired object *type*. This must be one of the `Type` values listed in Section 2.3.31.6, "Object::Type".

Return value. *None*.

2.3.37.57 Table::setPartitionBalance()

Description. Sets the table's partition balancing scheme.

Signature.

```
void setPartitionBalance
    (
      Object::PartitionBalance scheme
    )
```

Parameters. *scheme* is the partition balancing scheme to be used for the table. This is a value of type `PartitionBalance`.

Return value. *None*.

Prior to NDB 7.5.4, this method was known as `setFragmentCountType()`.

2.3.37.58 Table::setRangeListData()

Description. This method sets an array containing information that maps range values and list values to fragments. This is essentially a sorted map consisting of fragment-ID/value pairs. For range partitions there is one pair per fragment. For list partitions it could be any number of pairs, but at least as many pairs as there are fragments.

Signature.

```
void setRangeListData
    (
      const void* data,
      Uint32      len
    )
```

Parameters. This method requires the following two parameters:

- A pointer to the range or list *data* containing the ID/value pairs
- The length (*len*) of this data, as a 32-bit unsigned integer.

Return value. *None*.

2.3.37.59 Table::setRowChecksumIndicator()

Description. *Set the row checksum indicator.*

Signature.

```
void setRowChecksumIndicator
    (
      bool value
    ) const
```

Parameters. A `true`/`false` *value*.

Return value. *None*.

2.3.37.60 Table::setRowGCIIndicator()

Description. *Sets the row GCI indicator.*

Signature.

```
void setRowGCIIndicator
    (
      bool value
    ) const
```

Parameters. A `true`/`false` *`value`*.

Return value. *None*.

2.3.37.61 Table::setStatusInvalid()

Description. Forces the table's status to be invalidated.

Signature.

```
void setStatusInvalid
    (
      void
    ) const
```

Parameters. *None*.

Return value. *None*.

2.3.37.62 Table::setTablespace()

Description. This method sets the tablespace for the table.

Signatures. Using the name of the tablespace:

```
void setTablespace
    (
      const char* name
    )
```

Using a `Tablespace` object:

```
void setTablespace
    (
      const class Tablespace& tablespace
    )
```

Parameters. This method can be called with a single argument, which can be of either one of these two types:

1. The *`name`* of the tablespace (a string).
2. A reference to an existing `Tablespace` instance.

See Section 2.3.38, “The Tablespace Class”.

Return value. *None*.

2.3.37.63 Table::setTablespaceData()

Description. This method sets the tablespace information for each fragment, and includes a tablespace ID and a tablespace version.

Signature.

```
void setTablespaceData
    (
      const void* data,
      Uint32      len
    )
```

Parameters. This method requires the following two parameters:

- A pointer to the *data* containing the tablespace ID and version
- The length (*len*) of this data, as a 32-bit unsigned integer.

Return value. *None*.

2.3.37.64 Table::setTablespaceNames()

Description. Sets the names of the tablespaces used by the table fragments.

Signature.

```
void setTablespaceNames
    (
      const void* data
      Uint32      len
    )
```

Parameters. This method takes the following two parameters:

- A pointer to the tablespace names *data*
- The length (*len*) of the names data, as a 32-bit unsigned integer.

Return value. *None*.

2.3.37.65 Table::SingleUserMode

Description. Single user mode specifies access rights to the table when single user mode is in effect.

Enumeration values. Possible values for `SingleUserMode` are shown, along with descriptions, in the following table:

Value	Description
`SingleUserModeLocked`	The table is locked (unavailable).
`SingleUserModeReadOnly`	The table is available in read-only mode.
`SingleUserModeReadWrite`	The table is available in read-write mode.

2.3.37.66 Table::validate()

Description. This method validates the definition for a new table prior to its being created, and executes the `Table::aggregate()` method, as well as performing additional checks. `validate()` is called automatically when a table is created or retrieved. For this reason, it is usually not necessary to call `aggregate()` or `validate()` directly.

Warning

Even after the `validate()` method is called, there may still exist errors which can be detected only by the `NDB` kernel when the table is actually created.

Note

This method was added in MySQL 5.1.12 (see Bug #21690).

Signature.

```
int validate
    (
```

```
    struct NdbError& error
  )
```

Parameters. A reference to an `NdbError` object.

Return value. An integer, whose value is `0` on success, and `-1` if the table is in an inconsistent state. In the latter case, the *`error`* is also set.

2.3.38 The Tablespace Class

This section discusses the `Tablespace` class and its public members.

Parent class. `NdbDictionary`

Child classes. *None*

Description. The `Tablespace` class models an NDB Cluster Disk Data tablespace, which contains the datafiles used to store Cluster Disk Data. For an overview of Cluster Disk Data and their characteristics, see CREATE TABLESPACE Syntax, in the MySQL Manual.

Note

Currently, only unindexed column data can be stored on disk. Indexes and indexes columns are always stored in memory.

NDB Cluster prior to MySQL 5.1 does not support Disk Data storage, and so does not support tablespaces; thus the `Tablespace` class is unavailable for NDB API applications written against these older releases.

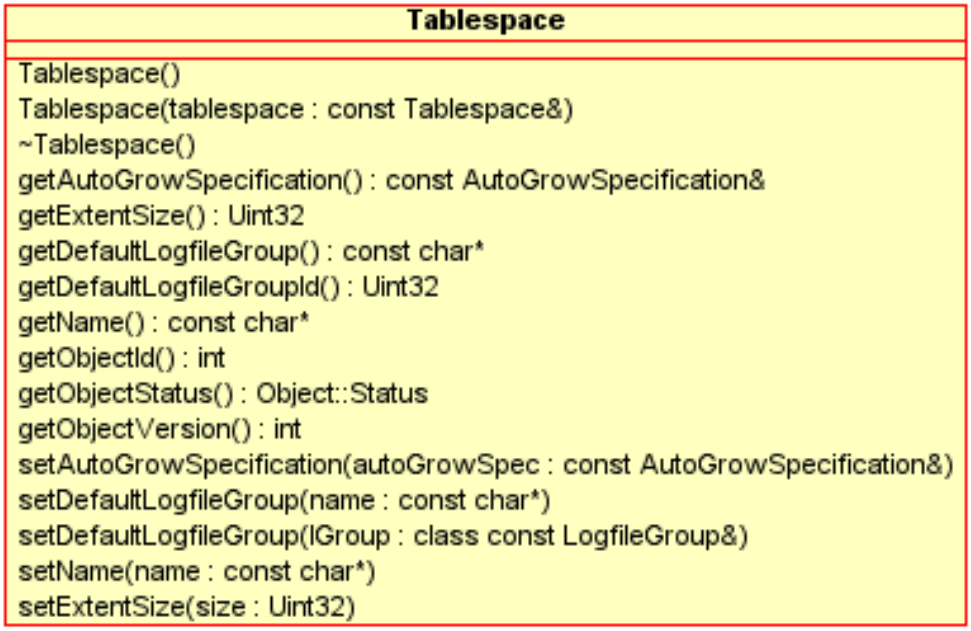
Methods. The following table lists the public methods of this class and the purpose or use of each method:

Method	Purpose / Use
`Tablespace()`	Class constructor
`~Tablespace()`	Virtual destructor method
`getAutoGrowSpecification()`	Used to obtain the `AutoGrowSpecification` structure associated with the tablespace
`getDefaultLogfileGroup()`	Gets the name of the tablespace's default log file group
`getDefaultLogfileGroupId()`	Gets the ID of the tablespace's default log file group
`getExtentSize()`	Gets the extent size used by the tablespace
`getName()`	Gets the name of the tablespace
`getObjectId()`	Gets the object ID of a `Tablespace` instance
`getObjectStatus()`	Used to obtain the `Object::Status` of the `Tablespace` instance for which it is called
`getObjectVersion()`	Gets the object version of the `Tablespace` object for which it is invoked
`setAutoGrowSpecification()`	Used to set the auto-grow characteristics of the tablespace
`setDefaultLogfileGroup()`	Sets the tablespace's default log file group
`setExtentSize()`	Sets the size of the extents used by the tablespace
`setName()`	Sets the name for the tablespace

Types. The Tablespace class defines no public types of its own; however, two of its methods make use of the `AutoGrowSpecification` data structure.

Class diagram. This diagram shows all the available methods and enumerated types of the `Tablespace` class:

Figure 2.32 NdbDictionary::Object::Tablespace



2.3.38.1 Tablespace Constructor

Description. These methods are used to create a new instance of `Tablespace`, or to copy an existing one.

Note

The `Dictionary` class also supplies methods for creating and dropping tablespaces.

Signatures. New instance:

```
Tablespace
    (
      void
    )
```

Copy constructor:

```
Tablespace
    (
      const Tablespace& tablespace
    )
```

Parameters. New instance: *None*. Copy constructor: a reference to an existing `Tablespace` instance.

Return value. A `Tablespace` object.

Destructor. The class defines a virtual destructor `~Tablespace()` which takes no arguments and returns no value.

2.3.38.2 Tablespace::getAutoGrowSpecification()

Description.

Signature.

```
const AutoGrowSpecification& getAutoGrowSpecification
    (
      void
    ) const
```

Parameters. *None*.

Return value. A reference to the structure which describes the tablespace auto-grow characteristics; for details, see Section 2.3.1, “The AutoGrowSpecification Structure”.

2.3.38.3 Tablespace::getDefaultLogfileGroup()

Description. This method retrieves the name of the tablespace's default log file group.

Note

Alternatively, you may wish to obtain the ID of the default log file group; see Section 2.3.38.4, “Tablespace::getDefaultLogfileGroupId()”.

Signature.

```
const char* getDefaultLogfileGroup
    (
      void
    ) const
```

Parameters. *None*.

Return value. The name of the log file group (string value as character pointer).

2.3.38.4 Tablespace::getDefaultLogfileGroupId()

Description. This method retrieves the ID of the tablespace's default log file group.

Note

You can also obtain directly the name of the default log file group rather than its ID; see Section 2.3.38.3, “Tablespace::getDefaultLogfileGroup()”, for more information.

Signature.

```
Uint32 getDefaultLogfileGroupId
    (
      void
    ) const
```

Parameters. *None*.

Return value. The ID of the log file group, as an unsigned 32-bit integer.

2.3.38.5 Tablespace::getExtentSize()

Description. This method is used to retrieve the *extent size*—that is the size of the memory allocation units—used by the tablespace.

Note

The same extent size is used for all datafiles contained in a given tablespace.

Signature.

```
Uint32 getExtentSize
    (
      void
    ) const
```

Parameters. *None*.

Return value. The tablespace's extent size in bytes, as an unsigned 32-bit integer.

2.3.38.6 Tablespace::getObjectId()

Description. This method retrieves the tablespace's object ID.

Signature.

```
virtual int getObjectId
    (
      void
    ) const
```

Parameters. *None*.

Return value. The object ID, as an integer.

2.3.38.7 Tablespace::getName()

Description. This method retrieves the name of the tablespace.

Signature.

```
const char* getName
    (
      void
    ) const
```

Parameters. *None*.

Return value. The name of the tablespace, a string value (as a character pointer).

2.3.38.8 Tablespace::getObjectStatus()

Description. This method is used to retrieve the object status of a tablespace.

Signature.

```
virtual Object::Status getObjectStatus
    (
      void
    ) const
```

Parameters. *None*.

Return value. An `Object::Status` value.

2.3.38.9 Tablespace::getObjectVersion()

Description. This method gets the tablespace object version (see NDB Schema Object Versions).

Signature.

```
virtual int getObjectVersion
  (
    void
  ) const
```

Parameters. *None*.

Return value. The object version, as an integer.

2.3.38.10 Tablespace::setAutoGrowSpecification()

Description. This method is used to set the auto-grow characteristics of the tablespace.

Signature.

```
void setAutoGrowSpecification
    (
      const AutoGrowSpecification& autoGrowSpec
    )
```

Parameters. This method takes a single parameter, an `AutoGrowSpecification` data structure.

Return value. *None*.

2.3.38.11 Tablespace::setDefaultLogfileGroup()

Description. This method is used to set a tablespace's default log file group.

Signature. This method can be called in two different ways. The first of these uses the name of the log file group, as shown here:

```
void setDefaultLogfileGroup
    (
      const char* name
    )
```

This method can also be called by passing it a reference to a `LogfileGroup` object:

```
void setDefaultLogfileGroup
    (
      const class LogfileGroup& lGroup
    )
```

Note

There is no method for setting a log file group as the default for a tablespace by referencing the log file group's ID. (In other words, there is no `set*()` method corresponding to `getDefaultLogfileGroupId()`.)

Parameters. Either the *`name`* of the log file group to be assigned to the tablespace, or a reference *`lGroup`* to this log file group.

Return value. *None*.

2.3.38.12 Tablespace::setExtentSize()

Description. This method sets the tablespace's extent size.

Signature.

```
void setExtentSize
    (
      Uint32 size
    )
```

Parameters. The *`size`* to be used for this tablespace's extents, in bytes.

Return value. *None*.

2.3.38.13 Tablespace::setName()

Description. This method sets the name of the tablespace.

Signature.

```
void setName
    (
      const char* name
    ) const
```

Parameters. The *`name`* of the tablespace, a string (character pointer).

Return value. *None*.

2.3.39 The Undofile Class

The section discusses the `Undofile` class and its public methods.

Parent class. `NdbDictionary`

Child classes. *None*

Description. The `Undofile` class models an NDB Cluster Disk Data undofile, which stores data used for rolling back transactions.

Note

Currently, only unindexed column data can be stored on disk. Indexes and indexes columns are always stored in memory.

NDB Cluster prior to MySQL 5.1 does not support Disk Data storage, and so does not support undo files; thus the `Undofile` class is unavailable for NDB API applications written against these older releases.

Methods. The following table lists the public methods of this class and the purpose or use of each method:

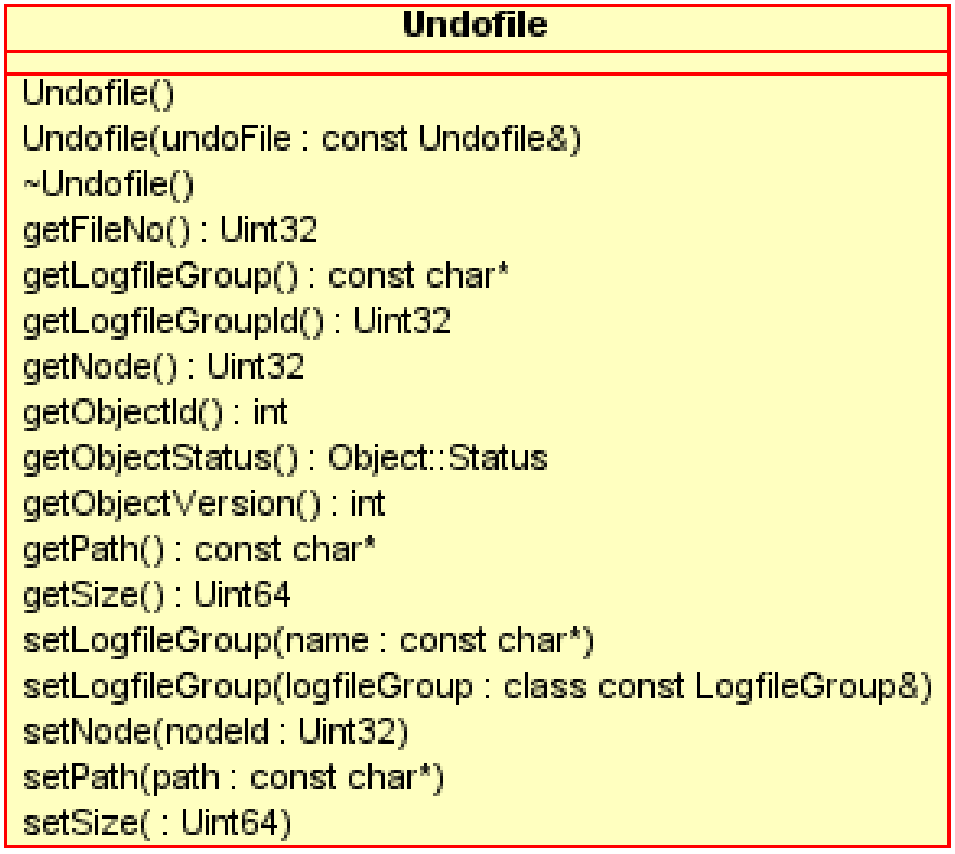
Method	Purpose / Use
`Undofile()`	Class constructor
`~Undofile()`	Virtual destructor
`getFileNo()`	Removed in NDB 7.5.0 (Bug #47960, Bug #11756088)
`getLogfileGroup()`	Gets the name of the log file group to which the undo file belongs
`getLogfileGroupId()`	Gets the ID of the log file group to which the undo file belongs
`getNode()`	Removed in NDB 7.5.0 (Bug #47960, Bug #11756088)
`getObjectId()`	Gets the undo file's object ID

Method	Purpose / Use
`getObjectStatus()`	Gets the undo file's `Status`
`getObjectVersion()`	Gets the undo file's object version
`getPath()`	Gets the undo file's file system path
`getSize()`	Gets the size of the undo file
`setLogfileGroup()`	Sets the undo file's log file group using the name of the log file group or a reference to the corresponding `LogfileGroup` object
`setNode()`	Removed in NDB 7.5.0 (Bug #47960, Bug #11756088)
`setPath()`	Sets the file system path for the undo file
`setSize()`	Sets the undo file's size

Types. The `Undofile` class defines no public types.

Class diagram. This diagram shows all the available methods of the `Undofile` class:

Figure 2.33 NdbDictionary::Object::Undofile



2.3.39.1 Undofile Constructor

Description. The class constructor can be used to create a new `Undofile` instance, or to copy an existing one.

Signatures. Creates a new instance:

```
Undofile
    (
      void
```

```
    )
```

Copy constructor:

```
Undofile
    (
      const Undofile& undoFile
    )
```

Parameters. New instance: *None*. The copy constructor takes a single argument—a reference to the `Undofile` object to be copied.

Return value. An `Undofile` object.

Destructor. The class defines a virtual destructor which takes no arguments and has the return type `void`.

2.3.39.2 Undofile::getFileNo()

Description. This method did not work as intended, and was removed in NDB 7.5.0 (Bug #47960, Bug #11756088).

Signature.

```
Uint32 getFileNo
    (
      void
    ) const
```

Parameters. *None*.

Return value. The number of the undo file, as an unsigned 32-bit integer.

2.3.39.3 Undofile::getLogfileGroup()

Description. This method retrieves the name of the log file group to which the undo file belongs.

Signature.

```
const char* getLogfileGroup
    (
      void
    ) const
```

Parameters. *None*.

Return value. The name of the log file group, a string value (as a character pointer).

2.3.39.4 Undofile::getLogfileGroupId()

Description. This method retrieves the ID of the log file group to which the undo file belongs.

Note

It is also possible to obtain the name of the log file group directly. See Section 2.3.39.3, “Undofile::getLogfileGroup()”

Signature.

```
Uint32 getLogfileGroupId
    (
```

```
      void
    ) const
```

Parameters. *None*.

Return value. The ID of the log file group, as an unsigned 32-bit integer.

2.3.39.5 Undofile::getNode()

Description. This method did not work as intended, and was removed in NDB 7.5.0 (Bug #47960, Bug #11756088).

Signature.

```
Uint32 getNode
    (
      void
    ) const
```

Parameters. *None*.

Return value. The node ID, as an unsigned 32-bit integer.

2.3.39.6 Undofile::getObjectId()

Description. This method retrieves the undo file's object ID.

Signature.

```
virtual int getObjectId
    (
      void
    ) const
```

Parameters. *None*.

Return value. The object ID, as an integer.

2.3.39.7 Undofile::getObjectStatus()

Description. This method is used to retrieve the object status of an undo file.

Signature.

```
virtual Object::Status getObjectStatus
    (
      void
    ) const
```

Parameters. *None*.

Return value. An `Object::Status` value.

2.3.39.8 Undofile::getObjectVersion()

Description. This method gets the undo file's object version (see NDB Schema Object Versions).

Signature.

```
virtual int getObjectVersion
  (
```

```
    void
  ) const
```

Parameters. *None*.

Return value. The object version, as an integer.

2.3.39.9 Undofile::getPath()

Description. This method retrieves the path matching the location of the undo file on the data node's file system.

Signature.

```
const char* getPath
    (
      void
    ) const
```

Parameters. *None*.

Return value. The file system path, a string (as a character pointer).

2.3.39.10 Undofile::getSize()

Description. This method gets the size of the undo file in bytes.

Signature.

```
Uint64 getSize
    (
      void
    ) const
```

Parameters. *None*.

Return value. The size in bytes of the undo file, as an unsigned 64-bit integer.

2.3.39.11 Undofile::setLogfileGroup()

Description. Given either a name or an object reference to a log file group, the `setLogfileGroup()` method assigns the undo file to that log file group.

Signature. Using a log file group name:

```
void setLogfileGroup
    (
      const char* name
    )
```

Using a reference to an instance of `LogfileGroup`:

```
void setLogfileGroup
    (
      const class LogfileGroup & logfileGroup
    )
```

Parameters. The *`name`* of the log file group (a character pointer), or a reference to a `LogfileGroup` instance.

Return value. *None*.

2.3.39.12 Undofile::setNode()

Description. This method did not work as intended, and was removed in NDB 7.5.0 (Bug #47960, Bug #11756088).

Signature.

```
void setNode
    (
      Uint32 nodeId
    )
```

Parameters. The *`nodeId`* of the data node where the undo file is to be placed; this is an unsigned 32-bit integer.

Return value. *None*.

2.3.39.13 Undofile::setPath()

Description. This method is used to set the file system path of the undo file on the data node where it resides.

Signature.

```
void setPath
    (
      const char* path
    )
```

Parameters. The desired *`path`* to the undo file.

Return value. *None*.

2.3.39.14 Undofile::setSize()

Description. Sets the size of the undo file in bytes.

Signature.

```
void setSize
    (
      Uint64 size
    )
```

Parameters. The intended *`size`* of the undo file in bytes, as an unsigned 64-bit integer.

Return value. *None*.

2.4 NDB API Errors and Error Handling

This section contains a discussion of error handling in NDB API applications as well as listing listings of the most common `NDB` error codes and messages, along with their classifications and likely causes for which they might be raised.

For information about the `NdbError` structure, which is used to convey error information to NDB API applications, see Section 2.3.20, “The NdbError Structure”.

Important

It is strongly recommended that you *not* depend on specific error codes in your `NDB` API applications, as they are subject to change over time. Instead,

you should use the `NdbError::Status` and error classification in your source code, or consult the output of `perror --ndb error_code` to obtain information about a specific error code.

If you find a situation in which you need to use a specific error code in your application, please file a bug report at http://bugs.mysql.com/ so that we can update the corresponding status and classification.

2.4.1 Handling NDB API Errors

This section describes how NDB API errors can be detected and mapped onto particular operations.

NDB API errors can be generated in either of two ways:

- When an operation is defined
- When an operation is executed

Errors raised during operation definition. Errors generated during operation definition result in a failure return code from the method called. The actual error can be determined by examining the relevant `NdbOperation` object, or the operation's `NdbTransaction` object.

Errors raised during operation execution. Errors occurring during operation execution cause the transaction of which they are a part to be aborted unless the `AO_IgnoreError` abort option is set for the operation.

By default, read operations are run with `AO_IgnoreError`, and write operations are run with `AbortOnError`, but this can be overridden by the user. When an error during execution causes a transaction to be aborted, the `execute()` method returns a failure return code. If an error is ignored due to `AO_IgnoreError` being set on the operation, the `execute()` method returns a success code, and the user must examine all operations for failure using `NdbOperation::getNdbError()`. For this reason, the return value of `getNdbError()` should usually be checked, even if `execute()` returns success. If the client application does not keep track of `NdbOperation` objects during execution, then `NdbTransaction::getNextCompletedOperation()` can be used to iterate over them.

You should also be aware that use of `NdbBlob` can result in extra operations being added to the batches executed. This means that, when iterating over completed operations using `getNextCompletedOperation()`, you may encounter operations related to `NdbBlob` objects which were not defined by your application.

Note

A read whose `LockMode` is `CommittedRead` cannot be `AbortOnError`. In this case, it is always be `IgnoreError`.

In all cases where operation-specific errors arise, an execution error with an operation is marked against both the operation and the associated transaction object. Where there are multiple operation errors in a single `NdbTransaction::execute()` call, due to operation batching and the use of `AO_IgnoreError`, only the first is marked against the `NdbTransaction` object. The remaining errors are recorded against the corresponding `NdbOperation` objects only.

It is also possible for errors to occur during execution—such as a data node failure—which are marked against the transaction object, but *not* against the underlying operation objects. This is because these errors apply to the transaction as a whole, and not to individual operations within the transaction.

For this reason, applications should use `NdbTransaction::getNdbError()` as the first way to determine whether an `NdbTransaction::execute()` call failed. If the batch of operations being executed included operations with the `AO_IgnoreError` abort option set, then it is possible that there

were multiple failures, and the completed operations should be checked individually for errors using `NdbOperation::getNdbError()`.

Implicit NdbTransaction::execute() calls in scan and BLOB methods. Scan operations are executed in the same way as other operations, and also have implicit `execute()` calls within the `NdbScanOperation::nextResult()` method. When `NdbScanOperation::nextResult()` indicates failure (that is, if the method returns `-1`), the transaction object should be checked for an error. The `NdbScanOperation` may also contain the error, but only if the error is not operation-specific.

Some `BLOB` manipulation methods also have implicit internal `execute()` calls, and so can experience operation execution failures at these points. The following `NdbBlob` methods can generate implicit `execute()` calls; this means that they also require checks of the `NdbTransaction` object for errors via `NdbTransaction::getNdbError()` if they return an error code:

- `setNull()`
- `truncate()`
- `readData()`
- `writeData()`

Summary. In general, it is possible for an error to occur during execution (resulting in a failure return code) when calling any of the following methods:

- `NdbTransaction::execute()`
- `NdbBlob::setNull()`
- `NdbBlob::truncate()`
- `NdbBlob::readData()`
- `NdbBlob::writeData()`
- `NdbScanOperation::nextResult()`

Note

This method does *not* perform an implicit `execute()` call. The `NdbBlob` methods can cause other defined operations to be executed when these methods are called; however, `nextResult()` calls do not do so.

If this happens, the `NdbTransaction::getNdbError()` method should be called to identify the first error that occurred. When operations are batched, and there are `IgnoreError` operations in the batch, there may be multiple operations with errors in the transaction. These can be found by using `NdbTransaction::getNextCompletedOperation()` to iterate over the set of completed operations, calling `NdbOperation::getNdbError()` for each operation.

When `IgnoreError` has been set on any operations in a batch of operations to be executed, the `NdbTransaction::execute()` method indicates success even where errors have actually occurred, as long as none of these errors caused a transaction to be aborted. To determine whether there were any ignored errors, the transaction error status should be checked using `NdbTransaction::getNdbError()`. *Only if this indicates success can you be certain that no errors occurred*. If an error code is returned by this method, and operations were batched, then you should iterate over all completed operations to find all the operations with ignored errors.

Example (*pseudocode*). We begin by executing a transaction which may have batched operations and a mix of `AO_IgnoreError` and `AbortOnError` abort options:

```
int execResult= NdbTransaction.execute(args);
```

Note

For the number and permitted values of *`args`*, see Section 2.3.30.6, “NdbTransaction::execute()”.

Next, because errors on `AO_IgnoreError` operations do not affect execResult—that is, the value returned by `execute()`—we check for errors on the transaction:

```
NdbError err= NdbTransaction.getNdbError();

if (err.code != 0)
{
```

An nonzero value for the error code means that an error was raised on the transaction. This could be due to any of the following conditions:

- A transaction-wide error, such as a data node failure, that caused the transaction to be aborted
- A single operation-specific error, such as a constraint violation, that caused the transaction to be aborted
- A single operation-specific ignored error, such as no data found, that did not cause the transaction to be aborted
- The first of many operation-specific ignored errors, such as no data found when batching, that did not cause the transaction to be aborted
- First of a number of operation-specific ignored errors such as no data found (when batching) before an aborting operation error (transaction aborted)

```
  if (execResult != 0)
  {
```

The transaction has been aborted. The recommended strategy for handling the error in this case is to test the transaction error status and take appropriate action based on its value:

```
    switch (err.status)
    {
      case value1:
        //  statement block handling value1 ...
      case value2:
        //  statement block handling value2 ...
        //  (etc. ...)
      case valueN:
        //  statement block handling valueN ...
    }
```

Since the transaction was aborted, it is generally necessary to iterate over the completed operations (if any) and find the errors raised by each only if you wish to do so for reporting purposes.

```
  }
  else
  {
```

The transaction itself was not aborted, but there must be one or more ignored errors. In this case, you should iterate over the operations to determine what happened and handle the cause accordingly.

```
```

```
   }
}
```

To handle a `NdbScanOperation::nextResult()` which returns `-1`, indicating that the operation failed (omitting cases where the operation was successful):

```
int nextrc= NdbScanOperation.nextResult(args);
```

Note

For the number and permitted values of *`args`*, see Section 2.3.29.6, "NdbScanOperation::nextResult()".

```
if (nextrc == -1)
{
```

First, you should check the `NdbScanOperation` object for any errors:

```
  NdbError err= NdbScanOperation.getNdbError();

  if (err.code == 0)
  {
```

No error was found in the scan operation; the error must belong to the transaction as whole.

```
  }
    err= NdbTransaction.getNdbError();
```

Now you can handle the error based on the error status:

```
    switch (err.status)
    {
      case value1:
        //  statement block handling value1 ...
      case value2:
        //  statement block handling value2 ...
        //  (etc. ...)
      case valueN:
        //  statement block handling valueN ...
    }
}
```

For information about NDB API error classification and status codes, see Section 2.4.4, "NDB Error Classifications". While you should not rely on a specific error code or message text in your NDB API applications—since error codes and messages are both subject to change over time—it can be useful to check error codes and messages to help determine why a particular failure occurred. For more information about these, see Section 2.4.2, "NDB Error Codes: by Type". For more about `NdbError` and the types of information which can be obtained from `NdbError` objects, see Section 2.3.20, "The NdbError Structure".

2.4.2 NDB Error Codes: by Type

This section contains a number of tables, one for each type of NDB API error. The error types include those in the following list:

- No error
- Application error

- Scan application error
- Configuration or application error (currently unused)
- No data found
- Constraint violation
- Schema error
- User defined error
- Insufficient space
- Temporary Resource error
- Node Recovery error
- Overload error
- Timeout expired
- Node shutdown
- Internal temporary
- Unknown result error
- Unknown error code (currently unused)
- Internal error
- Function not implemented

The information in each table includes, for each error:

- The numeric NDB error code
- The corresponding MySQL error code
- The NDB classification code

  See Section 2.4.4, “NDB Error Classifications”, for the meanings of these classification codes.

- The text of the error message

Similar errors have been grouped together in each table.

Note

You can always obtain the latest error codes and information from the file `storage/ndb/src/ndbapi/ndberror.c`.

2.4.2.1 No error Errors

The following table lists and describes `NDB` **`No error`** errors (error type `NE`).

Table 2.1 This table describes NDB API No error errors

NDB Error Code	MySQL Error Code	NDB Error Type	Error Message
0	0	NE	No error

2.4.2.2 Application error Errors

The following table lists and describes NDB **Application error** errors (error type AE).

Table 2.2 This table describes NDB API Application error errors

NDB Error Code	MySQL Error Code	NDB Error Type	Error Message
QRY_REQ_ARG	4271	AE	Required argument is NULL
QRY_TOO_FEW_KEY_VALUES	4272	AE	All required 'key' values was not specified
QRY_TOO_MANY_KEY_VALUES	4273	AE	Too many 'key' or 'bound' values was specified
QRY_OPERAND_HAS_WRONG_TYPE	4274	AE	Incompatible datatype specified in operand argument
QRY_CHAR_OPERAND_TRUNCATED	4275	AE	Character operand was right truncated
QRY_NUM_OPERAND_RANGE	4276	AE	Numeric operand out of range
QRY_MULTIPLE_PARENTS	4277	AE	Multiple 'parents' specified in linkedValues for this operation
QRY_UNKNOWN_PARENT	4278	AE	Unknown 'parent' specified in linkedValue
QRY_UNRELATED_INDEX	4279	AE	Specified 'index' does not belong to specified 'table'
QRY_WRONG_INDEX_TYPE	4280	AE	Wrong type of index specified for this operation
QRY_OPERAND_ALREADY_BOUND	4281	AE	Can't use same operand value to specify different column values
QRY_DEFINITION_TOO_LARGE	4282	AE	Query definition too large.
QRY_RESULT_ROW_ALREADY_DEFINED	4283	AE	Result row already defined for NdbQueryOperation.
QRY_HAS_ZERO_OPERATIONS	4284	AE	Query defintion should have at least one operation.
QRY_IN_ERROR_STATE	4285	AE	A previous query operation failed, which you missed to catch.
QRY_ILLEGAL_STATE	4286	AE	Query is in illegal state for this operation.
QRY_WRONG_OPERATION_TYPE	4287	AE	This method cannot be invoked on this type of operation (lookup/scan/index scan).
QRY_SCAN_ORDER_ALREADY_SET	4288	AE	Index scan order was already set in query definition.
QRY_PARAMETER_HAS_WRONG_TYPE	4289	AE	Parameter value has an incompatible datatype
QRY_CHAR_PARAMETER_TRUNCATED	4290	AE	Character Parameter was right truncated
QRY_MULTIPLE_SCAN_SORTED	4291	AE	Query with multiple scans may not be sorted.
QRY_SEQUENTIAL_SCAN_SORTED	4292	AE	Parallelism cannot be restricted for sorted scans.

NDB Error Code	MySQL Error Code	NDB Error Type	Error Message
QRY_BATCH_SIZE_TOO_SMALL	DMEC	AE	Batch size for sub scan cannot be smaller than number of fragments.
QRY_EMPTY_PROJECTION	DMEC	AE	Query has operation with empty projection.
NO_CONTACT_WITH_PROCESS	DMEC	AE	No contact with the process (dead ?).
WRONG_PROCESS_TYPE	DMEC	AE	The process has wrong type. Expected a DB process.
SEND_OR_RECEIVE_FAILED	DMEC	AE	Send to process or receive failed.
INVALID_ERROR_NUMBER	DMEC	AE	Invalid error number. Should be >= 0.
INVALID_TRACE_NUMBER	DMEC	AE	Invalid trace number.
INVALID_BLOCK_NAME	DMEC	AE	Invalid block name
NODE_SHUTDOWN_IN_PROGESS	DMEC	AE	Node shutdown in progress
SYSTEM_SHUTDOWN_IN_PROGRESS	DMEC	AE	System shutdown in progress
NODE_SHUTDOWN_WOULD_CAUSE_SYSTEM_CRASH	DMEC	AE	Node shutdown would cause system crash
UNSUPPORTED_MULTI_NODE_SHUTDOWN	DMEC	AE	Unsupported multi node shutdown. Abort option required.
NODE_NOT_API_NODE	DMEC	AE	The specified node is not an API node.
OPERATION_NOT_ALLOWED_START_STOP	DMEC	AE	Operation not allowed while nodes are starting or stopping.
NO_CONTACT_WITH_DB_NODES	DMEC	AE	No contact with database nodes }
242	DMEC	AE	Zero concurrency in scan
244	DMEC	AE	Too high concurrency in scan
261	DMEC	AE	DML count in transaction exceeds config parameter MaxDMLOperationsPerTransaction
269	DMEC	AE	No condition and attributes to read in scan
281	HA_ERR_NO_CONNECTION	AE	Operation not allowed due to cluster shutdown in progress
299	DMEC	AE	Operation not allowed or aborted due to single user mode
311	DMEC	AE	Undefined partition used in setPartitionId
320	DMEC	AE	Invalid no of nodes specified for new nodegroup
321	DMEC	AE	Invalid nodegroup id
322	DMEC	AE	Invalid node(s) specified for new nodegroup, node already in nodegroup
323	DMEC	AE	Invalid nodegroup id, nodegroup already existing
324	DMEC	AE	Invalid node(s) specified for new nodegroup, no node in nodegroup is started

NDB Error Code	MySQL Error Code	NDB Error Type	Error Message
325	DMEC	AE	Invalid node(s) specified for new nodegroup, node ID invalid or undefined
417	DMEC	AE	Bad operation reference - double unlock
720	DMEC	AE	Attribute name reused in table definition
763	DMEC	AE	DDL is not supported with mixed data-node versions
771	HA_WRONG_CREATE_OPTION	AE	Given NODEGROUP doesn't exist in this cluster
776	DMEC	AE	Index created on temporary table must itself be temporary
777	DMEC	AE	Cannot create a temporary index on a non-temporary table
778	DMEC	AE	A temporary table or index must be specified as not logging
789	HA_WRONG_CREATE_OPTION	AE	Logfile group not found
793	DMEC	AE	Object definition too big
794	DMEC	AE	Schema feature requires data node upgrade
798	DMEC	AE	A disk table must not be specified as no logging
823	DMEC	AE	Too much attrinfo from application in tuple manager
829	DMEC	AE	Corrupt data received for insert/update
831	DMEC	AE	Too many nullable/bitfields in table definition
850	DMEC	AE	Too long or too short default value
851	DMEC	AE	Maximum 8052 bytes of FIXED columns supported, use varchar or COLUMN_FORMAT DYNAMIC instead
874	DMEC	AE	Too much attrinfo (e.g. scan filter) for scan in tuple manager
876	DMEC	AE	876
877	DMEC	AE	877
878	DMEC	AE	878
879	DMEC	AE	879
880	DMEC	AE	Tried to read too much - too many getValue calls
884	DMEC	AE	Stack overflow in interpreter
885	DMEC	AE	Stack underflow in interpreter
886	DMEC	AE	More than 65535 instructions executed in interpreter
892	DMEC	AE	Unsupported type in scan filter
897	DMEC	AE	Update attempt of primary key via ndbcluster internal api (if this occurs via

NDB Error Code	MySQL Error Code	NDB Error Type	Error Message
			the MySQL server it is a bug, please report)
913	DMEC	AE	Invalid index for index stats update
920	DMEC	AE	Row operation defined after refreshTuple()
1233	DMEC	AE	Table read-only
1302	DMEC	AE	A backup is already running
1306	DMEC	AE	Backup not supported in diskless mode (change Diskless)
1329	DMEC	AE	Backup during software upgrade not supported
1342	DMEC	AE	Backup failed to allocate buffers (check configuration)
1343	DMEC	AE	Backup failed to setup fs buffers (check configuration)
1344	DMEC	AE	Backup failed to allocate tables (check configuration)
1345	DMEC	AE	Backup failed to insert file header (check configuration)
1346	DMEC	AE	Backup failed to insert table list (check configuration)
1347	DMEC	AE	Backup failed to allocate table memory (check configuration)
1348	DMEC	AE	Backup failed to allocate file record (check configuration)
1349	DMEC	AE	Backup failed to allocate attribute record (check configuration)
1701	DMEC	AE	Node already reserved
1702	DMEC	AE	Node already connected
1704	DMEC	AE	Node type mismatch
4004	DMEC	AE	Attribute name or id not found in the table
4100	DMEC	AE	Status Error in NDB
4101	DMEC	AE	No connections to NDB available and connect failed
4102	DMEC	AE	Type in NdbTamper not correct
4103	DMEC	AE	No schema connections to NDB available and connect failed
4104	DMEC	AE	Ndb Init in wrong state, destroy Ndb object and create a new
4105	DMEC	AE	Too many Ndb objects
4106	DMEC	AE	All Not NULL attribute have not been defined
4114	DMEC	AE	Transaction is already completed

NDB Error Code	MySQL Error Code	NDB Error Type	Error Message
4116	DMEC	AE	Operation was not defined correctly, probably missing a key
4117	DMEC	AE	Could not start transporter, configuration error
4118	DMEC	AE	Parameter error in API call
4120	DMEC	AE	Scan already complete
4121	DMEC	AE	Cannot set name twice for an Ndb object
4122	DMEC	AE	Cannot set name after Ndb object is initialised
4123	DMEC	AE	Free percent out of range. Allowed range is 1-99
4200	DMEC	AE	Status Error when defining an operation
4201	DMEC	AE	Variable Arrays not yet supported
4202	DMEC	AE	Set value on tuple key attribute is not allowed
4203	DMEC	AE	Trying to set a NOT NULL attribute to NULL
4204	DMEC	AE	Set value and Read/Delete Tuple is incompatible
4205	DMEC	AE	No Key attribute used to define tuple
4206	DMEC	AE	Not allowed to equal key attribute twice
4207	DMEC	AE	Key size is limited to 4092 bytes
4208	DMEC	AE	Trying to read a non-stored attribute
4209	DMEC	AE	Length parameter in equal/setValue is incorrect
4210	DMEC	AE	Ndb sent more info than the length he specified
4211	DMEC	AE	Inconsistency in list of NdbRecAttr-objects
4212	DMEC	AE	Ndb reports NULL value on Not NULL attribute
4213	DMEC	AE	Not all data of an attribute has been received
4214	DMEC	AE	Not all attributes have been received
4215	DMEC	AE	More data received than reported in TCKEYCONF message
4216	DMEC	AE	More than 8052 bytes in setValue cannot be handled
4217	DMEC	AE	It is not allowed to increment any other than unsigned ints
4218	DMEC	AE	Currently not allowed to increment NULL-able attributes
4219	DMEC	AE	Maximum size of interpretative attributes are 64 bits

NDB Error Code	MySQL Error Code	NDB Error Type	Error Message
4220	DMEC	AE	Maximum size of interpretative attributes are 64 bits
4221	DMEC	AE	Trying to jump to a non-defined label
4222	DMEC	AE	Label was not found, internal error
4223	DMEC	AE	Not allowed to create jumps to yourself
4224	DMEC	AE	Not allowed to jump to a label in a different subroutine
4225	DMEC	AE	All primary keys defined, call setValue/ getValue
4226	DMEC	AE	Bad number when defining a label
4227	DMEC	AE	Bad number when defining a subroutine
4228	DMEC	AE	Illegal interpreter function in scan definition
4229	DMEC	AE	Illegal register in interpreter function definition
4230	DMEC	AE	Illegal state when calling getValue, probably not a read
4231	DMEC	AE	Illegal state when calling interpreter routine
4232	DMEC	AE	Parallelism can only be between 1 and 240
4233	DMEC	AE	Calling execute (synchronous) when already prepared asynchronous transaction exists
4234	DMEC	AE	Illegal to call setValue in this state
4235	DMEC	AE	No callback from execute
4236	DMEC	AE	Trigger name too long
4237	DMEC	AE	Too many triggers
4238	DMEC	AE	Trigger not found
4239	DMEC	AE	Trigger with given name already exists
4240	DMEC	AE	Unsupported trigger type
4241	DMEC	AE	Index name too long
4242	DMEC	AE	Too many indexes
4243	DMEC	AE	Index not found
4247	DMEC	AE	Illegal index/trigger create/drop/alter request
4248	DMEC	AE	Trigger/index name invalid
4249	DMEC	AE	Invalid table
4250	DMEC	AE	Invalid index type or index logging option
4251	HA_ERR_FOUND_DUPP_UNIQUE	AE	Cannot create unique index, duplicate keys found
4252	DMEC	AE	Failed to allocate space for index
4253	DMEC	AE	Failed to create index table

NDB Error Code	MySQL Error Code	NDB Error Type	Error Message
4254	DMEC	AE	Table not an index table
4255	DMEC	AE	Hash index attributes must be specified in same order as table attributes
4256	DMEC	AE	Must call Ndb::init() before this function
4257	DMEC	AE	Tried to read too much - too many getValue calls
4258	DMEC	AE	Cannot create unique index, duplicate attributes found in definition
4259	DMEC	AE	Invalid set of range scan bounds
4264	DMEC	AE	Invalid usage of blob attribute
4265	DMEC	AE	The method is not valid in current blob state
4266	DMEC	AE	Invalid blob seek position
4271	DMEC	AE	Invalid index object, not retrieved via getIndex()
4272	DMEC	AE	Table definition has undefined column
4275	DMEC	AE	The blob method is incompatible with operation type or lock mode
4276	DMEC	AE	Missing NULL ptr in end of keyData list
4277	DMEC	AE	Key part len is to small for column
4278	DMEC	AE	Supplied buffer to small
4279	DMEC	AE	Malformed string
4280	DMEC	AE	Inconsistent key part length
4281	DMEC	AE	Too many keys specified for key bound in scanIndex
4282	DMEC	AE	range_no not strictly increasing in ordered multi-range index scan
4283	DMEC	AE	key_record in index scan is not an index ndbrecord
4284	DMEC	AE	Cannot mix NdbRecAttr and NdbRecord methods in one operation
4285	DMEC	AE	NULL NdbRecord pointer
4286	DMEC	AE	Invalid range_no (must be < 4096)
4287	DMEC	AE	The key_record and attribute_record in primary key operation do not belong to the same table
4288	DMEC	AE	Blob handle for column not available
4289	DMEC	AE	API version mismatch or wrong sizeof(NdbDictionary::RecordSpecification)
4290	DMEC	AE	Missing column specification in NdbDictionary::RecordSpecification
4291	DMEC	AE	Duplicate column specification in NdbDictionary::RecordSpecification
4292	DMEC	AE	NdbRecord for tuple access is not an index key NdbRecord

NDB Error Code	MySQL Error Code	NDB Error Type	Error Message
4293	DMEC	AE	Error returned from application scanIndex() callback
4294	DMEC	AE	Scan filter is too large, discarded
4295	DMEC	AE	Column is NULL in Get/SetValueSpec structure
4296	DMEC	AE	Invalid AbortOption
4297	DMEC	AE	Invalid or unsupported OperationOptions structure
4298	DMEC	AE	Invalid or unsupported ScanOptions structure
4299	DMEC	AE	Incorrect combination of ScanOption flags, extraGetValues ptr and numExtraGetValues
4300	DMEC	AE	Tuple Key Type not correct
4301	DMEC	AE	Fragment Type not correct
4302	DMEC	AE	Minimum Load Factor not correct
4303	DMEC	AE	Maximum Load Factor not correct
4304	DMEC	AE	Maximum Load Factor smaller than Minimum
4305	DMEC	AE	K value must currently be set to 6
4306	DMEC	AE	Memory Type not correct
4307	DMEC	AE	Invalid table name
4308	DMEC	AE	Attribute Size not correct
4309	DMEC	AE	Fixed array too large, maximum 64000 bytes
4310	DMEC	AE	Attribute Type not correct
4311	DMEC	AE	Storage Mode not correct
4312	DMEC	AE	Null Attribute Type not correct
4313	DMEC	AE	Index only storage for non-key attribute
4314	DMEC	AE	Storage Type of attribute not correct
4315	DMEC	AE	No more key attributes allowed after defining variable length key attribute
4316	DMEC	AE	Key attributes are not allowed to be NULL attributes
4317	DMEC	AE	Too many primary keys defined in table
4318	DMEC	AE	Invalid attribute name or number
4319	DMEC	AE	createAttribute called at erroneus place
4322	DMEC	AE	Attempt to define distribution key when not prepared to
4323	DMEC	AE	Distribution Key set on table but not defined on first attribute
4324	DMEC	AE	Attempt to define distribution group when not prepared to

NDB Error Code	MySQL Error Code	NDB Error Type	Error Message
4325	DMEC	AE	Distribution Group set on table but not defined on first attribute
4326	DMEC	AE	Distribution Group with erroneus number of bits
4327	DMEC	AE	Distribution key is only supported on part of primary key
4328	DMEC	AE	Disk memory attributes not yet supported
4329	DMEC	AE	Variable stored attributes not yet supported
4335	DMEC	AE	Only one autoincrement column allowed per table. Having a table without primary key uses an autoincremented hidden key, i.e. a table without a primary key can not have an autoincremented column
4340	DMEC	AE	Result or attribute record must be a base table ndbrecord, not an index ndbrecord
4341	DMEC	AE	Not all keys read when using option SF_OrderBy
4400	DMEC	AE	Status Error in NdbSchemaCon
4401	DMEC	AE	Only one schema operation per schema transaction
4402	DMEC	AE	No schema operation defined before calling execute
4410	DMEC	AE	Schema transaction is already started
4411	DMEC	AE	Schema transaction not possible until upgrade complete
4501	DMEC	AE	Insert in hash table failed when getting table information from Ndb
4502	DMEC	AE	GetValue not allowed in Update operation
4503	DMEC	AE	GetValue not allowed in Insert operation
4504	DMEC	AE	SetValue not allowed in Read operation
4505	DMEC	AE	NULL value not allowed in primary key search
4506	DMEC	AE	Missing getValue/setValue when calling execute
4507	DMEC	AE	Missing operation request when calling execute
4508	DMEC	AE	GetValue not allowed for NdbRecord defined operation
4509	DMEC	AE	Non SF_MultiRange scan cannot have more than one bound
4510	DMEC	AE	User specified partition id not allowed for scan takeover operation

NDB Error Code	MySQL Error Code	NDB Error Type	Error Message
4511	DMEC	AE	Blobs not allowed in NdbRecord delete result record
4512	DMEC	AE	Incorrect combination of OperationOptions optionsPresent, extraGet/SetValues ptr and numExtraGet/SetValues
4513	DMEC	AE	Only one scan bound allowed for non-NdbRecord setBound() API
4514	DMEC	AE	Can only call setBound/equal() for an NdbIndexScanOperation
4515	DMEC	AE	Method not allowed for NdbRecord, use OperationOptions or ScanOptions structure instead
4516	DMEC	AE	Illegal instruction in interpreted program
4517	DMEC	AE	Bad label in branch instruction
4518	DMEC	AE	Too many instructions in interpreted program
4519	DMEC	AE	NdbInterpretedCode::finalise() not called
4520	DMEC	AE	Call to undefined subroutine
4521	DMEC	AE	Call to undefined subroutine, internal error
4522	DMEC	AE	setBound() called twice for same key
4523	DMEC	AE	Pseudo columns not supported by NdbRecord
4524	DMEC	AE	NdbInterpretedCode is for different table
4535	DMEC	AE	Attempt to set bound on non key column
4536	DMEC	AE	NdbScanFilter constructor taking NdbOperation is not supported for NdbRecord
4537	DMEC	AE	Wrong API. Use NdbInterpretedCode for NdbRecord operations
4538	DMEC	AE	NdbInterpretedCode instruction requires that table is set
4539	DMEC	AE	NdbInterpretedCode not supported for operation type
4540	DMEC	AE	Attempt to pass an Index column to createRecord. Use base table columns only
4542	DMEC	AE	Unknown partition information type
4543	DMEC	AE	Duplicate partitioning information supplied
4544	DMEC	AE	Wrong partitionInfo type for table

NDB Error Code	MySQL Error Code	NDB Error Type	Error Message
4545	DMEC	AE	Invalid or Unsupported PartitionInfo structure
4546	DMEC	AE	Explicit partitioning info not allowed for table and operation
4547	DMEC	AE	RecordSpecification has overlapping offsets
4548	DMEC	AE	RecordSpecification has too many elements
4549	DMEC	AE	getLockHandle only supported for primary key read with a lock
4550	DMEC	AE	Cannot releaseLockHandle until operation executed
4551	DMEC	AE	NdbLockHandle already released
4552	DMEC	AE	NdbLockHandle does not belong to transaction
4553	DMEC	AE	NdbLockHandle original operation not executed successfully
4554	DMEC	AE	NdbBlob can only be closed from Active state
4555	DMEC	AE	NdbBlob cannot be closed with pending operations
4556	DMEC	AE	RecordSpecification has illegal value in column_flags
4600	DMEC	AE	Transaction is already started
4601	DMEC	AE	Transaction is not started
4602	DMEC	AE	You must call getNdbOperation before executeScan
4603	DMEC	AE	There can only be ONE operation in a scan transaction
4604	DMEC	AE	takeOverScanOp, to take over a scanned row one must explicitly request keyinfo on readTuples call
4605	DMEC	AE	You may only call readTuples() once for each operation
4607	DMEC	AE	There may only be one operation in a scan transaction
4608	DMEC	AE	You can not takeOverScan unless you have used openScanExclusive
4609	DMEC	AE	You must call nextScanResult before trying to takeOverScan
4707	DMEC	AE	Too many event have been defined
4708	DMEC	AE	Event name is too long
4709	DMEC	AE	Can't accept more subscribers
4710	DMEC	AE	Event not found
4711	DMEC	AE	Creation of event failed

NDB Error Code	MySQL Error Code	NDB Error Type	Error Message
4712	DMEC	AE	Stopped event operation does not exist. Already stopped?
4714	DMEC	AE	Index stats sys tables NDB_INDEX_STAT_PREFIX do not exist
4715	DMEC	AE	Index stats for specified index do not exist
4716	DMEC	AE	Index stats methods usage error
4717	DMEC	AE	Index stats cannot allocate memory
4720	DMEC	AE	Index stats sys tables NDB_INDEX_STAT_PREFIX partly missing or invalid
4723	DMEC	AE	Mysqld: index stats request ignored due to recent error
4724	DMEC	AE	Mysqld: index stats request aborted by stats thread
4725	DMEC	AE	Index stats were deleted by another process
21000	HA_ERR_CANNOT_ADD_FOREIGN	AE	Create foreign key failed - parent key is primary key and on-update-cascade is not allowed
21026	HA_ERR_CANNOT_ADD_FOREIGN	AE	Create foreign key failed in NDB - parent index is not unique index
21040	DMEC	AE	Drop foreign key failed in NDB - foreign key not found
21060	DMEC	AE	Build foreign key failed in NDB - foreign key not found
21081	HA_ERR_DROP_INDEX_FK	AE	Drop index not allowed in NDB - used as parent index of a foreign key
21082	HA_ERR_DROP_INDEX_FK	AE	Drop index not allowed in NDB - used as child index of a foreign key
21090	HA_ERR_CANNOT_ADD_FOREIGN	AE	Create foreign key failed in NDB - name contains invalid character (/)

2.4.2.3 No data found Errors

The following table lists and describes NDB **No data found** errors (error type ND).

Table 2.3 This table describes NDB API No data found errors

NDB Error Code	MySQL Error Code	NDB Error Type	Error Message
626	HA_ERR_KEY_NOT_FOUND	ND	Tuple did not exist

2.4.2.4 Constraint violation Errors

The following table lists and describes NDB **Constraint violation** errors (error type CV).

Table 2.4 This table describes NDB API Constraint violation errors

NDB Error Code	MySQL Error Code	NDB Error Type	Error Message
255	HA_ERR_NO_REFERENCED_ROW	CV	Foreign key constraint violated: No parent row found
256	HA_ERR_ROW_IS_REFERENCED	CV	Foreign key constraint violated: Referenced row exists
630	HA_ERR_FOUND_DUPP_KEY	CV	Tuple already existed when attempting to insert
839	DMEC	CV	Illegal null attribute
840	DMEC	CV	Trying to set a NOT NULL attribute to NULL
893	HA_ERR_FOUND_DUPP_KEY	CV	Constraint violation e.g. duplicate value in unique index

2.4.2.5 Schema error Errors

The following table lists and describes `NDB` **`Schema error`**errors (error type `SE`).

Table 2.5 This table describes NDB API Schema error errors

NDB Error Code	MySQL Error Code	NDB Error Type	Error Message
241	HA_ERR_TABLE_DEF_CHANGED	SE	Invalid schema object version
283	HA_ERR_NO_SUCH_TABLE	SE	Table is being dropped
284	HA_ERR_TABLE_DEF_CHANGED	SE	Table not defined in transaction coordinator
285	DMEC	SE	Unknown table error in transaction coordinator
703	DMEC	SE	Invalid table format
704	DMEC	SE	Attribute name too long
705	DMEC	SE	Table name too long
707	DMEC	SE	No more table metadata records (increase MaxNoOfTables)
708	DMEC	SE	No more attribute metadata records (increase MaxNoOfAttributes)
709	HA_ERR_NO_SUCH_TABLE	SE	No such table existed
710	DMEC	SE	Internal: Get by table name not supported, use table id.
712	DMEC	SE	No more hashmap metadata records
723	HA_ERR_NO_SUCH_TABLE	SE	No such table existed
736	DMEC	SE	Unsupported array size
737	HA_WRONG_CREATE_OPTION	SE	Attribute array size too big
738	HA_WRONG_CREATE_OPTION	SE	Record too big
739	HA_WRONG_CREATE_OPTION	SE	Unsupported primary key length
740	HA_WRONG_CREATE_OPTION	SE	Nullable primary key not supported
741	DMEC	SE	Unsupported alter table

NDB Error Code	MySQL Error Code	NDB Error Type	Error Message
743	HA_WRONG_CREATE_OPTION	SE	Unsupported character set in table or index
744	DMEC	SE	Character string is invalid for given character set
745	HA_WRONG_CREATE_OPTION	SE	Distribution key not supported for char attribute (use binary attribute)
750	IE	SE	Invalid file type
751	DMEC	SE	Out of file records
752	DMEC	SE	Invalid file format
753	IE	SE	Invalid filegroup for file
754	IE	SE	Invalid filegroup version when creating file
755	HA_MISSING_CREATE_OPTION	SE	Invalid tablespace
756	DMEC	SE	Index on disk column is not supported
757	DMEC	SE	Varsize bitfield not supported
758	DMEC	SE	Tablespace has changed
759	DMEC	SE	Invalid tablespace version
760	DMEC	SE	File already exists,
761	DMEC	SE	Unable to drop table as backup is in progress
762	DMEC	SE	Unable to alter table as backup is in progress
764	HA_WRONG_CREATE_OPTION	SE	Invalid extent size
765	DMEC	SE	Out of filegroup records
766	DMEC	SE	Cant drop file, no such file
767	DMEC	SE	Cant drop filegroup, no such filegroup
768	DMEC	SE	Cant drop filegroup, filegroup is used
769	DMEC	SE	Drop undofile not supported, drop logfile group instead
770	DMEC	SE	Cant drop file, file is used
773	DMEC	SE	Out of string memory, please modify StringMemory config parameter
774	DMEC	SE	Invalid schema object for drop
775	DMEC	SE	Create file is not supported when Diskless=1
779	HA_WRONG_CREATE_OPTION	SE	Invalid undo buffer size
790	HA_WRONG_CREATE_OPTION	SE	Invalid hashmap
791	HA_WRONG_CREATE_OPTION	SE	Too many total bits in bitfields
792	DMEC	SE	Default value for primary key column not supported
796	DMEC	SE	Out of schema transaction memory
799	HA_WRONG_CREATE_OPTION	SE	Non default partitioning without partitions

NDB Error Code	MySQL Error Code	NDB Error Type	Error Message
881	DMEC	SE	Unable to create table, out of data pages (increase DataMemory)
906	DMEC	SE	Unsupported attribute type in index
907	DMEC	SE	Unsupported character set in table or index
910	HA_ERR_NO_SUCH_TABLE	SE	Index is being dropped
1224	HA_WRONG_CREATE_OPTION	SE	Too many fragments
1225	DMEC	SE	Table not defined in local query handler
1226	HA_ERR_NO_SUCH_TABLE	SE	Table is being dropped
1227	HA_WRONG_CREATE_OPTION	SE	Invalid schema version
1228	DMEC	SE	Cannot use drop table for drop index
1229	DMEC	SE	Too long frm data supplied
1231	DMEC	SE	Invalid table or index to scan
1232	DMEC	SE	Invalid table or index to scan
1407	DMEC	SE	Subscription not found in subscriber manager
1415	DMEC	SE	Subscription not unique in subscriber manager
1417	DMEC	SE	Table in suscription not defined, probably dropped
1418	DMEC	SE	Subscription dropped, no new subscribers allowed
1419	DMEC	SE	Subscription already dropped
1421	DMEC	SE	Partially connected API in NdbOperation::execute()
1422	DMEC	SE	Out of subscription records
1423	DMEC	SE	Out of table records in SUMA
1424	DMEC	SE	Out of MaxNoOfConcurrentSubOperations
1425	DMEC	SE	Subscription being defined...while trying to stop subscriber
1426	DMEC	SE	No such subscriber
1503	DMEC	SE	Out of filegroup records
1504	DMEC	SE	Out of logbuffer memory(specify smaller undo_buffer_size or increase SharedGlobalMemory)
1508	DMEC	SE	Out of file records
1509	DMEC	SE	File system error, check if path,permissions etc
1512	DMEC	SE	File read error
1514	DMEC	SE	Currently there is a limit of one logfile group
1515	DMEC	SE	Currently there is a 4G limit of one undo/data-file in 32-bit host

NDB Error Code	MySQL Error Code	NDB Error Type	Error Message
1516	DMEC	SE	File too small
4713	DMEC	SE	Column defined in event does not exist in table
20019	HA_ERR_NO_SUCH_TABLE	SE	Query table not defined
20020	HA_ERR_NO_SUCH_TABLE	SE	Query table is being dropped
20021	HA_ERR_TABLE_DEF_CHANGED	SE	Query table definition has changed
21022	DMEC	SE	Create foreign key failed in NDB - parent table is not table
21023	DMEC	SE	Create foreign key failed in NDB - invalid parent table version
21024	DMEC	SE	Create foreign key failed in NDB - child table is not table
21025	DMEC	SE	Create foreign key failed in NDB - invalid child table version
21027	DMEC	SE	Create foreign key failed in NDB - invalid parent index version
21028	DMEC	SE	Create foreign key failed in NDB - child index is not index
21029	DMEC	SE	Create foreign key failed in NDB - invalid child index version
21041	DMEC	SE	Drop foreign key failed in NDB - invalid foreign key version
21042	DMEC	SE	Drop foreign key failed in NDB - foreign key not found in TC
21061	DMEC	SE	Build foreign key failed in NDB - invalid foreign key version
21080	HA_ERR_ROW_IS_REFERENCED	SE	Drop table not allowed in NDB - referenced by foreign key on another table

2.4.2.6 Schema object already exists Errors

The following table lists and describes NDB **Schema object already exists**errors (error type OE).

Table 2.6 This table describes NDB API Schema object already exists errors

NDB Error Code	MySQL Error Code	NDB Error Type	Error Message
721	HA_ERR_TABLE_EXIST	OE	Schema object with given name already exists
746	DMEC	OE	Event name already exists
4244	HA_ERR_TABLE_EXIST	OE	Index or table with given name already exists

2.4.2.7 User defined error Errors

The following table lists and describes NDB **User defined error**errors (error type UD).

Table 2.7 This table describes NDB API User defined error errors

NDB Error Code	MySQL Error Code	NDB Error Type	Error Message
1321	DMEC	UD	Backup aborted by user request
4260	DMEC	UD	NdbScanFilter: Operator is not defined in NdbScanFilter::Group
4261	DMEC	UD	NdbScanFilter: Column is NULL
4262	DMEC	UD	NdbScanFilter: Condition is out of bounds

2.4.2.8 Insufficient space Errors

The following table lists and describes `NDB` **`Insufficient space`**errors (error type `IS`).

Table 2.8 This table describes NDB API Insufficient space errors

NDB Error Code	MySQL Error Code	NDB Error Type	Error Message
623	HA_ERR_RECORD_FILE_FULL	IS	623
624	HA_ERR_RECORD_FILE_FULL	IS	624
625	HA_ERR_INDEX_FILE_FULL	IS	Out of memory in Ndb Kernel, hash index part (increase IndexMemory)
633	HA_ERR_INDEX_FILE_FULL	IS	Table fragment hash index has reached maximum possible size
640	DMEC	IS	Too many hash indexes (should not happen)
747	DMEC	IS	Out of event records
826	HA_ERR_RECORD_FILE_FULL	IS	Too many tables and attributes (increase MaxNoOfAttributes or MaxNoOfTables)
827	HA_ERR_RECORD_FILE_FULL	IS	Out of memory in Ndb Kernel, table data (increase DataMemory)
889	HA_ERR_RECORD_FILE_FULL	IS	Table fragment fixed data reference has reached maximum possible value (specify MAXROWS or increase no of partitions)
902	HA_ERR_RECORD_FILE_FULL	IS	Out of memory in Ndb Kernel, ordered index data (increase DataMemory)
903	HA_ERR_INDEX_FILE_FULL	IS	Too many ordered indexes (increase MaxNoOfOrderedIndexes)
904	HA_ERR_INDEX_FILE_FULL	IS	Out of fragment records (increase MaxNoOfOrderedIndexes)
905	DMEC	IS	Out of attribute records (increase MaxNoOfAttributes)
908	DMEC	IS	Invalid ordered index tree node size
1303	DMEC	IS	Out of resources
1412	DMEC	IS	Can't accept more subscribers, out of space in pool

NDB Error Code	MySQL Error Code	NDB Error Type	Error Message
1416	DMEC	IS	Can't accept more subscriptions, out of space in pool
1601	HA_ERR_RECORD_FILE_FULL	IS	Out extents, tablespace full
1602	DMEC	IS	No datafile in tablespace
1603	HA_ERR_RECORD_FILE_FULL	IS	Table fragment fixed data reference has reached maximum possible value (specify MAXROWS or increase no of partitions)

2.4.2.9 Temporary Resource error Errors

The following table lists and describes NDB **Temporary Resource error**errors (error type TR).

Table 2.9 This table describes NDB API Temporary Resource error errors

NDB Error Code	MySQL Error Code	NDB Error Type	Error Message
217	DMEC	TR	217
218	DMEC	TR	Out of LongMessageBuffer
219	DMEC	TR	219
221	DMEC	TR	Too many concurrently fired triggers (increase MaxNoOfFiredTriggers)
233	DMEC	TR	Out of operation records in transaction coordinator (increase MaxNoOfConcurrentOperations)
245	DMEC	TR	Too many active scans
273	DMEC	TR	Out of transaction markers databuffer in transaction coordinator
275	DMEC	TR	Out of transaction records for complete phase (increase MaxNoOfConcurrentTransactions)
279	DMEC	TR	Out of transaction markers in transaction coordinator
288	DMEC	TR	Out of index operations in transaction coordinator (increase MaxNoOfConcurrentIndexOperations)
289	DMEC	TR	Out of transaction buffer memory in TC (increase TransactionBufferMemory)
291	DMEC	TR	Out of scanfrag records in TC (increase MaxNoOfLocalScans)
312	DMEC	TR	Out of LongMessageBuffer
414	DMEC	TR	414
418	DMEC	TR	Out of transaction buffers in LQH
419	DMEC	TR	419
488	DMEC	TR	Too many active scans
489	DMEC	TR	Too many active scans
490	DMEC	TR	Too many active scans

NDB Error Code	MySQL Error Code	NDB Error Type	Error Message
748	DMEC	TR	Busy during read of event table
780	DMEC	TR	Too many schema transactions
783	DMEC	TR	Too many schema operations
784	DMEC	TR	Invalid schema transaction state
785	DMEC	TR	Schema object is busy with another schema transaction
788	DMEC	TR	Missing schema operation at takeover of schema transaction
805	DMEC	TR	Out of attrinfo records in tuple manager
830	DMEC	TR	Out of add fragment operation records
873	DMEC	TR	Out of attrinfo records for scan in tuple manager
899	DMEC	TR	Rowid already allocated
915	DMEC	TR	No free index stats op
918	DMEC	TR	Cannot prepare index stats update
919	DMEC	TR	Cannot execute index stats update
1217	DMEC	TR	Out of operation records in local data manager (increase MaxNoOfLocalOperations)
1218	DMEC	TR	Send Buffers overloaded in NDB kernel
1220	DMEC	TR	REDO log files overloaded (increase FragmentLogFileSize)
1222	DMEC	TR	Out of transaction markers in LQH
1234	DMEC	TR	REDO log files overloaded (increase disk hardware)
1350	DMEC	TR	Backup failed: file already exists (use 'START BACKUP <backup id>')
1411	DMEC	TR	Subscriber manager busy with adding/ removing a subscriber
1413	DMEC	TR	Subscriber manager busy with adding the subscription
1414	DMEC	TR	Subscriber manager has subscribers on this subscription
1420	DMEC	TR	Subscriber manager busy with adding/ removing a table
1501	DMEC	TR	Out of undo space
2810	DMEC	TR	No space left on the device
2811	DMEC	TR	Error with file permissions, please check file system
2815	DMEC	TR	Error in reading files, please check file system
4021	DMEC	TR	Out of Send Buffer space in NDB API
4022	DMEC	TR	Out of Send Buffer space in NDB API
4032	DMEC	TR	Out of Send Buffer space in NDB API

NDB Error Code	MySQL Error Code	NDB Error Type	Error Message
20000	DMEC	TR	Query aborted due out of operation records
20006	DMEC	TR	Query aborted due to out of LongMessageBuffer
20008	DMEC	TR	Query aborted due to out of query memory
20015	DMEC	TR	Query aborted due to out of row memory
21020	DMEC	TR	Create foreign key failed in NDB - no more object records

2.4.2.10 Node Recovery error Errors

The following table lists and describes NDB **Node Recovery error**errors (error type NR).

Table 2.10 This table describes NDB API Node Recovery error errors

NDB Error Code	MySQL Error Code	NDB Error Type	Error Message
250	DMEC	NR	Node where lock was held crashed, restart scan transaction
286	DMEC	NR	Node failure caused abort of transaction
499	DMEC	NR	Scan take over error, restart scan transaction
786	DMEC	NR	Schema transaction aborted due to node-failure
1204	DMEC	NR	Temporary failure, distribution changed
1405	DMEC	NR	Subscriber manager busy with node recovery
1427	DMEC	NR	Api node died, when SUB_START_REQ reached node
4002	DMEC	NR	Send to NDB failed
4007	DMEC	NR	Send to ndbd node failed
4010	DMEC	NR	Node failure caused abort of transaction
4013	DMEC	NR	Request timed out in waiting for node failure
4025	DMEC	NR	Node failure caused abort of transaction
4027	DMEC	NR	Node failure caused abort of transaction
4028	DMEC	NR	Node failure caused abort of transaction
4029	DMEC	NR	Node failure caused abort of transaction
4031	DMEC	NR	Node failure caused abort of transaction
4033	DMEC	NR	Send to NDB failed
4035	DMEC	NR	Cluster temporary unavailable
4115	DMEC	NR	Transaction was committed but all read information was not received due to node crash

NDB Error Code	MySQL Error Code	NDB Error Type	Error Message
4119	DMEC	NR	Simple/dirty read failed due to node failure
20016	DMEC	NR	Query aborted due to node failure

2.4.2.11 Overload error Errors

The following table lists and describes NDB **Overload error** errors (error type OL).

Table 2.11 This table describes NDB API Overload error errors

NDB Error Code	MySQL Error Code	NDB Error Type	Error Message
410	DMEC	OL	REDO log files overloaded (decrease TimeBetweenLocalCheckpoints or increase NoOfFragmentLogFiles)
677	DMEC	OL	Index UNDO buffers overloaded (increase UndoIndexBuffer)
701	DMEC	OL	System busy with other schema operation
711	DMEC	OL	System busy with node restart, schema operations not allowed
891	DMEC	OL	Data UNDO buffers overloaded (increase UndoDataBuffer)
1221	DMEC	OL	REDO buffers overloaded (increase RedoBuffer)
4006	DMEC	OL	Connect failure - out of connection objects (increase MaxNoOfConcurrentTransactions)

2.4.2.12 Timeout expired Errors

The following table lists and describes NDB **Timeout expired** errors (error type TO).

Table 2.12 This table describes NDB API Timeout expired errors

NDB Error Code	MySQL Error Code	NDB Error Type	Error Message
237	HA_ERR_LOCK_WAIT_TIMEOUT	TO	Transaction had timed out when trying to commit it
266	HA_ERR_LOCK_WAIT_TIMEOUT	TO	Time-out in NDB, probably caused by deadlock
4351	DMEC	TO	Timeout/deadlock during index build

2.4.2.13 Node shutdown Errors

The following table lists and describes NDB **Node shutdown** errors (error type NS).

Table 2.13 This table describes NDB API Node shutdown errors

NDB Error Code	MySQL Error Code	NDB Error Type	Error Message
270	DMEC	NS	Transaction aborted due to node shutdown
280	DMEC	NS	Transaction aborted due to node shutdown
1223	DMEC	NS	Read operation aborted due to node shutdown
4023	DMEC	NS	Transaction aborted due to node shutdown
4030	DMEC	NS	Transaction aborted due to node shutdown
4034	DMEC	NS	Transaction aborted due to node shutdown

2.4.2.14 Internal temporary Errors

The following table lists and describes `NDB` **`Internal temporary`**errors (error type `IT`).

Table 2.14 This table describes NDB API Internal temporary errors

NDB Error Code	MySQL Error Code	NDB Error Type	Error Message
702	DMEC	IT	Request to non-master
787	DMEC	IT	Schema transaction aborted
1703	DMEC	IT	Node failure handling not completed

2.4.2.15 Unknown result error Errors

The following table lists and describes `NDB` **`Unknown result error`**errors (error type `UR`).

Table 2.15 This table describes NDB API Unknown result error errors

NDB Error Code	MySQL Error Code	NDB Error Type	Error Message
4008	DMEC	UR	Receive from NDB failed
4009	HA_ERR_NO_CONNECTION	UR	Cluster Failure
4012	DMEC	UR	Request ndbd time-out, maybe due to high load or communication problems

2.4.2.16 Internal error Errors

The following table lists and describes `NDB` **`Internal error`**errors (error type `IE`).

Table 2.16 This table describes NDB API Internal error errors

NDB Error Code	MySQL Error Code	NDB Error Type	Error Message
202	DMEC	IE	202
203	DMEC	IE	203
207	DMEC	IE	207

NDB Error Code	MySQL Error Code	NDB Error Type	Error Message
208	DMEC	IE	208
209	DMEC	IE	Communication problem, signal error
220	DMEC	IE	220
230	DMEC	IE	230
232	DMEC	IE	232
238	DMEC	IE	238
271	DMEC	IE	Simple Read transaction without any attributes to read
272	DMEC	IE	Update operation without any attributes to update
276	DMEC	IE	276
277	DMEC	IE	277
278	DMEC	IE	278
287	DMEC	IE	Index corrupted
290	DMEC	IE	Corrupt key in TC, unable to xfrm
292	DMEC	IE	Inconsistent index state in TC block
293	DMEC	IE	Inconsistent trigger state in TC block
294	DMEC	IE	Unlocked operation has out of range index
295	DMEC	IE	Unlocked operation has invalid state
298	DMEC	IE	Invalid distribution key
306	DMEC	IE	Out of fragment records in DIH
416	DMEC	IE	Bad state handling unlock request
631	DMEC	IE	631
632	DMEC	IE	632
706	DMEC	IE	Inconsistency during table creation
749	HA_WRONG_CREATE_OPTION	IE	Primary Table in wrong state
772	HA_WRONG_CREATE_OPTION	IE	Given fragmentType doesn't exist
781	DMEC	IE	Invalid schema transaction key from NDB API
782	DMEC	IE	Invalid schema transaction id from NDB API
795	DMEC	IE	Out of LongMessageBuffer in DICT
809	DMEC	IE	809
812	DMEC	IE	812
833	DMEC	IE	833
871	DMEC	IE	871
882	DMEC	IE	882
883	DMEC	IE	883
887	DMEC	IE	887
888	DMEC	IE	888
890	DMEC	IE	890

NDB Error Code	MySQL Error Code	NDB Error Type	Error Message
896	DMEC	IE	Tuple corrupted - wrong checksum or column data in invalid format
901	DMEC	IE	Inconsistent ordered index. The index needs to be dropped and recreated
909	DMEC	IE	No free index scan op
914	DMEC	IE	Invalid index stats request
916	DMEC	IE	Invalid index stats sys tables
917	DMEC	IE	Invalid index stats sys tables data
1300	DMEC	IE	Undefined error
1301	DMEC	IE	Backup issued to not master (reissue command to master)
1304	DMEC	IE	Sequence failure
1305	DMEC	IE	Backup definition not implemented
1322	DMEC	IE	Backup already completed
1323	DMEC	IE	1323
1324	DMEC	IE	Backup log buffer full
1325	DMEC	IE	File or scan error
1326	DMEC	IE	Backup aborted due to node failure
1327	DMEC	IE	1327
1340	DMEC	IE	Backup undefined error
1428	DMEC	IE	No replica to scan on this node (internal index stats error)
1429	DMEC	IE	Subscriber node undefined in SubStartReq (config change?)
1502	DMEC	IE	Filegroup already exists
1505	DMEC	IE	Invalid filegroup
1506	DMEC	IE	Invalid filegroup version
1507	DMEC	IE	File no already inuse
1510	DMEC	IE	File meta data error
1511	DMEC	IE	Out of memory
1513	DMEC	IE	Filegroup not online
1700	DMEC	IE	Undefined error
4000	DMEC	IE	MEMORY ALLOCATION ERROR
4001	DMEC	IE	Signal Definition Error
4005	DMEC	IE	Internal Error in NdbApi
4011	DMEC	IE	Internal Error in NdbApi
4107	DMEC	IE	Simple Transaction and Not Start
4108	DMEC	IE	Faulty operation type
4109	DMEC	IE	Faulty primary key attribute length
4110	DMEC	IE	Faulty length in ATTRINFO signal
4111	DMEC	IE	Status Error in NdbConnection
4113	DMEC	IE	Too many operations received

NDB Error Code	MySQL Error Code	NDB Error Type	Error Message
4263	DMEC	IE	Invalid blob attributes or invalid blob parts table
4267	DMEC	IE	Corrupted blob value
4268	DMEC	IE	Error in blob head update forced rollback of transaction
4269	DMEC	IE	No connection to ndb management server
4270	DMEC	IE	Unknown blob error
4273	DMEC	IE	No blob table in dict cache
4274	DMEC	IE	Corrupted main table PK in blob operation
4320	DMEC	IE	Cannot use the same object twice to create table
4321	DMEC	IE	Trying to start two schema transactions
4344	DMEC	IE	Only DBDICT and TRIX can send requests to TRIX
4345	DMEC	IE	TRIX block is not available yet, probably due to node failure
4346	DMEC	IE	Internal error at index create/build
4347	DMEC	IE	Bad state at alter index
4348	DMEC	IE	Inconsistency detected at alter index
4349	DMEC	IE	Inconsistency detected at index usage
4350	DMEC	IE	Transaction already aborted
4718	DMEC	IE	Index stats samples data or memory cache is invalid
4719	DMEC	IE	Index stats internal error
4721	DMEC	IE	Mysqld: index stats thread not open for requests
4722	DMEC	IE	Mysqld: index stats entry unexpectedly not found
4731	DMEC	IE	Event not found
20001	DMEC	IE	Query aborted due to empty query tree
20002	DMEC	IE	Query aborted due to invalid request
20003	DMEC	IE	Query aborted due to unknown query operation
20004	DMEC	IE	Query aborted due to invalid tree node specification
20005	DMEC	IE	Query aborted due to invalid tree parameter specification
20007	DMEC	IE	Query aborted due to invalid pattern
20009	DMEC	IE	Query aborted due to query node too big
20010	DMEC	IE	Query aborted due to query node parameters too big

NDB Error Code	MySQL Error Code	NDB Error Type	Error Message
20011	DMEC	IE	Query aborted due to both tree and parameters contain interpreted program
20012	DMEC	IE	Query aborted due to invalid tree parameter specification: Key parameter bits mismatch
20013	DMEC	IE	Query aborted due to invalid tree parameter specification: Incorrect key parameter count
20014	DMEC	IE	Query aborted due to internal error
20017	DMEC	IE	Query aborted due to invalid node count
20018	DMEC	IE	Query aborted due to index fragment not found
21021	DMEC	IE	Create foreign key failed in NDB - invalid request
21030	DMEC	IE	Create foreign key failed in NDB - object already exists in TC
21031	DMEC	IE	Create foreign key failed in NDB - no more object records in TC
21032	DMEC	IE	Create foreign key failed in NDB - invalid request to TC

2.4.2.17 Function not implemented Errors

The following table lists and describes NDB **Function not implemented**errors (error type NI).

Table 2.17 This table describes NDB API Function not implemented errors

NDB Error Code	MySQL Error Code	NDB Error Type	Error Message
797	DMEC	NI	Wrong fragment count for fully replicated table
4003	DMEC	NI	Function not implemented yet

2.4.3 NDB Error Codes: Single Listing

The following table lists all NDB errors, ordered by error code. Each row also includes the error's NDB error type, the corresponding MySQL Server error, and the text of the error message.

Table 2.18 This table describes NDB errors

NDB Error Code	MySQL Error Name	NDB Error Type	Error Message
QRY_REQ_ARG	DMEC	Application error	Required argument is NULL
QRY_TOO_FEW_KEY_VALUES	DMEC	Application error	All required 'key' values was not specified
QRY_TOO_MANY_KEY_VALUES	DMEC	Application error	Too many 'key' or 'bound' values was specified

NDB Error Code	MySQL Error Name	NDB Error Type	Error Message
QRY_OPERAND_HAS_WRONG_TYPE	ER_WRONG_ARGUMENTS	Application error	Incompatible datatype specified in operand argument
QRY_CHAR_OPERAND_TRUNCATED	ER_WARN_DATA_TRUNCATED	Application error	Character operand was right truncated
QRY_NUM_OPERAND_RANGE	ER_WARN_DATA_OUT_OF_RANGE	Application error	Numeric operand out of range
QRY_MULTIPLE_PARENTS	ER_WRONG_ARGUMENTS	Application error	Multiple 'parents' specified in linkedValues for this operation
QRY_UNKNOWN_PARENT	ER_KEY_DOES_NOT_EXIST	Application error	Unknown 'parent' specified in linkedValue
QRY_UNRELATED_INDEX	ER_WRONG_ARGUMENTS	Application error	Specified 'index' does not belong to specified 'table'
QRY_WRONG_INDEX_TYPE	ER_WRONG_USAGE	Application error	Wrong type of index specified for this operation
QRY_OPERAND_ALREADY_BOUND	ER_WRONG_USAGE	Application error	Can't use same operand value to specify different column values
QRY_DEFINITION_TOO_LARGE	ER_TOO_BIG_SELECT	Application error	Query definition too large.
QRY_RESULT_ROW_ALREADY_DEFINED	ER_WRONG_USAGE	Application error	Result row already defined for NdbQueryOperation.
QRY_HAS_ZERO_OPERATIONS	ER_ZERO_DEFINITION	Application error	Query defintion should have at least one operation.
QRY_IN_ERROR_STATE	ER_WRONG_USAGE	Application error	A previous query operation failed, which you missed to catch.
QRY_ILLEGAL_STATE	ER_WRONG_USAGE	Application error	Query is in illegal state for this operation.
QRY_WRONG_OPERATION_TYPE	ER_WRONG_USAGE	Application error	This method cannot be invoked on this type of operation (lookup/scan/index scan).
QRY_SCAN_ORDER_ALREADY_SET	ER_WRONG_USAGE	Application error	Index scan order was already set in query definition.
QRY_PARAMETER_HAS_WRONG_TYPE	ER_WRONG_ARGUMENTS	Application error	Parameter value has an incompatible datatype
QRY_CHAR_PARAMETER_TRUNCATED	ER_WARN_DATA_TRUNCATED	Application error	Character Parameter was right truncated
QRY_MULTIPLE_SCAN_SORTED	ER_WRONG_USAGE	Application error	Query with multiple scans may not be sorted.
QRY_SEQUENTIAL_SCAN_SORTED	ER_WRONG_USAGE	Application error	Parallelism cannot be restricted for sorted scans.
QRY_BATCH_SIZE_TOO_SMALL	ER_WRONG_USAGE	Application error	Batch size for sub scan cannot be smaller than number of fragments.
QRY_EMPTY_PROJECTION	ER_WRONG_USAGE	Application error	Query has operation with empty projection.
NO_CONTACT_WITH_PROCESS	ER_QUERY_EXEC_FAIL	Application error	No contact with the process (dead ?).

NDB Error Code	MySQL Error Name	NDB Error Type	Error Message
WRONG_PROCESS_TYPE	DMEC	Application error	The process has wrong type. Expected a DB process.
SEND_OR_RECEIVE_FAILED	DMEC	Application error	Send to process or receive failed.
INVALID_ERROR_NUMBER	DMEC	Application error	Invalid error number. Should be >= 0.
INVALID_TRACE_NUMBER	DMEC	Application error	Invalid trace number.
INVALID_BLOCK_NAME	DMEC	Application error	Invalid block name
NODE_SHUTDOWN_IN_PROGESS	DMEC	Application error	Node shutdown in progress
SYSTEM_SHUTDOWN_IN_PROGRESS	DMEC	Application error	System shutdown in progress
NODE_SHUTDOWN_WOULD_CAUSE_SYSTEM_CRASH	DMEC	Application error	Node shutdown would cause system crash
UNSUPPORTED_NODE_SHUTDOWN	DMEC	Application error	Unsupported multi node shutdown. Abort option required.
NODE_NOT_API_NODE	DMEC	Application error	The specified node is not an API node.
OPERATION_NOT_ALLOWED_START_STOP	DMEC	Application error	Operation not allowed while nodes are starting or stopping.
NO_CONTACT_WITH_DB_NODES	DMEC	Application error	No contact with database nodes }
0	0	No error	No error
202	DMEC	Internal error	202
203	DMEC	Internal error	203
207	DMEC	Internal error	207
208	DMEC	Internal error	208
209	DMEC	Internal error	Communication problem, signal error
217	DMEC	Temporary Resource error	217
218	DMEC	Temporary Resource error	Out of LongMessageBuffer
219	DMEC	Temporary Resource error	219
220	DMEC	Internal error	220

NDB Error Code	MySQL Error Name	NDB Error Type	Error Message
221	DMEC	Temporary Resource error	Too many concurrently fired triggers (increase MaxNoOfFiredTriggers)
230	DMEC	Internal error	230
232	DMEC	Internal error	232
233	DMEC	Temporary Resource error	Out of operation records in transaction coordinator (increase MaxNoOfConcurrentOperations)
237	HA_ERR_LOCK_WAIT_TIMEOUT	Timeout expired	Transaction had timed out when trying to commit it
238	DMEC	Internal error	238
241	HA_ERR_TABLE_DEF_CHANGED	Schema error	Invalid schema object version
242	DMEC	Application error	Zero concurrency in scan
244	DMEC	Application error	Too high concurrency in scan
245	DMEC	Temporary Resource error	Too many active scans
250	DMEC	Node Recovery error	Node where lock was held crashed, restart scan transaction
255	HA_ERR_NO_REFERENCED_ROW	Constraint violation	Foreign key constraint violated: No parent row found
256	HA_ERR_ROW_IS_REFERENCED	Constraint violation	Foreign key constraint violated: Referenced row exists
261	DMEC	Application error	DML count in transaction exceeds config parameter MaxDMLOperationsPerTransaction
266	HA_ERR_LOCK_WAIT_TIMEOUT	Timeout expired	Time-out in NDB, probably caused by deadlock
269	DMEC	Application error	No condition and attributes to read in scan
270	DMEC	Node shutdown	Transaction aborted due to node shutdown
271	DMEC	Internal error	Simple Read transaction without any attributes to read
272	DMEC	Internal error	Update operation without any attributes to update
273	DMEC	Temporary Resource error	Out of transaction markers databuffer in transaction coordinator

NDB Error Code	MySQL Error Name	NDB Error Type	Error Message
275	DMEC	Temporary Resource error	Out of transaction records for complete phase (increase MaxNoOfConcurrentTransactions)
276	DMEC	Internal error	276
277	DMEC	Internal error	277
278	DMEC	Internal error	278
279	DMEC	Temporary Resource error	Out of transaction markers in transaction coordinator
280	DMEC	Node shutdown	Transaction aborted due to node shutdown
281	HA_ERR_NO_CONNECTION	Application error	Operation not allowed due to cluster shutdown in progress
283	HA_ERR_NO_SUCH_TABLE	Schema error	Table is being dropped
284	HA_ERR_TABLE_DEF_CHANGED	Schema error	Table not defined in transaction coordinator
285	DMEC	Schema error	Unknown table error in transaction coordinator
286	DMEC	Node Recovery error	Node failure caused abort of transaction
287	DMEC	Internal error	Index corrupted
288	DMEC	Temporary Resource error	Out of index operations in transaction coordinator (increase MaxNoOfConcurrentIndexOperations)
289	DMEC	Temporary Resource error	Out of transaction buffer memory in TC (increase TransactionBufferMemory)
290	DMEC	Internal error	Corrupt key in TC, unable to xfrm
291	DMEC	Temporary Resource error	Out of scanfrag records in TC (increase MaxNoOfLocalScans)
292	DMEC	Internal error	Inconsistent index state in TC block
293	DMEC	Internal error	Inconsistent trigger state in TC block
294	DMEC	Internal error	Unlocked operation has out of range index
295	DMEC	Internal error	Unlocked operation has invalid state

NDB Error Code	MySQL Error Name	NDB Error Type	Error Message
298	DMEC	Internal error	Invalid distribution key
299	DMEC	Application error	Operation not allowed or aborted due to single user mode
306	DMEC	Internal error	Out of fragment records in DIH
311	DMEC	Application error	Undefined partition used in setPartitionId
312	DMEC	Temporary Resource error	Out of LongMessageBuffer
320	DMEC	Application error	Invalid no of nodes specified for new nodegroup
321	DMEC	Application error	Invalid nodegroup id
322	DMEC	Application error	Invalid node(s) specified for new nodegroup, node already in nodegroup
323	DMEC	Application error	Invalid nodegroup id, nodegroup already existing
324	DMEC	Application error	Invalid node(s) specified for new nodegroup, no node in nodegroup is started
325	DMEC	Application error	Invalid node(s) specified for new nodegroup, node ID invalid or undefined
410	DMEC	Overload error	REDO log files overloaded (decrease TimeBetweenLocalCheckpoints or increase NoOfFragmentLogFiles)
414	DMEC	Temporary Resource error	414
416	DMEC	Internal error	Bad state handling unlock request
417	DMEC	Application error	Bad operation reference - double unlock
418	DMEC	Temporary Resource error	Out of transaction buffers in LQH
419	DMEC	Temporary Resource error	419
488	DMEC	Temporary Resource error	Too many active scans
489	DMEC	Temporary Resource error	Too many active scans

NDB Error Code	MySQL Error Name	NDB Error Type	Error Message
490	DMEC	Temporary Resource error	Too many active scans
499	DMEC	Node Recovery error	Scan take over error, restart scan transaction
623	HA_ERR_RECORD_FILE_FULL	Insufficient space	623
624	HA_ERR_RECORD_FILE_FULL	Insufficient space	624
625	HA_ERR_INDEX_FILE_FULL	Insufficient space	Out of memory in Ndb Kernel, hash index part (increase IndexMemory)
626	HA_ERR_KEY_NOT_FOUND	No data found	Tuple did not exist
630	HA_ERR_FOUND_DUPP_KEY	Constraint violation	Tuple already existed when attempting to insert
631	DMEC	Internal error	631
632	DMEC	Internal error	632
633	HA_ERR_INDEX_FILE_FULL	Insufficient space	Table fragment hash index has reached maximum possible size
640	DMEC	Insufficient space	Too many hash indexes (should not happen)
677	DMEC	Overload error	Index UNDO buffers overloaded (increase UndoIndexBuffer)
701	DMEC	Overload error	System busy with other schema operation
702	DMEC	Internal temporary	Request to non-master
703	DMEC	Schema error	Invalid table format
704	DMEC	Schema error	Attribute name too long
705	DMEC	Schema error	Table name too long
706	DMEC	Internal error	Inconsistency during table creation
707	DMEC	Schema error	No more table metadata records (increase MaxNoOfTables)
708	DMEC	Schema error	No more attribute metadata records (increase MaxNoOfAttributes)
709	HA_ERR_NO_SUCH_TABLE	Schema error	No such table existed
710	DMEC	Schema error	Internal: Get by table name not supported, use table id.

NDB Error Code	MySQL Error Name	NDB Error Type	Error Message
711	DMEC	Overload error	System busy with node restart, schema operations not allowed
712	DMEC	Schema error	No more hashmap metadata records
720	DMEC	Application error	Attribute name reused in table definition
721	HA_ERR_TABLE_EXIST	Schema object already exists	Schema object with given name already exists
723	HA_ERR_NO_SUCH_TABLE	Schema error	No such table existed
736	DMEC	Schema error	Unsupported array size
737	HA_WRONG_CREATE_OPTION	Schema error	Attribute array size too big
738	HA_WRONG_CREATE_OPTION	Schema error	Record too big
739	HA_WRONG_CREATE_OPTION	Schema error	Unsupported primary key length
740	HA_WRONG_CREATE_OPTION	Schema error	Nullable primary key not supported
741	DMEC	Schema error	Unsupported alter table
743	HA_WRONG_CREATE_OPTION	Schema error	Unsupported character set in table or index
744	DMEC	Schema error	Character string is invalid for given character set
745	HA_WRONG_CREATE_OPTION	Schema error	Distribution key not supported for char attribute (use binary attribute)
746	DMEC	Schema object already exists	Event name already exists
747	DMEC	Insufficient space	Out of event records
748	DMEC	Temporary Resource error	Busy during read of event table
749	HA_WRONG_CREATE_OPTION	Internal error	Primary Table in wrong state
750	IE	Schema error	Invalid file type
751	DMEC	Schema error	Out of file records

NDB Error Code	MySQL Error Name	NDB Error Type	Error Message
752	DMEC	Schema error	Invalid file format
753	IE	Schema error	Invalid filegroup for file
754	IE	Schema error	Invalid filegroup version when creating file
755	HA_MISSING_CREATE_OPTION	Schema error	Invalid tablespace
756	DMEC	Schema error	Index on disk column is not supported
757	DMEC	Schema error	Varsize bitfield not supported
758	DMEC	Schema error	Tablespace has changed
759	DMEC	Schema error	Invalid tablespace version
760	DMEC	Schema error	File already exists,
761	DMEC	Schema error	Unable to drop table as backup is in progress
762	DMEC	Schema error	Unable to alter table as backup is in progress
763	DMEC	Application error	DDL is not supported with mixed data-node versions
764	HA_WRONG_CREATE_OPTION	Schema error	Invalid extent size
765	DMEC	Schema error	Out of filegroup records
766	DMEC	Schema error	Cant drop file, no such file
767	DMEC	Schema error	Cant drop filegroup, no such filegroup
768	DMEC	Schema error	Cant drop filegroup, filegroup is used
769	DMEC	Schema error	Drop undofile not supported, drop logfile group instead
770	DMEC	Schema error	Cant drop file, file is used
771	HA_WRONG_CREATE_OPTION	Application error	Given NODEGROUP doesn't exist in this cluster
772	HA_WRONG_CREATE_OPTION	Internal error	Given fragmentType doesn't exist
773	DMEC	Schema error	Out of string memory, please modify StringMemory config parameter
774	DMEC	Schema error	Invalid schema object for drop

NDB Error Code	MySQL Error Name	NDB Error Type	Error Message
775	DMEC	Schema error	Create file is not supported when Diskless=1
776	DMEC	Application error	Index created on temporary table must itself be temporary
777	DMEC	Application error	Cannot create a temporary index on a non-temporary table
778	DMEC	Application error	A temporary table or index must be specified as not logging
779	HA_WRONG_CREATE_OPTION	Schema error	Invalid undo buffer size
780	DMEC	Temporary Resource error	Too many schema transactions
781	DMEC	Internal error	Invalid schema transaction key from NDB API
782	DMEC	Internal error	Invalid schema transaction id from NDB API
783	DMEC	Temporary Resource error	Too many schema operations
784	DMEC	Temporary Resource error	Invalid schema transaction state
785	DMEC	Temporary Resource error	Schema object is busy with another schema transaction
786	DMEC	Node Recovery error	Schema transaction aborted due to node-failure
787	DMEC	Internal temporary	Schema transaction aborted
788	DMEC	Temporary Resource error	Missing schema operation at takeover of schema transaction
789	HA_WRONG_CREATE_OPTION	Application error	Logfile group not found
790	HA_WRONG_CREATE_OPTION	Schema error	Invalid hashmap
791	HA_WRONG_CREATE_OPTION	Schema error	Too many total bits in bitfields
792	DMEC	Schema error	Default value for primary key column not supported
793	DMEC	Application error	Object definition too big
794	DMEC	Application error	Schema feature requires data node upgrade

NDB Error Code	MySQL Error Name	NDB Error Type	Error Message
795	DMEC	Internal error	Out of LongMessageBuffer in DICT
796	DMEC	Schema error	Out of schema transaction memory
797	DMEC	Function not implemented	Wrong fragment count for fully replicated table
798	DMEC	Application error	A disk table must not be specified as no logging
799	HA_WRONG_CREATE_OPTION	Schema error	Non default partitioning without partitions
805	DMEC	Temporary Resource error	Out of attrinfo records in tuple manager
809	DMEC	Internal error	809
812	DMEC	Internal error	812
823	DMEC	Application error	Too much attrinfo from application in tuple manager
826	HA_ERR_RECORD_FILE_FULL	Insufficient space	Too many tables and attributes (increase MaxNoOfAttributes or MaxNoOfTables)
827	HA_ERR_RECORD_FILE_FULL	Insufficient space	Out of memory in Ndb Kernel, table data (increase DataMemory)
829	DMEC	Application error	Corrupt data received for insert/update
830	DMEC	Temporary Resource error	Out of add fragment operation records
831	DMEC	Application error	Too many nullable/bitfields in table definition
833	DMEC	Internal error	833
839	DMEC	Constraint violation	Illegal null attribute
840	DMEC	Constraint violation	Trying to set a NOT NULL attribute to NULL
850	DMEC	Application error	Too long or too short default value
851	DMEC	Application error	Maximum 8052 bytes of FIXED columns supported, use varchar or COLUMN_FORMAT DYNAMIC instead
871	DMEC	Internal error	871

NDB Error Code	MySQL Error Name	NDB Error Type	Error Message
873	DMEC	Temporary Resource error	Out of attrinfo records for scan in tuple manager
874	DMEC	Application error	Too much attrinfo (e.g. scan filter) for scan in tuple manager
876	DMEC	Application error	876
877	DMEC	Application error	877
878	DMEC	Application error	878
879	DMEC	Application error	879
880	DMEC	Application error	Tried to read too much - too many getValue calls
881	DMEC	Schema error	Unable to create table, out of data pages (increase DataMemory)
882	DMEC	Internal error	882
883	DMEC	Internal error	883
884	DMEC	Application error	Stack overflow in interpreter
885	DMEC	Application error	Stack underflow in interpreter
886	DMEC	Application error	More than 65535 instructions executed in interpreter
887	DMEC	Internal error	887
888	DMEC	Internal error	888
889	HA_ERR_RECORD_FILE_FULL	Insufficient space	Table fragment fixed data reference has reached maximum possible value (specify MAXROWS or increase no of partitions)
890	DMEC	Internal error	890
891	DMEC	Overload error	Data UNDO buffers overloaded (increase UndoDataBuffer)
892	DMEC	Application error	Unsupported type in scan filter
893	HA_ERR_FOUND_DUPP_KEY	Constraint violation	Constraint violation e.g. duplicate value in unique index
896	DMEC	Internal error	Tuple corrupted - wrong checksum or column data in invalid format

NDB Error Code	MySQL Error Name	NDB Error Type	Error Message
897	DMEC	Application error	Update attempt of primary key via ndbcluster internal api (if this occurs via the MySQL server it is a bug, please report)
899	DMEC	Temporary Resource error	Rowid already allocated
901	DMEC	Internal error	Inconsistent ordered index. The index needs to be dropped and recreated
902	HA_ERR_RECORD_FILE_FULL	Insufficient space	Out of memory in Ndb Kernel, ordered index data (increase DataMemory)
903	HA_ERR_INDEX_FILE_FULL	Insufficient space	Too many ordered indexes (increase MaxNoOfOrderedIndexes)
904	HA_ERR_INDEX_FILE_FULL	Insufficient space	Out of fragment records (increase MaxNoOfOrderedIndexes)
905	DMEC	Insufficient space	Out of attribute records (increase MaxNoOfAttributes)
906	DMEC	Schema error	Unsupported attribute type in index
907	DMEC	Schema error	Unsupported character set in table or index
908	DMEC	Insufficient space	Invalid ordered index tree node size
909	DMEC	Internal error	No free index scan op
910	HA_ERR_NO_SUCH_TABLE	Schema error	Index is being dropped
913	DMEC	Application error	Invalid index for index stats update
914	DMEC	Internal error	Invalid index stats request
915	DMEC	Temporary Resource error	No free index stats op
916	DMEC	Internal error	Invalid index stats sys tables
917	DMEC	Internal error	Invalid index stats sys tables data
918	DMEC	Temporary Resource error	Cannot prepare index stats update
919	DMEC	Temporary Resource error	Cannot execute index stats update
920	DMEC	Application error	Row operation defined after refreshTuple()

NDB Error Code	MySQL Error Name	NDB Error Type	Error Message
1204	DMEC	Node Recovery error	Temporary failure, distribution changed
1217	DMEC	Temporary Resource error	Out of operation records in local data manager (increase MaxNoOfLocalOperations)
1218	DMEC	Temporary Resource error	Send Buffers overloaded in NDB kernel
1220	DMEC	Temporary Resource error	REDO log files overloaded (increase FragmentLogFileSize)
1221	DMEC	Overload error	REDO buffers overloaded (increase RedoBuffer)
1222	DMEC	Temporary Resource error	Out of transaction markers in LQH
1223	DMEC	Node shutdown	Read operation aborted due to node shutdown
1224	HA_WRONG_CREATE_OPTION	Schema error	Too many fragments
1225	DMEC	Schema error	Table not defined in local query handler
1226	HA_ERR_NO_SUCH_TABLE	Schema error	Table is being dropped
1227	HA_WRONG_CREATE_OPTION	Schema error	Invalid schema version
1228	DMEC	Schema error	Cannot use drop table for drop index
1229	DMEC	Schema error	Too long frm data supplied
1231	DMEC	Schema error	Invalid table or index to scan
1232	DMEC	Schema error	Invalid table or index to scan
1233	DMEC	Application error	Table read-only
1234	DMEC	Temporary Resource error	REDO log files overloaded (increase disk hardware)
1300	DMEC	Internal error	Undefined error
1301	DMEC	Internal error	Backup issued to not master (reissue command to master)
1302	DMEC	Application error	A backup is already running

NDB Error Code	MySQL Error Name	NDB Error Type	Error Message
1303	DMEC	Insufficient space	Out of resources
1304	DMEC	Internal error	Sequence failure
1305	DMEC	Internal error	Backup definition not implemented
1306	DMEC	Application error	Backup not supported in diskless mode (change Diskless)
1321	DMEC	User defined error	Backup aborted by user request
1322	DMEC	Internal error	Backup already completed
1323	DMEC	Internal error	1323
1324	DMEC	Internal error	Backup log buffer full
1325	DMEC	Internal error	File or scan error
1326	DMEC	Internal error	Backup aborted due to node failure
1327	DMEC	Internal error	1327
1329	DMEC	Application error	Backup during software upgrade not supported
1340	DMEC	Internal error	Backup undefined error
1342	DMEC	Application error	Backup failed to allocate buffers (check configuration)
1343	DMEC	Application error	Backup failed to setup fs buffers (check configuration)
1344	DMEC	Application error	Backup failed to allocate tables (check configuration)
1345	DMEC	Application error	Backup failed to insert file header (check configuration)
1346	DMEC	Application error	Backup failed to insert table list (check configuration)
1347	DMEC	Application error	Backup failed to allocate table memory (check configuration)
1348	DMEC	Application error	Backup failed to allocate file record (check configuration)
1349	DMEC	Application error	Backup failed to allocate attribute record (check configuration)
1350	DMEC	Temporary Resource error	Backup failed: file already exists (use 'START BACKUP <backup id>')

NDB Error Code	MySQL Error Name	NDB Error Type	Error Message
1405	DMEC	Node Recovery error	Subscriber manager busy with node recovery
1407	DMEC	Schema error	Subscription not found in subscriber manager
1411	DMEC	Temporary Resource error	Subscriber manager busy with adding/removing a subscriber
1412	DMEC	Insufficient space	Can't accept more subscribers, out of space in pool
1413	DMEC	Temporary Resource error	Subscriber manager busy with adding the subscription
1414	DMEC	Temporary Resource error	Subscriber manager has subscribers on this subscription
1415	DMEC	Schema error	Subscription not unique in subscriber manager
1416	DMEC	Insufficient space	Can't accept more subscriptions, out of space in pool
1417	DMEC	Schema error	Table in suscription not defined, probably dropped
1418	DMEC	Schema error	Subscription dropped, no new subscribers allowed
1419	DMEC	Schema error	Subscription already dropped
1420	DMEC	Temporary Resource error	Subscriber manager busy with adding/removing a table
1421	DMEC	Schema error	Partially connected API in NdbOperation::execute()
1422	DMEC	Schema error	Out of subscription records
1423	DMEC	Schema error	Out of table records in SUMA
1424	DMEC	Schema error	Out of MaxNoOfConcurrentSubOperations
1425	DMEC	Schema error	Subscription being defined...while trying to stop subscriber
1426	DMEC	Schema error	No such subscriber
1427	DMEC	Node Recovery error	Api node died, when SUB_START_REQ reached node
1428	DMEC	Internal error	No replica to scan on this node (internal index stats error)

NDB Error Code	MySQL Error Name	NDB Error Type	Error Message
1429	DMEC	Internal error	Subscriber node undefined in SubStartReq (config change?)
1501	DMEC	Temporary Resource error	Out of undo space
1502	DMEC	Internal error	Filegroup already exists
1503	DMEC	Schema error	Out of filegroup records
1504	DMEC	Schema error	Out of logbuffer memory(specify smaller undo_buffer_size or increase SharedGlobalMemory)
1505	DMEC	Internal error	Invalid filegroup
1506	DMEC	Internal error	Invalid filegroup version
1507	DMEC	Internal error	File no already inuse
1508	DMEC	Schema error	Out of file records
1509	DMEC	Schema error	File system error, check if path,permissions etc
1510	DMEC	Internal error	File meta data error
1511	DMEC	Internal error	Out of memory
1512	DMEC	Schema error	File read error
1513	DMEC	Internal error	Filegroup not online
1514	DMEC	Schema error	Currently there is a limit of one logfile group
1515	DMEC	Schema error	Currently there is a 4G limit of one undo/data-file in 32-bit host
1516	DMEC	Schema error	File too small
1601	HA_ERR_RECORD_FILE_FULL	Insufficient space	Out extents, tablespace full
1602	DMEC	Insufficient space	No datafile in tablespace
1603	HA_ERR_RECORD_FILE_FULL	Insufficient space	Table fragment fixed data reference has reached maximum possible value (specify MAXROWS or increase no of partitions)
1700	DMEC	Internal error	Undefined error

NDB Error Code	MySQL Error Name	NDB Error Type	Error Message
1701	DMEC	Application error	Node already reserved
1702	DMEC	Application error	Node already connected
1703	DMEC	Internal temporary	Node failure handling not completed
1704	DMEC	Application error	Node type mismatch
2810	DMEC	Temporary Resource error	No space left on the device
2811	DMEC	Temporary Resource error	Error with file permissions, please check file system
2815	DMEC	Temporary Resource error	Error in reading files, please check file system
4000	DMEC	Internal error	MEMORY ALLOCATION ERROR
4001	DMEC	Internal error	Signal Definition Error
4002	DMEC	Node Recovery error	Send to NDB failed
4003	DMEC	Function not implemented	Function not implemented yet
4004	DMEC	Application error	Attribute name or id not found in the table
4005	DMEC	Internal error	Internal Error in NdbApi
4006	DMEC	Overload error	Connect failure - out of connection objects (increase MaxNoOfConcurrentTransactions)
4007	DMEC	Node Recovery error	Send to ndbd node failed
4008	DMEC	Unknown result error	Receive from NDB failed
4009	HA_ERR_NO_CONNECTION	Unknown result error	Cluster Failure
4010	DMEC	Node Recovery error	Node failure caused abort of transaction

NDB Error Code	MySQL Error Name	NDB Error Type	Error Message
4011	DMEC	Internal error	Internal Error in NdbApi
4012	DMEC	Unknown result error	Request ndbd time-out, maybe due to high load or communication problems
4013	DMEC	Node Recovery error	Request timed out in waiting for node failure
4021	DMEC	Temporary Resource error	Out of Send Buffer space in NDB API
4022	DMEC	Temporary Resource error	Out of Send Buffer space in NDB API
4023	DMEC	Node shutdown	Transaction aborted due to node shutdown
4025	DMEC	Node Recovery error	Node failure caused abort of transaction
4027	DMEC	Node Recovery error	Node failure caused abort of transaction
4028	DMEC	Node Recovery error	Node failure caused abort of transaction
4029	DMEC	Node Recovery error	Node failure caused abort of transaction
4030	DMEC	Node shutdown	Transaction aborted due to node shutdown
4031	DMEC	Node Recovery error	Node failure caused abort of transaction
4032	DMEC	Temporary Resource error	Out of Send Buffer space in NDB API
4033	DMEC	Node Recovery error	Send to NDB failed
4034	DMEC	Node shutdown	Transaction aborted due to node shutdown
4035	DMEC	Node Recovery error	Cluster temporary unavailable
4100	DMEC	Application error	Status Error in NDB
4101	DMEC	Application error	No connections to NDB available and connect failed

NDB Error Code	MySQL Error Name	NDB Error Type	Error Message
4102	DMEC	Application error	Type in NdbTamper not correct
4103	DMEC	Application error	No schema connections to NDB available and connect failed
4104	DMEC	Application error	Ndb Init in wrong state, destroy Ndb object and create a new
4105	DMEC	Application error	Too many Ndb objects
4106	DMEC	Application error	All Not NULL attribute have not been defined
4107	DMEC	Internal error	Simple Transaction and Not Start
4108	DMEC	Internal error	Faulty operation type
4109	DMEC	Internal error	Faulty primary key attribute length
4110	DMEC	Internal error	Faulty length in ATTRINFO signal
4111	DMEC	Internal error	Status Error in NdbConnection
4113	DMEC	Internal error	Too many operations received
4114	DMEC	Application error	Transaction is already completed
4115	DMEC	Node Recovery error	Transaction was committed but all read information was not received due to node crash
4116	DMEC	Application error	Operation was not defined correctly, probably missing a key
4117	DMEC	Application error	Could not start transporter, configuration error
4118	DMEC	Application error	Parameter error in API call
4119	DMEC	Node Recovery error	Simple/dirty read failed due to node failure
4120	DMEC	Application error	Scan already complete
4121	DMEC	Application error	Cannot set name twice for an Ndb object
4122	DMEC	Application error	Cannot set name after Ndb object is initialised
4123	DMEC	Application error	Free percent out of range. Allowed range is 1-99
4200	DMEC	Application error	Status Error when defining an operation

NDB Error Code	MySQL Error Name	NDB Error Type	Error Message
4201	DMEC	Application error	Variable Arrays not yet supported
4202	DMEC	Application error	Set value on tuple key attribute is not allowed
4203	DMEC	Application error	Trying to set a NOT NULL attribute to NULL
4204	DMEC	Application error	Set value and Read/Delete Tuple is incompatible
4205	DMEC	Application error	No Key attribute used to define tuple
4206	DMEC	Application error	Not allowed to equal key attribute twice
4207	DMEC	Application error	Key size is limited to 4092 bytes
4208	DMEC	Application error	Trying to read a non-stored attribute
4209	DMEC	Application error	Length parameter in equal/setValue is incorrect
4210	DMEC	Application error	Ndb sent more info than the length he specified
4211	DMEC	Application error	Inconsistency in list of NdbRecAttr-objects
4212	DMEC	Application error	Ndb reports NULL value on Not NULL attribute
4213	DMEC	Application error	Not all data of an attribute has been received
4214	DMEC	Application error	Not all attributes have been received
4215	DMEC	Application error	More data received than reported in TCKEYCONF message
4216	DMEC	Application error	More than 8052 bytes in setValue cannot be handled
4217	DMEC	Application error	It is not allowed to increment any other than unsigned ints
4218	DMEC	Application error	Currently not allowed to increment NULL-able attributes
4219	DMEC	Application error	Maximum size of interpretative attributes are 64 bits
4220	DMEC	Application error	Maximum size of interpretative attributes are 64 bits
4221	DMEC	Application error	Trying to jump to a non-defined label
4222	DMEC	Application error	Label was not found, internal error
4223	DMEC	Application error	Not allowed to create jumps to yourself

NDB Error Code	MySQL Error Name	NDB Error Type	Error Message
4224	DMEC	Application error	Not allowed to jump to a label in a different subroutine
4225	DMEC	Application error	All primary keys defined, call setValue/ getValue
4226	DMEC	Application error	Bad number when defining a label
4227	DMEC	Application error	Bad number when defining a subroutine
4228	DMEC	Application error	Illegal interpreter function in scan definition
4229	DMEC	Application error	Illegal register in interpreter function definition
4230	DMEC	Application error	Illegal state when calling getValue, probably not a read
4231	DMEC	Application error	Illegal state when calling interpreter routine
4232	DMEC	Application error	Parallelism can only be between 1 and 240
4233	DMEC	Application error	Calling execute (synchronous) when already prepared asynchronous transaction exists
4234	DMEC	Application error	Illegal to call setValue in this state
4235	DMEC	Application error	No callback from execute
4236	DMEC	Application error	Trigger name too long
4237	DMEC	Application error	Too many triggers
4238	DMEC	Application error	Trigger not found
4239	DMEC	Application error	Trigger with given name already exists
4240	DMEC	Application error	Unsupported trigger type
4241	DMEC	Application error	Index name too long
4242	DMEC	Application error	Too many indexes
4243	DMEC	Application error	Index not found
4244	HA_ERR_TABLE_EXIST	Schema object already exists	Index or table with given name already exists

NDB Error Code	MySQL Error Name	NDB Error Type	Error Message
4247	DMEC	Application error	Illegal index/trigger create/drop/alter request
4248	DMEC	Application error	Trigger/index name invalid
4249	DMEC	Application error	Invalid table
4250	DMEC	Application error	Invalid index type or index logging option
4251	HA_ERR_FOUND_DUPP_UNIQUE	Application error	Cannot create unique index, duplicate keys found
4252	DMEC	Application error	Failed to allocate space for index
4253	DMEC	Application error	Failed to create index table
4254	DMEC	Application error	Table not an index table
4255	DMEC	Application error	Hash index attributes must be specified in same order as table attributes
4256	DMEC	Application error	Must call Ndb::init() before this function
4257	DMEC	Application error	Tried to read too much - too many getValue calls
4258	DMEC	Application error	Cannot create unique index, duplicate attributes found in definition
4259	DMEC	Application error	Invalid set of range scan bounds
4260	DMEC	User defined error	NdbScanFilter: Operator is not defined in NdbScanFilter::Group
4261	DMEC	User defined error	NdbScanFilter: Column is NULL
4262	DMEC	User defined error	NdbScanFilter: Condition is out of bounds
4263	DMEC	Internal error	Invalid blob attributes or invalid blob parts table
4264	DMEC	Application error	Invalid usage of blob attribute
4265	DMEC	Application error	The method is not valid in current blob state
4266	DMEC	Application error	Invalid blob seek position
4267	DMEC	Internal error	Corrupted blob value

NDB Error Code	MySQL Error Name	NDB Error Type	Error Message
4268	DMEC	Internal error	Error in blob head update forced rollback of transaction
4269	DMEC	Internal error	No connection to ndb management server
4270	DMEC	Internal error	Unknown blob error
4271	DMEC	Application error	Invalid index object, not retrieved via getIndex()
4272	DMEC	Application error	Table definition has undefined column
4273	DMEC	Internal error	No blob table in dict cache
4274	DMEC	Internal error	Corrupted main table PK in blob operation
4275	DMEC	Application error	The blob method is incompatible with operation type or lock mode
4276	DMEC	Application error	Missing NULL ptr in end of keyData list
4277	DMEC	Application error	Key part len is to small for column
4278	DMEC	Application error	Supplied buffer to small
4279	DMEC	Application error	Malformed string
4280	DMEC	Application error	Inconsistent key part length
4281	DMEC	Application error	Too many keys specified for key bound in scanIndex
4282	DMEC	Application error	range_no not strictly increasing in ordered multi-range index scan
4283	DMEC	Application error	key_record in index scan is not an index ndbrecord
4284	DMEC	Application error	Cannot mix NdbRecAttr and NdbRecord methods in one operation
4285	DMEC	Application error	NULL NdbRecord pointer
4286	DMEC	Application error	Invalid range_no (must be < 4096)
4287	DMEC	Application error	The key_record and attribute_record in primary key operation do not belong to the same table
4288	DMEC	Application error	Blob handle for column not available
4289	DMEC	Application error	API version mismatch or wrong sizeof(NdbDictionary::RecordSpecification)

NDB Error Code	MySQL Error Name	NDB Error Type	Error Message
4290	DMEC	Application error	Missing column specification in NdbDictionary::RecordSpecification
4291	DMEC	Application error	Duplicate column specification in NdbDictionary::RecordSpecification
4292	DMEC	Application error	NdbRecord for tuple access is not an index key NdbRecord
4293	DMEC	Application error	Error returned from application scanIndex() callback
4294	DMEC	Application error	Scan filter is too large, discarded
4295	DMEC	Application error	Column is NULL in Get/SetValueSpec structure
4296	DMEC	Application error	Invalid AbortOption
4297	DMEC	Application error	Invalid or unsupported OperationOptions structure
4298	DMEC	Application error	Invalid or unsupported ScanOptions structure
4299	DMEC	Application error	Incorrect combination of ScanOption flags, extraGetValues ptr and numExtraGetValues
4300	DMEC	Application error	Tuple Key Type not correct
4301	DMEC	Application error	Fragment Type not correct
4302	DMEC	Application error	Minimum Load Factor not correct
4303	DMEC	Application error	Maximum Load Factor not correct
4304	DMEC	Application error	Maximum Load Factor smaller than Minimum
4305	DMEC	Application error	K value must currently be set to 6
4306	DMEC	Application error	Memory Type not correct
4307	DMEC	Application error	Invalid table name
4308	DMEC	Application error	Attribute Size not correct
4309	DMEC	Application error	Fixed array too large, maximum 64000 bytes
4310	DMEC	Application error	Attribute Type not correct
4311	DMEC	Application error	Storage Mode not correct

NDB Error Code	MySQL Error Name	NDB Error Type	Error Message
4312	DMEC	Application error	Null Attribute Type not correct
4313	DMEC	Application error	Index only storage for non-key attribute
4314	DMEC	Application error	Storage Type of attribute not correct
4315	DMEC	Application error	No more key attributes allowed after defining variable length key attribute
4316	DMEC	Application error	Key attributes are not allowed to be NULL attributes
4317	DMEC	Application error	Too many primary keys defined in table
4318	DMEC	Application error	Invalid attribute name or number
4319	DMEC	Application error	createAttribute called at erroneus place
4320	DMEC	Internal error	Cannot use the same object twice to create table
4321	DMEC	Internal error	Trying to start two schema transactions
4322	DMEC	Application error	Attempt to define distribution key when not prepared to
4323	DMEC	Application error	Distribution Key set on table but not defined on first attribute
4324	DMEC	Application error	Attempt to define distribution group when not prepared to
4325	DMEC	Application error	Distribution Group set on table but not defined on first attribute
4326	DMEC	Application error	Distribution Group with erroneus number of bits
4327	DMEC	Application error	Distribution key is only supported on part of primary key
4328	DMEC	Application error	Disk memory attributes not yet supported
4329	DMEC	Application error	Variable stored attributes not yet supported
4335	DMEC	Application error	Only one autoincrement column allowed per table. Having a table without primary key uses an autoincremented hidden key, i.e. a table without a primary key can not have an autoincremented column
4340	DMEC	Application error	Result or attribute record must be a base table ndbrecord, not an index ndbrecord

NDB Error Code	MySQL Error Name	NDB Error Type	Error Message
4341	DMEC	Application error	Not all keys read when using option SF_OrderBy
4344	DMEC	Internal error	Only DBDICT and TRIX can send requests to TRIX
4345	DMEC	Internal error	TRIX block is not available yet, probably due to node failure
4346	DMEC	Internal error	Internal error at index create/build
4347	DMEC	Internal error	Bad state at alter index
4348	DMEC	Internal error	Inconsistency detected at alter index
4349	DMEC	Internal error	Inconsistency detected at index usage
4350	DMEC	Internal error	Transaction already aborted
4351	DMEC	Timeout expired	Timeout/deadlock during index build
4400	DMEC	Application error	Status Error in NdbSchemaCon
4401	DMEC	Application error	Only one schema operation per schema transaction
4402	DMEC	Application error	No schema operation defined before calling execute
4410	DMEC	Application error	Schema transaction is already started
4411	DMEC	Application error	Schema transaction not possible until upgrade complete
4501	DMEC	Application error	Insert in hash table failed when getting table information from Ndb
4502	DMEC	Application error	GetValue not allowed in Update operation
4503	DMEC	Application error	GetValue not allowed in Insert operation
4504	DMEC	Application error	SetValue not allowed in Read operation
4505	DMEC	Application error	NULL value not allowed in primary key search
4506	DMEC	Application error	Missing getValue/setValue when calling execute
4507	DMEC	Application error	Missing operation request when calling execute
4508	DMEC	Application error	GetValue not allowed for NdbRecord defined operation
4509	DMEC	Application error	Non SF_MultiRange scan cannot have more than one bound

NDB Error Code	MySQL Error Name	NDB Error Type	Error Message
4510	DMEC	Application error	User specified partition id not allowed for scan takeover operation
4511	DMEC	Application error	Blobs not allowed in NdbRecord delete result record
4512	DMEC	Application error	Incorrect combination of OperationOptions optionsPresent, extraGet/SetValues ptr and numExtraGet/SetValues
4513	DMEC	Application error	Only one scan bound allowed for non-NdbRecord setBound() API
4514	DMEC	Application error	Can only call setBound/equal() for an NdbIndexScanOperation
4515	DMEC	Application error	Method not allowed for NdbRecord, use OperationOptions or ScanOptions structure instead
4516	DMEC	Application error	Illegal instruction in interpreted program
4517	DMEC	Application error	Bad label in branch instruction
4518	DMEC	Application error	Too many instructions in interpreted program
4519	DMEC	Application error	NdbInterpretedCode::finalise() not called
4520	DMEC	Application error	Call to undefined subroutine
4521	DMEC	Application error	Call to undefined subroutine, internal error
4522	DMEC	Application error	setBound() called twice for same key
4523	DMEC	Application error	Pseudo columns not supported by NdbRecord
4524	DMEC	Application error	NdbInterpretedCode is for different table
4535	DMEC	Application error	Attempt to set bound on non key column
4536	DMEC	Application error	NdbScanFilter constructor taking NdbOperation is not supported for NdbRecord
4537	DMEC	Application error	Wrong API. Use NdbInterpretedCode for NdbRecord operations
4538	DMEC	Application error	NdbInterpretedCode instruction requires that table is set
4539	DMEC	Application error	NdbInterpretedCode not supported for operation type

NDB Error Code	MySQL Error Name	NDB Error Type	Error Message
4540	DMEC	Application error	Attempt to pass an Index column to createRecord. Use base table columns only
4542	DMEC	Application error	Unknown partition information type
4543	DMEC	Application error	Duplicate partitioning information supplied
4544	DMEC	Application error	Wrong partitionInfo type for table
4545	DMEC	Application error	Invalid or Unsupported PartitionInfo structure
4546	DMEC	Application error	Explicit partitioning info not allowed for table and operation
4547	DMEC	Application error	RecordSpecification has overlapping offsets
4548	DMEC	Application error	RecordSpecification has too many elements
4549	DMEC	Application error	getLockHandle only supported for primary key read with a lock
4550	DMEC	Application error	Cannot releaseLockHandle until operation executed
4551	DMEC	Application error	NdbLockHandle already released
4552	DMEC	Application error	NdbLockHandle does not belong to transaction
4553	DMEC	Application error	NdbLockHandle original operation not executed successfully
4554	DMEC	Application error	NdbBlob can only be closed from Active state
4555	DMEC	Application error	NdbBlob cannot be closed with pending operations
4556	DMEC	Application error	RecordSpecification has illegal value in column_flags
4600	DMEC	Application error	Transaction is already started
4601	DMEC	Application error	Transaction is not started
4602	DMEC	Application error	You must call getNdbOperation before executeScan
4603	DMEC	Application error	There can only be ONE operation in a scan transaction
4604	DMEC	Application error	takeOverScanOp, to take over a scanned row one must explicitly request keyinfo on readTuples call
4605	DMEC	Application error	You may only call readTuples() once for each operation

NDB Error Code	MySQL Error Name	NDB Error Type	Error Message
4607	DMEC	Application error	There may only be one operation in a scan transaction
4608	DMEC	Application error	You can not takeOverScan unless you have used openScanExclusive
4609	DMEC	Application error	You must call nextScanResult before trying to takeOverScan
4707	DMEC	Application error	Too many event have been defined
4708	DMEC	Application error	Event name is too long
4709	DMEC	Application error	Can't accept more subscribers
4710	DMEC	Application error	Event not found
4711	DMEC	Application error	Creation of event failed
4712	DMEC	Application error	Stopped event operation does not exist. Already stopped?
4713	DMEC	Schema error	Column defined in event does not exist in table
4714	DMEC	Application error	Index stats sys tables NDB_INDEX_STAT_PREFIX do not exist
4715	DMEC	Application error	Index stats for specified index do not exist
4716	DMEC	Application error	Index stats methods usage error
4717	DMEC	Application error	Index stats cannot allocate memory
4718	DMEC	Internal error	Index stats samples data or memory cache is invalid
4719	DMEC	Internal error	Index stats internal error
4720	DMEC	Application error	Index stats sys tables NDB_INDEX_STAT_PREFIX partly missing or invalid
4721	DMEC	Internal error	Mysqld: index stats thread not open for requests
4722	DMEC	Internal error	Mysqld: index stats entry unexpectedly not found
4723	DMEC	Application error	Mysqld: index stats request ignored due to recent error
4724	DMEC	Application error	Mysqld: index stats request aborted by stats thread
4725	DMEC	Application error	Index stats were deleted by another process

NDB Error Code	MySQL Error Name	NDB Error Type	Error Message
4731	DMEC	Internal error	Event not found
20000	DMEC	Temporary Resource error	Query aborted due out of operation records
20001	DMEC	Internal error	Query aborted due to empty query tree
20002	DMEC	Internal error	Query aborted due to invalid request
20003	DMEC	Internal error	Query aborted due to unknown query operation
20004	DMEC	Internal error	Query aborted due to invalid tree node specification
20005	DMEC	Internal error	Query aborted due to invalid tree parameter specification
20006	DMEC	Temporary Resource error	Query aborted due to out of LongMessageBuffer
20007	DMEC	Internal error	Query aborted due to invalid pattern
20008	DMEC	Temporary Resource error	Query aborted due to out of query memory
20009	DMEC	Internal error	Query aborted due to query node too big
20010	DMEC	Internal error	Query aborted due to query node parameters too big
20011	DMEC	Internal error	Query aborted due to both tree and parameters contain interpreted program
20012	DMEC	Internal error	Query aborted due to invalid tree parameter specification: Key parameter bits mismatch
20013	DMEC	Internal error	Query aborted due to invalid tree parameter specification: Incorrect key parameter count
20014	DMEC	Internal error	Query aborted due to internal error
20015	DMEC	Temporary Resource error	Query aborted due to out of row memory
20016	DMEC	Node Recovery error	Query aborted due to node failure
20017	DMEC	Internal error	Query aborted due to invalid node count
20018	DMEC	Internal error	Query aborted due to index fragment not found

NDB Error Code	MySQL Error Name	NDB Error Type	Error Message
20019	HA_ERR_NO_SUCH_TABLE	Schema error	Query table not defined
20020	HA_ERR_NO_SUCH_TABLE	Schema error	Query table is being dropped
20021	HA_ERR_TABLE_DEF_CHANGED	Schema error	Query table definition has changed
21000	HA_ERR_CANNOT_ADD_FOREIGN	Application error	Create foreign key failed - parent key is primary key and on-update-cascade is not allowed
21020	DMEC	Temporary Resource error	Create foreign key failed in NDB - no more object records
21021	DMEC	Internal error	Create foreign key failed in NDB - invalid request
21022	DMEC	Schema error	Create foreign key failed in NDB - parent table is not table
21023	DMEC	Schema error	Create foreign key failed in NDB - invalid parent table version
21024	DMEC	Schema error	Create foreign key failed in NDB - child table is not table
21025	DMEC	Schema error	Create foreign key failed in NDB - invalid child table version
21026	HA_ERR_CANNOT_ADD_FOREIGN	Application error	Create foreign key failed in NDB - parent index is not unique index
21027	DMEC	Schema error	Create foreign key failed in NDB - invalid parent index version
21028	DMEC	Schema error	Create foreign key failed in NDB - child index is not index
21029	DMEC	Schema error	Create foreign key failed in NDB - invalid child index version
21030	DMEC	Internal error	Create foreign key failed in NDB - object already exists in TC
21031	DMEC	Internal error	Create foreign key failed in NDB - no more object records in TC
21032	DMEC	Internal error	Create foreign key failed in NDB - invalid request to TC
21040	DMEC	Application error	Drop foreign key failed in NDB - foreign key not found
21041	DMEC	Schema error	Drop foreign key failed in NDB - invalid foreign key version
21042	DMEC	Schema error	Drop foreign key failed in NDB - foreign key not found in TC
21060	DMEC	Application error	Build foreign key failed in NDB - foreign key not found
21061	DMEC	Schema error	Build foreign key failed in NDB - invalid foreign key version

NDB Error Code	MySQL Error Name	NDB Error Type	Error Message
21080	HA_ERR_ROW_IS_REFERENCED	Schema error	Drop table not allowed in NDB - referenced by foreign key on another table
21081	HA_ERR_DROP_INDEX_FK	Application error	Drop index not allowed in NDB - used as parent index of a foreign key
21082	HA_ERR_DROP_INDEX_FK	Application error	Drop index not allowed in NDB - used as child index of a foreign key
21090	HA_ERR_CANNOT_ADD_FOREIGN	Application error	Create foreign key failed in NDB - name contains invalid character (/)

2.4.4 NDB Error Classifications

The following table lists the classification codes used for NDB API errors, and their descriptions. These can also be found in the file `/storage/ndb/src/ndbapi/ndberror.c`.

Classification Code	Error Status	Description
NE	*Success*	No error
AE	*Permanent error*	Application error
CE	*Permanent error*	Configuration or application error
ND	*Permanent error*	No data found
CV	*Permanent error*	Constraint violation
SE	*Permanent error*	Schema error
OE	*Permanent error*	Schema object already exists
UD	*Permanent error*	User defined error
IS	*Permanent error*	Insufficient space
TR	*Temporary error*	Temporary Resource error
NR	*Temporary error*	Node Recovery error
OL	*Temporary error*	Overload error
TO	*Temporary error*	Timeout expired
NS	*Temporary error*	Node shutdown
IT	*Temporary error*	Internal temporary
UR	*Unknown result*	Unknown result error
UE	*Unknown result*	Unknown error code
IE	*Permanent error*	Internal error
NI	*Permanent error*	Function not implemented
DMEC	*Default MySQL error code*	Used for NDB errors that are not otherwise mapped to MySQL error codes

2.5 NDB API Examples

This section provides code examples illustrating how to accomplish some basic tasks using the NDB API.

All of these examples can be compiled and run as provided, and produce sample output to demonstrate their effects.

Note

For an NDB API program to connect to the cluster, the cluster configuration file must have at least one `[api]` section that is not assigned to an SQL node and that can be accessed from the host where the NDB API application runs. You can also use an unassigned `[mysqld]` section for this purpose, although we recommend that you use [mysqld] sections for SQL nodes and `[api]` sections for NDB client programs. See NDB Cluster Configuration Files, and especially Defining SQL and Other API Nodes in an NDB Cluster, for more information.

2.5.1 NDB API Example Using Synchronous Transactions

This example illustrates the use of synchronous transactions in the NDB API. It first creates a database `ndb_examples` and a table `api_simple` (if these objects do not already exist) using the MySQL C API with an SQL node, then performs a series of basic data operations (insert, update, read, and select) on this table using the NDB API.

The compiled program takes two arguments:

1. The path to a MySQL socket file (`mysqld --socket` option)
2. An NDB Cluster connection string (see NDB Cluster Connection Strings)

The correct output from this program is as follows:

```
ATTR1  ATTR2
0      10
1       1
2      12
Detected that deleted tuple doesn't exist!
4      14
5       5
6      16
7       7
8      18
9       9
```

The source code for this example can be found in `storage/ndb/ndbapi-examples/ndbapi_simple/ndbapi_simple.cpp` in the NDB Cluster source tree, and is reproduced here:

```
/*
 *  ndbapi_simple.cpp: Using synchronous transactions in NDB API
 *
 *  Correct output from this program is:
 *
 *  ATTR1 ATTR2
 *    0    10
 *    1     1
 *    2    12
 *  Detected that deleted tuple doesn't exist!
 *    4    14
 *    5     5
 *    6    16
 *    7     7
 *    8    18
 *    9     9
 *
 */

#include <mysql.h>
#include <mysqld_error.h>
#include <NdbApi.hpp>
// Used for cout
#include <stdio.h>
#include <iostream>
```

```
static void run_application(MYSQL &, Ndb_cluster_connection &);

#define PRINT_ERROR(code,msg) \
  std::cout << "Error in " << __FILE__ << ", line: " << __LINE__ \
            << ", code: " << code \
            << ", msg: " << msg << "." << std::endl
#define MYSQLERROR(mysql) { \
  PRINT_ERROR(mysql_errno(&mysql),mysql_error(&mysql)); \
  exit(-1); }
#define APIERROR(error) { \
  PRINT_ERROR(error.code,error.message); \
  exit(-1); }

int main(int argc, char** argv)
{
  if (argc != 3)
  {
    std::cout << "Arguments are <socket mysqld> <connect_string cluster>.\n";
    exit(-1);
  }
  // ndb_init must be called first
  ndb_init();

  // connect to mysql server and cluster and run application
  {
    char * mysqld_sock  = argv[1];
    const char *connection_string = argv[2];
    // Object representing the cluster
    Ndb_cluster_connection cluster_connection(connection_string);

    // Connect to cluster management server (ndb_mgmd)
    if (cluster_connection.connect(4 /* retries               */,
       5 /* delay between retries */,
       1 /* verbose               */))
    {
      std::cout << "Cluster management server was not ready within 30 secs.\n";
      exit(-1);
    }

    // Optionally connect and wait for the storage nodes (ndbd's)
    if (cluster_connection.wait_until_ready(30,0) < 0)
    {
      std::cout << "Cluster was not ready within 30 secs.\n";
      exit(-1);
    }

    // connect to mysql server
    MYSQL mysql;
    if ( !mysql_init(&mysql) ) {
      std::cout << "mysql_init failed\n";
      exit(-1);
    }
    if ( !mysql_real_connect(&mysql, "localhost", "root", "", "",
        0, mysqld_sock, 0) )
      MYSQLERROR(mysql);

    // run the application code
    run_application(mysql, cluster_connection);
  }

  ndb_end(0);

  return 0;
}

static void create_table(MYSQL &);
static void do_insert(Ndb &);
static void do_update(Ndb &);
static void do_delete(Ndb &);
static void do_read(Ndb &);

static void run_application(MYSQL &mysql,
```

```
      Ndb_cluster_connection &cluster_connection)
{
  /****************************************
   * Connect to database via mysql-c          *ndb_examples
   ****************************************/
  mysql_query(&mysql, "CREATE DATABASE ndb_examples");
  if (mysql_query(&mysql, "USE ndb_examples") != 0) MYSQLERROR(mysql);
  create_table(mysql);

  /****************************************
   * Connect to database via NDB API           *
   ****************************************/
  // Object representing the database
  Ndb myNdb( &cluster_connection, "ndb_examples" );
  if (myNdb.init()) APIERROR(myNdb.getNdbError());

  /*
   * Do different operations on database
   */
  do_insert(myNdb);
  do_update(myNdb);
  do_delete(myNdb);
  do_read(myNdb);
}

/*********************************************************
 * Create a table named api_simple if it does not exist *
 *********************************************************/
static void create_table(MYSQL &mysql)
{
  while (mysql_query(&mysql,
    "CREATE TABLE"
    "  api_simple"
    "    (ATTR1 INT UNSIGNED NOT NULL PRIMARY KEY,"
    "     ATTR2 INT UNSIGNED NOT NULL)"
    "  ENGINE=NDB"))
  {
    if (mysql_errno(&mysql) == ER_TABLE_EXISTS_ERROR)
    {
      std::cout << "NDB Cluster already has example table: api_simple. "
      << "Dropping it..." << std::endl;
      mysql_query(&mysql, "DROP TABLE api_simple");
    }
    else MYSQLERROR(mysql);
  }
}

/*************************************************************************
 * Using 5 transactions, insert 10 tuples in table: (0,0),(1,1),...,(9,9) *
 *************************************************************************/
static void do_insert(Ndb &myNdb)
{
  const NdbDictionary::Dictionary* myDict= myNdb.getDictionary();
  const NdbDictionary::Table *myTable= myDict->getTable("api_simple");

  if (myTable == NULL)
    APIERROR(myDict->getNdbError());

  for (int i = 0; i < 5; i++) {
    NdbTransaction *myTransaction= myNdb.startTransaction();
    if (myTransaction == NULL) APIERROR(myNdb.getNdbError());

    NdbOperation *myOperation= myTransaction->getNdbOperation(myTable);
    if (myOperation == NULL) APIERROR(myTransaction->getNdbError());

    myOperation->insertTuple();
    myOperation->equal("ATTR1", i);
    myOperation->setValue("ATTR2", i);

    myOperation= myTransaction->getNdbOperation(myTable);
    if (myOperation == NULL) APIERROR(myTransaction->getNdbError());
```

```
    myOperation->insertTuple();
    myOperation->equal("ATTR1", i+5);
    myOperation->setValue("ATTR2", i+5);

    if (myTransaction->execute( NdbTransaction::Commit ) == -1)
      APIERROR(myTransaction->getNdbError());

    myNdb.closeTransaction(myTransaction);
  }
}

/*****************************************************************
 * Update the second attribute in half of the tuples (adding 10) *
 *****************************************************************/
static void do_update(Ndb &myNdb)
{
  const NdbDictionary::Dictionary* myDict= myNdb.getDictionary();
  const NdbDictionary::Table *myTable= myDict->getTable("api_simple");

  if (myTable == NULL)
    APIERROR(myDict->getNdbError());

  for (int i = 0; i < 10; i+=2) {
    NdbTransaction *myTransaction= myNdb.startTransaction();
    if (myTransaction == NULL) APIERROR(myNdb.getNdbError());

    NdbOperation *myOperation= myTransaction->getNdbOperation(myTable);
    if (myOperation == NULL) APIERROR(myTransaction->getNdbError());

    myOperation->updateTuple();
    myOperation->equal( "ATTR1", i );
    myOperation->setValue( "ATTR2", i+10);

    if( myTransaction->execute( NdbTransaction::Commit ) == -1 )
      APIERROR(myTransaction->getNdbError());

    myNdb.closeTransaction(myTransaction);
  }
}

/*********************************************
 * Delete one tuple (the one with primary key 3) *
 *********************************************/
static void do_delete(Ndb &myNdb)
{
  const NdbDictionary::Dictionary* myDict= myNdb.getDictionary();
  const NdbDictionary::Table *myTable= myDict->getTable("api_simple");

  if (myTable == NULL)
    APIERROR(myDict->getNdbError());

  NdbTransaction *myTransaction= myNdb.startTransaction();
  if (myTransaction == NULL) APIERROR(myNdb.getNdbError());

  NdbOperation *myOperation= myTransaction->getNdbOperation(myTable);
  if (myOperation == NULL) APIERROR(myTransaction->getNdbError());

  myOperation->deleteTuple();
  myOperation->equal( "ATTR1", 3 );

  if (myTransaction->execute(NdbTransaction::Commit) == -1)
    APIERROR(myTransaction->getNdbError());

  myNdb.closeTransaction(myTransaction);
}

/*****************************
 * Read and print all tuples *
 *****************************/
static void do_read(Ndb &myNdb)
{
  const NdbDictionary::Dictionary* myDict= myNdb.getDictionary();
```

```
  const NdbDictionary::Table *myTable= myDict->getTable("api_simple");

  if (myTable == NULL)
    APIERROR(myDict->getNdbError());

  std::cout << "ATTR1 ATTR2" << std::endl;

  for (int i = 0; i < 10; i++) {
    NdbTransaction *myTransaction= myNdb.startTransaction();
    if (myTransaction == NULL) APIERROR(myNdb.getNdbError());

    NdbOperation *myOperation= myTransaction->getNdbOperation(myTable);
    if (myOperation == NULL) APIERROR(myTransaction->getNdbError());

    myOperation->readTuple(NdbOperation::LM_Read);
    myOperation->equal("ATTR1", i);

    NdbRecAttr *myRecAttr= myOperation->getValue("ATTR2", NULL);
    if (myRecAttr == NULL) APIERROR(myTransaction->getNdbError());

    if(myTransaction->execute( NdbTransaction::Commit ) == -1)
      APIERROR(myTransaction->getNdbError());

    if (myTransaction->getNdbError().classification == NdbError::NoDataFound)
      if (i == 3)
        std::cout << "Detected that deleted tuple doesn't exist!" << std::endl;
      else
 APIERROR(myTransaction->getNdbError());

    if (i != 3) {
      printf(" %2d    %2d\n", i, myRecAttr->u_32_value());
    }
    myNdb.closeTransaction(myTransaction);
  }
}
```

2.5.2 NDB API Example Using Synchronous Transactions and Multiple Clusters

This example demonstrates synchronous transactions and connecting to multiple clusters in a single NDB API application.

The source code for this program may be found in the NDB Cluster source tree, in the file `storage/ndb/ndbapi-examples/ndbapi_simple_dual/main.cpp`.

Note

The example file was formerly named `ndbapi_simple_dual.cpp`.

```
/*
 *  ndbapi_simple_dual: Using synchronous transactions in NDB API
 *
 *  Correct output from this program is:
 *
 *  ATTR1 ATTR2
 *    0    10
 *    1     1
 *    2    12
 *  Detected that deleted tuple doesn't exist!
 *    4    14
 *    5     5
 *    6    16
 *    7     7
 *    8    18
 *    9     9
 *  ATTR1 ATTR2
 *    0    10
 *    1     1
```

```
 *    2    12
 *  Detected that deleted tuple doesn't exist!
 *    4    14
 *    5     5
 *    6    16
 *    7     7
 *    8    18
 *    9     9
 *
 */

#include <mysql.h>
#include <NdbApi.hpp>
// Used for cout
#include <stdio.h>
#include <iostream>

static void run_application(MYSQL &, Ndb_cluster_connection &, const char* table, const char* db);

#define PRINT_ERROR(code,msg) \
  std::cout << "Error in " << __FILE__ << ", line: " << __LINE__ \
            << ", code: " << code \
            << ", msg: " << msg << "." << std::endl
#define MYSQLERROR(mysql) { \
  PRINT_ERROR(mysql_errno(&mysql),mysql_error(&mysql)); \
  exit(-1); }
#define APIERROR(error) { \
  PRINT_ERROR(error.code,error.message); \
  exit(-1); }

int main(int argc, char** argv)
{
  if (argc != 5)
  {
    std::cout << "Arguments are <socket mysqld1> ",
              << "<connect_string cluster 1> ",
              << "<socket mysqld2> <connect_string cluster 2>.\n";
    exit(-1);
  }
  // ndb_init must be called first
  ndb_init();
  {
    char * mysqld1_sock  = argv[1];
    const char *connection_string1 = argv[2];
    char * mysqld2_sock = argv[3];
    const char *connection_string2 = argv[4];

    // Object representing the cluster 1
    Ndb_cluster_connection cluster1_connection(connection_string1);
    MYSQL mysql1;
    // Object representing the cluster 2
    Ndb_cluster_connection cluster2_connection(connection_string2);
    MYSQL mysql2;

    // connect to mysql server and cluster 1 and run application
    // Connect to cluster 1  management server (ndb_mgmd)
    if (cluster1_connection.connect(4 /* retries               */,
        5 /* delay between retries */,
        1 /* verbose               */))
    {
      std::cout << "Cluster 1 management server was not ready within 30 secs.\n";
      exit(-1);
    }
    // Optionally connect and wait for the storage nodes (ndbd's)
    if (cluster1_connection.wait_until_ready(30,0) < 0)
    {
      std::cout << "Cluster 1 was not ready within 30 secs.\n";
      exit(-1);
    }
    // connect to mysql server in cluster 1
    if ( !mysql_init(&mysql1) ) {
      std::cout << "mysql_init failed\n";
```

```
      exit(-1);
    }
    if ( !mysql_real_connect(&mysql1, "localhost", "root", "", "",
           0, mysqld1_sock, 0) )
      MYSQLERROR(mysql1);


    // connect to mysql server and cluster 2 and run application

    // Connect to cluster management server (ndb_mgmd)
    if (cluster2_connection.connect(4 /* retries               */,
        5 /* delay between retries */,
        1 /* verbose              */))
    {
      std::cout << "Cluster 2 management server was not ready within 30 secs.\n";
      exit(-1);
    }
    // Optionally connect and wait for the storage nodes (ndbd's)
    if (cluster2_connection.wait_until_ready(30,0) < 0)
    {
      std::cout << "Cluster 2 was not ready within 30 secs.\n";
      exit(-1);
    }
    // connect to mysql server in cluster 2
    if ( !mysql_init(&mysql2) ) {
      std::cout << "mysql_init failed\n";
      exit(-1);
    }
    if ( !mysql_real_connect(&mysql2, "localhost", "root", "", "",
           0, mysqld2_sock, 0) )
      MYSQLERROR(mysql2);

    // run the application code
    run_application(mysql1, cluster1_connection, "api_simple_dual_1", "ndb_examples");
    run_application(mysql2, cluster2_connection, "api_simple_dual_2", "ndb_examples");
  }
  // Note: all connections must have been destroyed before calling ndb_end()
  ndb_end(0);

  return 0;
}

static void create_table(MYSQL &, const char* table);
static void do_insert(Ndb &, const char* table);
static void do_update(Ndb &, const char* table);
static void do_delete(Ndb &, const char* table);
static void do_read(Ndb &, const char* table);

static void run_application(MYSQL &mysql,
       Ndb_cluster_connection &cluster_connection,
       const char* table,
       const char* db)
{
  /****************************************
   * Connect to database via mysql-c          *
   ****************************************/
 char db_stmt[256];
 sprintf(db_stmt, "CREATE DATABASE %s\n", db);
  mysql_query(&mysql, db_stmt);
 sprintf(db_stmt, "USE %s", db);
  if (mysql_query(&mysql, db_stmt) != 0) MYSQLERROR(mysql);
  create_table(mysql, table);

  /****************************************
   * Connect to database via NDB API           *
   ****************************************/
  // Object representing the database
  Ndb myNdb( &cluster_connection, db );
  if (myNdb.init()) APIERROR(myNdb.getNdbError());

  /*
   * Do different operations on database
```

```
   */
  do_insert(myNdb, table);
  do_update(myNdb, table);
  do_delete(myNdb, table);
  do_read(myNdb, table);
  /*
   * Drop the table
   */
  mysql_query(&mysql, db_stmt);
}

/********************************************************
 * Create a table named by table if it does not exist *
 ********************************************************/
static void create_table(MYSQL &mysql, const char* table)
{
  char create_stmt[256];

  sprintf(create_stmt, "CREATE TABLE %s \
          (ATTR1 INT UNSIGNED NOT NULL PRIMARY KEY,\
           ATTR2 INT UNSIGNED NOT NULL)\
          ENGINE=NDB", table);
  if (mysql_query(&mysql, create_stmt))
    MYSQLERROR(mysql);
}


/*************************************************************************
 * Using 5 transactions, insert 10 tuples in table: (0,0),(1,1),...,(9,9) *
 *************************************************************************/
static void do_insert(Ndb &myNdb, const char* table)
{
  const NdbDictionary::Dictionary* myDict= myNdb.getDictionary();
  const NdbDictionary::Table *myTable= myDict->getTable(table);

  if (myTable == NULL)
    APIERROR(myDict->getNdbError());

  for (int i = 0; i < 5; i++) {
    NdbTransaction *myTransaction= myNdb.startTransaction();
    if (myTransaction == NULL) APIERROR(myNdb.getNdbError());

    NdbOperation *myOperation= myTransaction->getNdbOperation(myTable);
    if (myOperation == NULL) APIERROR(myTransaction->getNdbError());

    myOperation->insertTuple();
    myOperation->equal("ATTR1", i);
    myOperation->setValue("ATTR2", i);

    myOperation= myTransaction->getNdbOperation(myTable);
    if (myOperation == NULL) APIERROR(myTransaction->getNdbError());

    myOperation->insertTuple();
    myOperation->equal("ATTR1", i+5);
    myOperation->setValue("ATTR2", i+5);

    if (myTransaction->execute( NdbTransaction::Commit ) == -1)
      APIERROR(myTransaction->getNdbError());

    myNdb.closeTransaction(myTransaction);
  }
}

/*****************************************************************
 * Update the second attribute in half of the tuples (adding 10) *
 *****************************************************************/
static void do_update(Ndb &myNdb, const char* table)
{
  const NdbDictionary::Dictionary* myDict= myNdb.getDictionary();
  const NdbDictionary::Table *myTable= myDict->getTable(table);

  if (myTable == NULL)
```

```
    APIERROR(myDict->getNdbError());

  for (int i = 0; i < 10; i+=2) {
    NdbTransaction *myTransaction= myNdb.startTransaction();
    if (myTransaction == NULL) APIERROR(myNdb.getNdbError());

    NdbOperation *myOperation= myTransaction->getNdbOperation(myTable);
    if (myOperation == NULL) APIERROR(myTransaction->getNdbError());

    myOperation->updateTuple();
    myOperation->equal( "ATTR1", i );
    myOperation->setValue( "ATTR2", i+10);

    if( myTransaction->execute( NdbTransaction::Commit ) == -1 )
      APIERROR(myTransaction->getNdbError());

    myNdb.closeTransaction(myTransaction);
  }
}

/*********************************************
 * Delete one tuple (the one with primary key 3) *
 *********************************************/
static void do_delete(Ndb &myNdb, const char* table)
{
  const NdbDictionary::Dictionary* myDict= myNdb.getDictionary();
  const NdbDictionary::Table *myTable= myDict->getTable(table);

  if (myTable == NULL)
    APIERROR(myDict->getNdbError());

  NdbTransaction *myTransaction= myNdb.startTransaction();
  if (myTransaction == NULL) APIERROR(myNdb.getNdbError());

  NdbOperation *myOperation= myTransaction->getNdbOperation(myTable);
  if (myOperation == NULL) APIERROR(myTransaction->getNdbError());

  myOperation->deleteTuple();
  myOperation->equal( "ATTR1", 3 );

  if (myTransaction->execute(NdbTransaction::Commit) == -1)
    APIERROR(myTransaction->getNdbError());

  myNdb.closeTransaction(myTransaction);
}

/*****************************
 * Read and print all tuples *
 *****************************/
static void do_read(Ndb &myNdb, const char* table)
{
  const NdbDictionary::Dictionary* myDict= myNdb.getDictionary();
  const NdbDictionary::Table *myTable= myDict->getTable(table);

  if (myTable == NULL)
    APIERROR(myDict->getNdbError());

  std::cout << "ATTR1 ATTR2" << std::endl;

  for (int i = 0; i < 10; i++) {
    NdbTransaction *myTransaction= myNdb.startTransaction();
    if (myTransaction == NULL) APIERROR(myNdb.getNdbError());

    NdbOperation *myOperation= myTransaction->getNdbOperation(myTable);
    if (myOperation == NULL) APIERROR(myTransaction->getNdbError());

    myOperation->readTuple(NdbOperation::LM_Read);
    myOperation->equal("ATTR1", i);

    NdbRecAttr *myRecAttr= myOperation->getValue("ATTR2", NULL);
    if (myRecAttr == NULL) APIERROR(myTransaction->getNdbError());
```

```
    if(myTransaction->execute( NdbTransaction::Commit ) == -1)
      if (i == 3) {
 std::cout << "Detected that deleted tuple doesn't exist!" << std::endl;
      } else {
 APIERROR(myTransaction->getNdbError());
      }

    if (i != 3) {
      printf(" %2d    %2d\n", i, myRecAttr->u_32_value());
    }
    myNdb.closeTransaction(myTransaction);
  }
}
```

2.5.3 NDB API Example: Handling Errors and Retrying Transactions

This program demonstrates handling errors and retrying failed transactions using the NDB API.

The source code for this example can be found in `storage/ndb/ndbapi-examples/ndbapi_retries/ndbapi_retries.cpp` in the NDB Cluster source tree.

There are many ways to program using the NDB API. In this example, we perform two inserts in the same transaction using `NdbTransaction::execute(NoCommit)`.

In NDB API applications, there are two types of failures to be taken into account:

1. **Transaction failures**: If nonpermanent, these can be handled by re-executing the transaction.
2. **Application errors**: These are indicated by `APIERROR`; they must be handled by the application programmer.

```
//
//  ndbapi_retries.cpp: Error handling and transaction retries
//
//  There are many ways to program using the NDB API.  In this example
//  we execute two inserts in the same transaction using
//  NdbConnection::execute(NoCommit).
//
//  Transaction failing is handled by re-executing the transaction
//  in case of non-permanent transaction errors.
//  Application errors (i.e. errors at points marked with APIERROR)
//  should be handled by the application programmer.

#include <mysql.h>
#include <mysqld_error.h>
#include <NdbApi.hpp>

// Used for cout
#include <iostream>

// Used for sleep (use your own version of sleep)
#include <unistd.h>
#define TIME_TO_SLEEP_BETWEEN_TRANSACTION_RETRIES 1

#define PRINT_ERROR(code,msg) \
  std::cout << "Error in " << __FILE__ << ", line: " << __LINE__ \
            << ", code: " << code \
            << ", msg: " << msg << "." << std::endl
#define MYSQLERROR(mysql) { \
  PRINT_ERROR(mysql_errno(&mysql),mysql_error(&mysql)); \
  exit(-1); }

//
//  APIERROR prints an NdbError object
//
#define APIERROR(error) \
  { std::cout << "API ERROR: " << error.code << " " << error.message \
              << std::endl \
              << "           " << "Status: " << error.status \
```

```
              << ", Classification: " << error.classification << std::endl\
              << "           " << "File: " << __FILE__ \
              << " (Line: " << __LINE__ << ")" << std::endl \
              ; \
  }

//
//  TRANSERROR prints all error info regarding an NdbTransaction
//
#define TRANSERROR(ndbTransaction) \
  { NdbError error = ndbTransaction->getNdbError(); \
    std::cout << "TRANS ERROR: " << error.code << " " << error.message \
              << std::endl \
              << "           " << "Status: " << error.status \
              << ", Classification: " << error.classification << std::endl \
              << "           " << "File: " << __FILE__ \
              << " (Line: " << __LINE__ << ")" << std::endl \
              ; \
    printTransactionError(ndbTransaction); \
  }

void printTransactionError(NdbTransaction *ndbTransaction) {
  const NdbOperation *ndbOp = NULL;
  int i=0;

  /**************************************************************
   * Print NdbError object of every operations in the transaction *
   **************************************************************/
  while ((ndbOp = ndbTransaction->getNextCompletedOperation(ndbOp)) != NULL) {
    NdbError error = ndbOp->getNdbError();
    std::cout << "           OPERATION " << i+1 << ": "
       << error.code << " " << error.message << std::endl
       << "           Status: " << error.status
       << ", Classification: " << error.classification << std::endl;
    i++;
  }
}


//
//  Example insert
//  @param myNdb          Ndb object representing NDB Cluster
//  @param myTransaction  NdbTransaction used for transaction
//  @param myTable        Table to insert into
//  @param error          NdbError object returned in case of errors
//  @return -1 in case of failures, 0 otherwise
//
int insert(int transactionId, NdbTransaction* myTransaction,
    const NdbDictionary::Table *myTable) {
  NdbOperation  *myOperation;          // For other operations

  myOperation = myTransaction->getNdbOperation(myTable);
  if (myOperation == NULL) return -1;

  if (myOperation->insertTuple() ||
      myOperation->equal("ATTR1", transactionId) ||
      myOperation->setValue("ATTR2", transactionId)) {
    APIERROR(myOperation->getNdbError());
    exit(-1);
  }

  return myTransaction->execute(NdbTransaction::NoCommit);
}


//
//  Execute function which re-executes (tries 10 times) the transaction
//  if there are temporary errors (e.g. the NDB Cluster is overloaded).
//  @return -1 failure, 1 success
//
int executeInsertTransaction(int transactionId, Ndb* myNdb,
        const NdbDictionary::Table *myTable) {
```

```
  int result = 0;                       // No result yet
  int noOfRetriesLeft = 10;
  NdbTransaction  *myTransaction;         // For other transactions
  NdbError ndberror;

  while (noOfRetriesLeft > 0 && !result) {

    /********************************
     * Start and execute transaction *
     ********************************/
    myTransaction = myNdb->startTransaction();
    if (myTransaction == NULL) {
      APIERROR(myNdb->getNdbError());
      ndberror = myNdb->getNdbError();
      result = -1;  // Failure
    } else if (insert(transactionId, myTransaction, myTable) ||
        insert(10000+transactionId, myTransaction, myTable) ||
        myTransaction->execute(NdbTransaction::Commit)) {
      TRANSERROR(myTransaction);
      ndberror = myTransaction->getNdbError();
      result = -1;  // Failure
    } else {
      result = 1;   // Success
    }

    /********************************
     * If failure, then analyze error *
     ********************************/
    if (result == -1) {
      switch (ndberror.status) {
      case NdbError::Success:
 break;
      case NdbError::TemporaryError:
 std::cout << "Retrying transaction..." << std::endl;
 sleep(TIME_TO_SLEEP_BETWEEN_TRANSACTION_RETRIES);
 --noOfRetriesLeft;
 result = 0;   // No completed transaction yet
 break;

      case NdbError::UnknownResult:
      case NdbError::PermanentError:
 std::cout << "No retry of transaction..." << std::endl;
 result = -1;  // Permanent failure
 break;
      }
    }

    /********************
     * Close transaction *
     ********************/
    if (myTransaction != NULL) {
      myNdb->closeTransaction(myTransaction);
    }
  }

  if (result != 1) exit(-1);
  return result;
}

/*********************************************************
 * Create a table named api_retries if it does not exist *
 *********************************************************/
static void create_table(MYSQL &mysql)
{
  while(mysql_query(&mysql,
    "CREATE TABLE "
    "  api_retries"
    "    (ATTR1 INT UNSIGNED NOT NULL PRIMARY KEY,"
    "     ATTR2 INT UNSIGNED NOT NULL)"
    "  ENGINE=NDB"))
  {
    if (mysql_errno(&mysql) == ER_TABLE_EXISTS_ERROR)
```

```
    {
      std::cout << "NDB Cluster already has example table: api_scan. "
      << "Dropping it..." << std::endl;
        mysql_query(&mysql, "DROP TABLE api_retries");
    }
    else MYSQLERROR(mysql);
  }
}


int main(int argc, char** argv)
{
  if (argc != 3)
  {
    std::cout << "Arguments are <socket mysqld> <connect_string cluster>.\n";
    exit(-1);
  }
  char * mysqld_sock  = argv[1];
  const char *connection_string = argv[2];
  ndb_init();

  Ndb_cluster_connection *cluster_connection=
    new Ndb_cluster_connection(connection_string); // Object representing the cluster

  int r= cluster_connection->connect(5 /* retries               */,
        3 /* delay between retries */,
        1 /* verbose               */);
  if (r > 0)
  {
    std::cout
      << "Cluster connect failed, possibly resolved with more retries.\n";
    exit(-1);
  }
  else if (r < 0)
  {
    std::cout
      << "Cluster connect failed.\n";
    exit(-1);
  }

  if (cluster_connection->wait_until_ready(30,30))
  {
    std::cout << "Cluster was not ready within 30 secs." << std::endl;
    exit(-1);
  }
  // connect to mysql server
  MYSQL mysql;
  if ( !mysql_init(&mysql) ) {
    std::cout << "mysql_init failed\n";
    exit(-1);
  }
  if ( !mysql_real_connect(&mysql, "localhost", "root", "", "",
      0, mysqld_sock, 0) )
    MYSQLERROR(mysql);

  /*****************************************
   * Connect to database via mysql-c          *
   *****************************************/
  mysql_query(&mysql, "CREATE DATABASE ndb_examples");
  if (mysql_query(&mysql, "USE ndb_examples") != 0) MYSQLERROR(mysql);
  create_table(mysql);

  Ndb* myNdb= new Ndb( cluster_connection,
        "ndb_examples" );  // Object representing the database

  if (myNdb->init() == -1) {
    APIERROR(myNdb->getNdbError());
    exit(-1);
  }

  const NdbDictionary::Dictionary* myDict= myNdb->getDictionary();
  const NdbDictionary::Table *myTable= myDict->getTable("api_retries");
```

```
  if (myTable == NULL)
  {
    APIERROR(myDict->getNdbError());
    return -1;
  }
  /************************************
   * Execute some insert transactions *
   ************************************/

  std::cout << "Ready to insert rows.  You will see notices for temporary "
    "errors, permenant errors, and retries. \n";
  for (int i = 10000; i < 20000; i++) {
    executeInsertTransaction(i, myNdb, myTable);
  }
  std::cout << "Done.\n";

  delete myNdb;
  delete cluster_connection;

  ndb_end(0);
  return 0;
}
```

2.5.4 NDB API Basic Scanning Example

This example illustrates how to use the NDB scanning API. It shows how to perform a scan, how to scan for an update, and how to scan for a delete, making use of the `NdbScanFilter` and `NdbScanOperation` classes.

The source code for this example may found in the NDB Cluster source tree, in the file `storage/ndb/ndbapi-examples/ndbapi_scan/ndbapi_scan.cpp`.

This example makes use of the following classes and methods:

- `Ndb_cluster_connection`:
  - `connect()`
  - `wait_until_ready()`
- `Ndb`:
  - `init()`
  - `getDictionary()`
  - `startTransaction()`
  - `closeTransaction()`
- `NdbTransaction`:
  - `getNdbScanOperation()`
  - `execute()`
- `NdbOperation`:
  - `insertTuple()`
  - `equal()`
  - `getValue()`
  - `setValue()`
- `NdbScanOperation`:

  - readTuples()
  - nextResult()
  - deleteCurrentTuple()
  - updateCurrentTuple()
- NdbDictionary:
  - Dictionary::getTable()
  - Table::getColumn()
  - Column::getLength()
- NdbScanFilter:
  - begin()
  - eq()
  - end()

```
/*
   Copyright (c) 2005, 2016, Oracle and/or its affiliates. All rights reserved.

   This program is free software; you can redistribute it and/or modify
   it under the terms of the GNU General Public License as published by
   the Free Software Foundation; version 2 of the License.

   This program is distributed in the hope that it will be useful,
   but WITHOUT ANY WARRANTY; without even the implied warranty of
   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
   GNU General Public License for more details.

   You should have received a copy of the GNU General Public License
   along with this program; if not, write to the Free Software
   Foundation, Inc., 51 Franklin St, Fifth Floor, Boston, MA 02110-1301  USA
*/


/*
 * ndbapi_scan.cpp:
 * Illustrates how to use the scan api in the NDBAPI.
 * The example shows how to do scan, scan for update and scan for delete
 * using NdbScanFilter and NdbScanOperation
 *
 * Classes and methods used in this example:
 *
 *  Ndb_cluster_connection
 *       connect()
 *       wait_until_ready()
 *
 *  Ndb
 *       init()
 *       getDictionary()
 *       startTransaction()
 *       closeTransaction()
 *
 *  NdbTransaction
 *       getNdbScanOperation()
 *       execute()
 *
 *  NdbScanOperation
 *       getValue()
 *       readTuples()
```

```
 *       nextResult()
 *       deleteCurrentTuple()
 *       updateCurrentTuple()
 *
 *  const NdbDictionary::Dictionary
 *       getTable()
 *
 *  const NdbDictionary::Table
 *       getColumn()
 *
 *  const NdbDictionary::Column
 *       getLength()
 *
 *  NdbOperation
 *       insertTuple()
 *       equal()
 *       setValue()
 *
 *  NdbScanFilter
 *       begin()
 *  eq()
 *  end()
 *
 */

#ifdef _WIN32
#include <winsock2.h>
#endif
#include <mysql.h>
#include <mysqld_error.h>
#include <NdbApi.hpp>
// Used for cout
#include <iostream>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <config.h>
#ifdef HAVE_SYS_SELECT_H
#include <sys/select.h>
#endif

/**
 * Helper sleep function
 */
static void
milliSleep(int milliseconds){
  struct timeval sleeptime;
  sleeptime.tv_sec = milliseconds / 1000;
  sleeptime.tv_usec = (milliseconds - (sleeptime.tv_sec * 1000)) * 1000000;
  select(0, 0, 0, 0, &sleeptime);
}


/**
 * Helper debugging macros
 */
#define PRINT_ERROR(code,msg) \
  std::cout << "Error in " << __FILE__ << ", line: " << __LINE__ \
            << ", code: " << code \
            << ", msg: " << msg << "." << std::endl
#define MYSQLERROR(mysql) { \
  PRINT_ERROR(mysql_errno(&mysql),mysql_error(&mysql)); \
  exit(-1); }
#define APIERROR(error) { \
  PRINT_ERROR(error.code,error.message); \
  exit(-1); }

struct Car
{
  /**
   * Note memset, so that entire char-fields are cleared
   *   as all 20 bytes are significant (as type is char)
```

```
   */
  Car() { memset(this, 0, sizeof(* this)); }

  unsigned int reg_no;
  char brand[20];
  char color[20];
};

/**
 * Function to drop table
 */
void drop_table(MYSQL &mysql)
{
  if (mysql_query(&mysql, "DROP TABLE IF EXISTS api_scan"))
    MYSQLERROR(mysql);
}


/**
 * Function to create table
 */
void create_table(MYSQL &mysql)
{
  while (mysql_query(&mysql,
    "CREATE TABLE"
    "  api_scan"
    "    (REG_NO INT UNSIGNED NOT NULL,"
    "     BRAND CHAR(20) NOT NULL,"
    "     COLOR CHAR(20) NOT NULL,"
    "     PRIMARY KEY USING HASH (REG_NO))"
    "  ENGINE=NDB"))
  {
    if (mysql_errno(&mysql) != ER_TABLE_EXISTS_ERROR)
      MYSQLERROR(mysql);
    std::cout << "NDB Cluster already has example table: api_scan. "
       << "Dropping it..." << std::endl;
    drop_table(mysql);
  }
}

int populate(Ndb * myNdb)
{
  int i;
  Car cars[15];

  const NdbDictionary::Dictionary* myDict= myNdb->getDictionary();
  const NdbDictionary::Table *myTable= myDict->getTable("api_scan");

  if (myTable == NULL)
    APIERROR(myDict->getNdbError());

  /**
   * Five blue mercedes
   */
  for (i = 0; i < 5; i++)
  {
    cars[i].reg_no = i;
    sprintf(cars[i].brand, "Mercedes");
    sprintf(cars[i].color, "Blue");
  }

  /**
   * Five black bmw
   */
  for (i = 5; i < 10; i++)
  {
    cars[i].reg_no = i;
    sprintf(cars[i].brand, "BMW");
    sprintf(cars[i].color, "Black");
  }

  /**
```

```
   * Five pink toyotas
   */
  for (i = 10; i < 15; i++)
  {
    cars[i].reg_no = i;
    sprintf(cars[i].brand, "Toyota");
    sprintf(cars[i].color, "Pink");
  }

  NdbTransaction* myTrans = myNdb->startTransaction();
  if (myTrans == NULL)
    APIERROR(myNdb->getNdbError());

  for (i = 0; i < 15; i++)
  {
    NdbOperation* myNdbOperation = myTrans->getNdbOperation(myTable);
    if (myNdbOperation == NULL)
      APIERROR(myTrans->getNdbError());
    myNdbOperation->insertTuple();
    myNdbOperation->equal("REG_NO", cars[i].reg_no);
    myNdbOperation->setValue("BRAND", cars[i].brand);
    myNdbOperation->setValue("COLOR", cars[i].color);
  }

  int check = myTrans->execute(NdbTransaction::Commit);

  myTrans->close();

  return check != -1;
}

int scan_delete(Ndb* myNdb,
  int column,
  const char * color)

{

  // Scan all records exclusive and delete
  // them one by one
  int                  retryAttempt = 0;
  const int            retryMax = 10;
  int deletedRows = 0;
  int check;
  NdbError              err;
  NdbTransaction *myTrans;
  NdbScanOperation *myScanOp;

  const NdbDictionary::Dictionary* myDict= myNdb->getDictionary();
  const NdbDictionary::Table *myTable= myDict->getTable("api_scan");

  if (myTable == NULL)
    APIERROR(myDict->getNdbError());

  /**
   * Loop as long as :
   *  retryMax not reached
   *  failed operations due to TEMPORARY erros
   *
   * Exit loop;
   *  retyrMax reached
   *  Permanent error (return -1)
   */
  while (true)
  {
    if (retryAttempt >= retryMax)
    {
      std::cout << "ERROR: has retried this operation " << retryAttempt
  << " times, failing!" << std::endl;
      return -1;
    }

    myTrans = myNdb->startTransaction();
```

```
   if (myTrans == NULL)
   {
     const NdbError err = myNdb->getNdbError();

     if (err.status == NdbError::TemporaryError)
     {
milliSleep(50);
retryAttempt++;
continue;
     }
     std::cout <<  err.message << std::endl;
     return -1;
   }

  /**
   * Get a scan operation.
   */
   myScanOp = myTrans->getNdbScanOperation(myTable);
   if (myScanOp == NULL)
   {
     std::cout << myTrans->getNdbError().message << std::endl;
     myNdb->closeTransaction(myTrans);
     return -1;
   }

   /**
    * Define a result set for the scan.
    */
   if(myScanOp->readTuples(NdbOperation::LM_Exclusive) != 0)
   {
     std::cout << myTrans->getNdbError().message << std::endl;
     myNdb->closeTransaction(myTrans);
     return -1;
   }

   /**
    * Use NdbScanFilter to define a search critera
    */
   NdbScanFilter filter(myScanOp) ;
   if(filter.begin(NdbScanFilter::AND) < 0  ||
      filter.cmp(NdbScanFilter::COND_EQ, column, color, 20) < 0 ||
      filter.end() < 0)
   {
     std::cout <<  myTrans->getNdbError().message << std::endl;
     myNdb->closeTransaction(myTrans);
     return -1;
   }

   /**
    * Start scan    (NoCommit since we are only reading at this stage);
    */
   if(myTrans->execute(NdbTransaction::NoCommit) != 0){
     err = myTrans->getNdbError();
     if(err.status == NdbError::TemporaryError){
std::cout << myTrans->getNdbError().message << std::endl;
myNdb->closeTransaction(myTrans);
milliSleep(50);
continue;
     }
     std::cout << err.code << std::endl;
     std::cout << myTrans->getNdbError().code << std::endl;
     myNdb->closeTransaction(myTrans);
     return -1;
   }


  /**
   * start of loop: nextResult(true) means that "parallelism" number of
   * rows are fetched from NDB and cached in NDBAPI
   */
   while((check = myScanOp->nextResult(true)) == 0){
     do
```

```
     {
 if (myScanOp->deleteCurrentTuple() != 0)
 {
   std::cout << myTrans->getNdbError().message << std::endl;
   myNdb->closeTransaction(myTrans);
   return -1;
 }
 deletedRows++;

 /**
  * nextResult(false) means that the records
  * cached in the NDBAPI are modified before
  * fetching more rows from NDB.
  */
     } while((check = myScanOp->nextResult(false)) == 0);

     /**
      * NoCommit when all cached tuple have been marked for deletion
      */
     if(check != -1)
     {
 check = myTrans->execute(NdbTransaction::NoCommit);
     }

     /**
      * Check for errors
      */
     err = myTrans->getNdbError();
     if(check == -1)
     {
 if(err.status == NdbError::TemporaryError)
 {
   std::cout << myTrans->getNdbError().message << std::endl;
   myNdb->closeTransaction(myTrans);
   milliSleep(50);
   continue;
 }
     }
     /**
      * End of loop
      */
   }
   /**
    * Commit all prepared operations
    */
   if(myTrans->execute(NdbTransaction::Commit) == -1)
   {
     if(err.status == NdbError::TemporaryError){
 std::cout << myTrans->getNdbError().message << std::endl;
 myNdb->closeTransaction(myTrans);
 milliSleep(50);
 continue;
     }
   }

   std::cout << myTrans->getNdbError().message << std::endl;
   myNdb->closeTransaction(myTrans);
   return 0;
 }

 if(myTrans!=0)
 {
   std::cout << myTrans->getNdbError().message << std::endl;
   myNdb->closeTransaction(myTrans);
 }
 return -1;
}


int scan_update(Ndb* myNdb,
  int update_column,
  const char * before_color,
```

```
  const char * after_color)

{

  // Scan all records exclusive and update
  // them one by one
  int                  retryAttempt = 0;
  const int            retryMax = 10;
  int updatedRows = 0;
  int check;
  NdbError              err;
  NdbTransaction *myTrans;
  NdbScanOperation *myScanOp;

  const NdbDictionary::Dictionary* myDict= myNdb->getDictionary();
  const NdbDictionary::Table *myTable= myDict->getTable("api_scan");

  if (myTable == NULL)
    APIERROR(myDict->getNdbError());

  /**
   * Loop as long as :
   *  retryMax not reached
   *  failed operations due to TEMPORARY erros
   *
   * Exit loop;
   *  retryMax reached
   *  Permanent error (return -1)
   */
  while (true)
  {

    if (retryAttempt >= retryMax)
    {
      std::cout << "ERROR: has retried this operation " << retryAttempt
 << " times, failing!" << std::endl;
      return -1;
    }

    myTrans = myNdb->startTransaction();
    if (myTrans == NULL)
    {
      const NdbError err = myNdb->getNdbError();

      if (err.status == NdbError::TemporaryError)
      {
 milliSleep(50);
 retryAttempt++;
 continue;
      }
      std::cout <<  err.message << std::endl;
      return -1;
    }

   /**
    * Get a scan operation.
    */
    myScanOp = myTrans->getNdbScanOperation(myTable);
    if (myScanOp == NULL)
    {
      std::cout << myTrans->getNdbError().message << std::endl;
      myNdb->closeTransaction(myTrans);
      return -1;
    }

    /**
     * Define a result set for the scan.
     */
    if( myScanOp->readTuples(NdbOperation::LM_Exclusive) )
    {
      std::cout << myTrans->getNdbError().message << std::endl;
      myNdb->closeTransaction(myTrans);
```

```
     return -1;
   }

   /**
    * Use NdbScanFilter to define a search critera
    */
   NdbScanFilter filter(myScanOp) ;
   if(filter.begin(NdbScanFilter::AND) < 0  ||
      filter.cmp(NdbScanFilter::COND_EQ, update_column, before_color, 20) <0||
      filter.end() <0)
   {
     std::cout <<  myTrans->getNdbError().message << std::endl;
     myNdb->closeTransaction(myTrans);
     return -1;
   }

   /**
    * Start scan    (NoCommit since we are only reading at this stage);
    */
   if(myTrans->execute(NdbTransaction::NoCommit) != 0)
   {
     err = myTrans->getNdbError();
     if(err.status == NdbError::TemporaryError){
std::cout << myTrans->getNdbError().message << std::endl;
myNdb->closeTransaction(myTrans);
milliSleep(50);
continue;
     }
     std::cout << myTrans->getNdbError().code << std::endl;
     myNdb->closeTransaction(myTrans);
     return -1;
   }

   /**
    * start of loop: nextResult(true) means that "parallelism" number of
    * rows are fetched from NDB and cached in NDBAPI
    */
   while((check = myScanOp->nextResult(true)) == 0){
     do {
/**
 * Get update operation
 */
NdbOperation * myUpdateOp = myScanOp->updateCurrentTuple();
if (myUpdateOp == 0)
{
  std::cout << myTrans->getNdbError().message << std::endl;
  myNdb->closeTransaction(myTrans);
  return -1;
}
updatedRows++;

/**
 * do the update
 */
myUpdateOp->setValue(update_column, after_color);
/**
 * nextResult(false) means that the records
 * cached in the NDBAPI are modified before
 * fetching more rows from NDB.
 */
     } while((check = myScanOp->nextResult(false)) == 0);

     /**
      * NoCommit when all cached tuple have been updated
      */
     if(check != -1)
     {
check = myTrans->execute(NdbTransaction::NoCommit);
     }

     /**
      * Check for errors
```

```
       */
      err = myTrans->getNdbError();
      if(check == -1)
      {
 if(err.status == NdbError::TemporaryError){
   std::cout << myTrans->getNdbError().message << std::endl;
   myNdb->closeTransaction(myTrans);
   milliSleep(50);
   continue;
 }
      }
      /**
       * End of loop
       */
    }

    /**
     * Commit all prepared operations
     */
    if(myTrans->execute(NdbTransaction::Commit) == -1)
    {
      if(err.status == NdbError::TemporaryError){
 std::cout << myTrans->getNdbError().message << std::endl;
 myNdb->closeTransaction(myTrans);
 milliSleep(50);
 continue;
      }
    }

    std::cout << myTrans->getNdbError().message << std::endl;
    myNdb->closeTransaction(myTrans);
    return 0;
  }


  if(myTrans!=0)
  {
    std::cout << myTrans->getNdbError().message << std::endl;
    myNdb->closeTransaction(myTrans);
  }
  return -1;
}



int scan_print(Ndb * myNdb)
{
// Scan all records exclusive and update
  // them one by one
  int                  retryAttempt = 0;
  const int            retryMax = 10;
  int fetchedRows = 0;
  int check;
  NdbError              err;
  NdbTransaction *myTrans;
  NdbScanOperation *myScanOp;
  /* Result of reading attribute value, three columns:
     REG_NO, BRAND, and COLOR
   */
  NdbRecAttr *     myRecAttr[3];

  const NdbDictionary::Dictionary* myDict= myNdb->getDictionary();
  const NdbDictionary::Table *myTable= myDict->getTable("api_scan");

  if (myTable == NULL)
    APIERROR(myDict->getNdbError());

  /**
   * Loop as long as :
   *  retryMax not reached
   *  failed operations due to TEMPORARY erros
   *
```

```
  * Exit loop;
  *  retyrMax reached
  *  Permanent error (return -1)
  */
 while (true)
 {

   if (retryAttempt >= retryMax)
   {
     std::cout << "ERROR: has retried this operation " << retryAttempt
 << " times, failing!" << std::endl;
     return -1;
   }

   myTrans = myNdb->startTransaction();
   if (myTrans == NULL)
   {
     const NdbError err = myNdb->getNdbError();

     if (err.status == NdbError::TemporaryError)
     {
milliSleep(50);
retryAttempt++;
continue;
     }
    std::cout << err.message << std::endl;
     return -1;
   }
   /*
    * Define a scan operation.
    * NDBAPI.
    */
   myScanOp = myTrans->getNdbScanOperation(myTable);
   if (myScanOp == NULL)
   {
     std::cout << myTrans->getNdbError().message << std::endl;
     myNdb->closeTransaction(myTrans);
     return -1;
   }

   /**
    * Read without locks, without being placed in lock queue
    */
   if( myScanOp->readTuples(NdbOperation::LM_CommittedRead) == -1)
   {
     std::cout << myTrans->getNdbError().message << std::endl;
     myNdb->closeTransaction(myTrans);
     return -1;
   }

   /**
    * Define storage for fetched attributes.
    * E.g., the resulting attributes of executing
    * myOp->getValue("REG_NO") is placed in myRecAttr[0].
    * No data exists in myRecAttr until transaction has commited!
    */
   myRecAttr[0] = myScanOp->getValue("REG_NO");
   myRecAttr[1] = myScanOp->getValue("BRAND");
   myRecAttr[2] = myScanOp->getValue("COLOR");
   if(myRecAttr[0] ==NULL || myRecAttr[1] == NULL || myRecAttr[2]==NULL)
   {
std::cout << myTrans->getNdbError().message << std::endl;
myNdb->closeTransaction(myTrans);
return -1;
   }
   /**
    * Start scan   (NoCommit since we are only reading at this stage);
    */
   if(myTrans->execute(NdbTransaction::NoCommit) != 0){
     err = myTrans->getNdbError();
     if(err.status == NdbError::TemporaryError){
std::cout << myTrans->getNdbError().message << std::endl;
```

```
 myNdb->closeTransaction(myTrans);
 milliSleep(50);
 continue;
      }
      std::cout << err.code << std::endl;
      std::cout << myTrans->getNdbError().code << std::endl;
      myNdb->closeTransaction(myTrans);
      return -1;
    }

    /**
     * start of loop: nextResult(true) means that "parallelism" number of
     * rows are fetched from NDB and cached in NDBAPI
     */
    while((check = myScanOp->nextResult(true)) == 0){
      do {

 fetchedRows++;
 /**
  * print  REG_NO unsigned int
  */
 std::cout << myRecAttr[0]->u_32_value() << "\t";

 /**
  * print  BRAND character string
  */
 std::cout << myRecAttr[1]->aRef() << "\t";

 /**
  * print  COLOR character string
  */
 std::cout << myRecAttr[2]->aRef() << std::endl;

 /**
  * nextResult(false) means that the records
  * cached in the NDBAPI are modified before
  * fetching more rows from NDB.
  */
      } while((check = myScanOp->nextResult(false)) == 0);

    }
    myNdb->closeTransaction(myTrans);
    return 1;
  }
  return -1;

}

void mysql_connect_and_create(MYSQL & mysql, const char *socket)
{
  bool ok;

  ok = mysql_real_connect(&mysql, "localhost", "root", "", "", 0, socket, 0);
  if(ok) {
    mysql_query(&mysql, "CREATE DATABASE ndb_examples");
    ok = ! mysql_select_db(&mysql, "ndb_examples");
  }
  if(ok) {
    create_table(mysql);
  }

  if(! ok) MYSQLERROR(mysql);
}

void ndb_run_scan(const char * connectstring)
{

  /*************************************************************
   * Connect to ndb cluster                                    *
   *************************************************************/

  Ndb_cluster_connection cluster_connection(connectstring);
```

```
  if (cluster_connection.connect(4, 5, 1))
  {
    std::cout << "Unable to connect to cluster within 30 secs." << std::endl;
    exit(-1);
  }
  // Optionally connect and wait for the storage nodes (ndbd's)
  if (cluster_connection.wait_until_ready(30,0) < 0)
  {
    std::cout << "Cluster was not ready within 30 secs.\n";
    exit(-1);
  }

  Ndb myNdb(&cluster_connection,"ndb_examples");
  if (myNdb.init(1024) == -1) {      // Set max 1024  parallel transactions
    APIERROR(myNdb.getNdbError());
    exit(-1);
  }

  /****************************************
   * Check table definition                *
   ****************************************/
  int column_color;
  {
    const NdbDictionary::Dictionary* myDict= myNdb.getDictionary();
    const NdbDictionary::Table *t= myDict->getTable("api_scan");
    if(t == NULL)
    {
      std::cout << "Dictionary::getTable() failed.";
      exit(-1);
    }
    Car car;
    if (t->getColumn("COLOR")->getLength() != sizeof(car.color) ||
t->getColumn("BRAND")->getLength() != sizeof(car.brand))
    {
      std::cout << "Wrong table definition" << std::endl;
      exit(-1);
    }
    column_color= t->getColumn("COLOR")->getColumnNo();
  }

  if(populate(&myNdb) > 0)
    std::cout << "populate: Success!" << std::endl;

  if(scan_print(&myNdb) > 0)
    std::cout << "scan_print: Success!" << std::endl  << std::endl;

  std::cout << "Going to delete all pink cars!" << std::endl;

  {
    /**
     * Note! color needs to be of exact the same size as column defined
     */
    Car tmp;
    sprintf(tmp.color, "Pink");
    if(scan_delete(&myNdb, column_color, tmp.color) > 0)
      std::cout << "scan_delete: Success!" << std::endl  << std::endl;
  }

  if(scan_print(&myNdb) > 0)
    std::cout << "scan_print: Success!" << std::endl  << std::endl;

  {
    /**
     * Note! color1 & 2 need to be of exact the same size as column defined
     */
    Car tmp1, tmp2;
    sprintf(tmp1.color, "Blue");
    sprintf(tmp2.color, "Black");
    std::cout << "Going to update all " << tmp1.color
       << " cars to " << tmp2.color << " cars!" << std::endl;
    if(scan_update(&myNdb, column_color, tmp1.color, tmp2.color) > 0)
      std::cout << "scan_update: Success!" << std::endl  << std::endl;
```

```
  }
  if(scan_print(&myNdb) > 0)
    std::cout << "scan_print: Success!" << std::endl  << std::endl;
}

int main(int argc, char** argv)
{
  if (argc != 3)
  {
    std::cout << "Arguments are <socket mysqld> <connect_string cluster>.\n";
    exit(-1);
  }
  char * mysqld_sock  = argv[1];
  const char *connectstring = argv[2];
  MYSQL mysql;

  mysql_init(& mysql);
  mysql_connect_and_create(mysql, mysqld_sock);

  ndb_init();
  ndb_run_scan(connectstring);
  ndb_end(0);

  mysql_close(&mysql);

  return 0;
}
```

2.5.5 NDB API Example: Using Secondary Indexes in Scans

This program illustrates how to use secondary indexes in the NDB API.

The source code for this example may be found in the NDB Cluster source tree, in `storage/ndb/ndbapi-examples/ndbapi_simple_index/main.cpp`.

Note

This file was previously named `ndbapi_simple_index.cpp`.

The correct output from this program is shown here:

```
ATTR1 ATTR2
0      10
1       1
2      12
Detected that deleted tuple doesn't exist!
4      14
5       5
6      16
7       7
8      18
9       9
```

The listing for this program is shown here:

```
#include <mysql.h>
#include <mysqld_error.h>
#include <NdbApi.hpp>

// Used for cout
#include <stdio.h>
#include <iostream>

#define PRINT_ERROR(code,msg) \
  std::cout << "Error in " << __FILE__ << ", line: " << __LINE__ \
            << ", code: " << code \
            << ", msg: " << msg << "." << std::endl
#define MYSQLERROR(mysql) { \
```

```
  PRINT_ERROR(mysql_errno(&mysql),mysql_error(&mysql)); \
  exit(-1); }
#define APIERROR(error) { \
  PRINT_ERROR(error.code,error.message); \
  exit(-1); }

int main(int argc, char** argv)
{
  if (argc != 3)
    {
    std::cout << "Arguments are <socket mysqld> <connect_string cluster>.\n";
    exit(-1);
  }
  char * mysqld_sock  = argv[1];
  const char *connection_string = argv[2];
  ndb_init();
  MYSQL mysql;

  /****************************************************************
   * Connect to mysql server and create table                     *
   ****************************************************************/
  {
    if ( !mysql_init(&mysql) ) {
      std::cout << "mysql_init failed\n";
      exit(-1);
    }
    if ( !mysql_real_connect(&mysql, "localhost", "root", "", "",
        0, mysqld_sock, 0) )
      MYSQLERROR(mysql);

    mysql_query(&mysql, "CREATE DATABASE ndb_examples_1");
    if (mysql_query(&mysql, "USE ndb_examples") != 0) MYSQLERROR(mysql);

    while (mysql_query(&mysql,
      "CREATE TABLE"
      "  api_simple_index"
      "    (ATTR1 INT UNSIGNED,"
      "     ATTR2 INT UNSIGNED NOT NULL,"
      "     PRIMARY KEY USING HASH (ATTR1),"
      "     UNIQUE MYINDEXNAME USING HASH (ATTR2))"
      "  ENGINE=NDB"))
    {
      if (mysql_errno(&mysql) == ER_TABLE_EXISTS_ERROR)
      {
        std::cout << "NDB Cluster already has example table: api_scan. "
        << "Dropping it..." << std::endl;
        mysql_query(&mysql, "DROP TABLE api_simple_index");
      }
      else MYSQLERROR(mysql);
    }
  }

  /****************************************************************
   * Connect to ndb cluster                                       *
   ****************************************************************/

  Ndb_cluster_connection *cluster_connection=
    new Ndb_cluster_connection(connection_string); // Object representing the cluster

  if (cluster_connection->connect(5,3,1))
  {
    std::cout << "Connect to cluster management server failed.\n";
    exit(-1);
  }

  if (cluster_connection->wait_until_ready(30,30))
  {
    std::cout << "Cluster was not ready within 30 secs.\n";
    exit(-1);
  }

  Ndb* myNdb = new Ndb( cluster_connection,
```

```
 "ndb_examples" );  // Object representing the database
if (myNdb->init() == -1) {
  APIERROR(myNdb->getNdbError());
  exit(-1);
}

const NdbDictionary::Dictionary* myDict= myNdb->getDictionary();
const NdbDictionary::Table *myTable= myDict->getTable("api_simple_index");
if (myTable == NULL)
  APIERROR(myDict->getNdbError());
const NdbDictionary::Index *myIndex= myDict->getIndex("MYINDEXNAME$unique","api_simple_index");
if (myIndex == NULL)
  APIERROR(myDict->getNdbError());

/*************************************************************************
 * Using 5 transactions, insert 10 tuples in table: (0,0),(1,1),...,(9,9) *
 *************************************************************************/
for (int i = 0; i < 5; i++) {
  NdbTransaction *myTransaction= myNdb->startTransaction();
  if (myTransaction == NULL) APIERROR(myNdb->getNdbError());

  NdbOperation *myOperation= myTransaction->getNdbOperation(myTable);
  if (myOperation == NULL) APIERROR(myTransaction->getNdbError());

  myOperation->insertTuple();
  myOperation->equal("ATTR1", i);
  myOperation->setValue("ATTR2", i);

  myOperation = myTransaction->getNdbOperation(myTable);
  if (myOperation == NULL) APIERROR(myTransaction->getNdbError());

  myOperation->insertTuple();
  myOperation->equal("ATTR1", i+5);
  myOperation->setValue("ATTR2", i+5);

  if (myTransaction->execute( NdbTransaction::Commit ) == -1)
    APIERROR(myTransaction->getNdbError());

  myNdb->closeTransaction(myTransaction);
}

/****************************************
 * Read and print all tuples using index *
 ****************************************/
std::cout << "ATTR1 ATTR2" << std::endl;

for (int i = 0; i < 10; i++) {
  NdbTransaction *myTransaction= myNdb->startTransaction();
  if (myTransaction == NULL) APIERROR(myNdb->getNdbError());

  NdbIndexOperation *myIndexOperation=
    myTransaction->getNdbIndexOperation(myIndex);
  if (myIndexOperation == NULL) APIERROR(myTransaction->getNdbError());

  myIndexOperation->readTuple(NdbOperation::LM_Read);
  myIndexOperation->equal("ATTR2", i);

  NdbRecAttr *myRecAttr= myIndexOperation->getValue("ATTR1", NULL);
  if (myRecAttr == NULL) APIERROR(myTransaction->getNdbError());

  if(myTransaction->execute( NdbTransaction::Commit,
                             NdbOperation::AbortOnError ) != -1)
    printf(" %2d    %2d\n", myRecAttr->u_32_value(), i);

  myNdb->closeTransaction(myTransaction);
}

/************************************************************
 * Update the second attribute in half of the tuples (adding 10) *
 ************************************************************/
for (int i = 0; i < 10; i+=2) {
  NdbTransaction *myTransaction= myNdb->startTransaction();
```

```
  if (myTransaction == NULL) APIERROR(myNdb->getNdbError());

  NdbIndexOperation *myIndexOperation=
    myTransaction->getNdbIndexOperation(myIndex);
  if (myIndexOperation == NULL) APIERROR(myTransaction->getNdbError());

  myIndexOperation->updateTuple();
  myIndexOperation->equal( "ATTR2", i );
  myIndexOperation->setValue( "ATTR2", i+10);

  if( myTransaction->execute( NdbTransaction::Commit ) == -1 )
    APIERROR(myTransaction->getNdbError());

  myNdb->closeTransaction(myTransaction);
}

/********************************************
 * Delete one tuple (the one with primary key 3) *
 ********************************************/
{
  NdbTransaction *myTransaction= myNdb->startTransaction();
  if (myTransaction == NULL) APIERROR(myNdb->getNdbError());

  NdbIndexOperation *myIndexOperation=
    myTransaction->getNdbIndexOperation(myIndex);
  if (myIndexOperation == NULL) APIERROR(myTransaction->getNdbError());

  myIndexOperation->deleteTuple();
  myIndexOperation->equal( "ATTR2", 3 );

  if (myTransaction->execute(NdbTransaction::Commit) == -1)
    APIERROR(myTransaction->getNdbError());

  myNdb->closeTransaction(myTransaction);
}

/*****************************
 * Read and print all tuples *
 *****************************/
{
  std::cout << "ATTR1 ATTR2" << std::endl;

  for (int i = 0; i < 10; i++) {
    NdbTransaction *myTransaction= myNdb->startTransaction();
    if (myTransaction == NULL) APIERROR(myNdb->getNdbError());

    NdbOperation *myOperation= myTransaction->getNdbOperation(myTable);
    if (myOperation == NULL) APIERROR(myTransaction->getNdbError());

    myOperation->readTuple(NdbOperation::LM_Read);
    myOperation->equal("ATTR1", i);

    NdbRecAttr *myRecAttr= myOperation->getValue("ATTR2", NULL);
    if (myRecAttr == NULL) APIERROR(myTransaction->getNdbError());

    if(myTransaction->execute( NdbTransaction::Commit,
                               NdbOperation::AbortOnError ) == -1)
if (i == 3) {
  std::cout << "Detected that deleted tuple doesn't exist!\n";
} else {
  APIERROR(myTransaction->getNdbError());
}

    if (i != 3) {
printf(" %2d    %2d\n", i, myRecAttr->u_32_value());
    }
    myNdb->closeTransaction(myTransaction);
  }
}

delete myNdb;
delete cluster_connection;
```

```
  ndb_end(0);
  return 0;
}
```

2.5.6 NDB API Example: Using NdbRecord with Hash Indexes

This program illustrates how to use secondary indexes in the NDB API with the aid of the `NdbRecord` interface.

The source code for this example may be found in the NDB Cluster source trees, in the file `storage/ndb/ndbapi-examples/ndbapi_s_i_ndbrecord/main.cpp`.

When run on a cluster having 2 data nodes, the correct output from this program is as shown here:

```
ATTR1 ATTR2
  0     0   (frag=0)
  1     1   (frag=1)
  2     2   (frag=1)
  3     3   (frag=0)
  4     4   (frag=1)
  5     5   (frag=1)
  6     6   (frag=0)
  7     7   (frag=0)
  8     8   (frag=1)
  9     9   (frag=0)
ATTR1 ATTR2
  0    10
  1     1
  2    12
Detected that deleted tuple doesn't exist!
  4    14
  5     5
  6    16
  7     7
  8    18
  9     9
```

The program listing is shown here:

```
//
//  ndbapi_simple_index_ndbrecord.cpp: Using secondary unique hash indexes
//  in NDB API, utilising the NdbRecord interface.
//
//  Correct output from this program is (from a two-node cluster):
//
// ATTR1 ATTR2
//   0     0   (frag=0)
//   1     1   (frag=1)
//   2     2   (frag=1)
//   3     3   (frag=0)
//   4     4   (frag=1)
//   5     5   (frag=1)
//   6     6   (frag=0)
//   7     7   (frag=0)
//   8     8   (frag=1)
//   9     9   (frag=0)
// ATTR1 ATTR2
//   0    10
//   1     1
//   2    12
// Detected that deleted tuple doesn't exist!
//   4    14
//   5     5
//   6    16
//   7     7
//   8    18
//   9     9
```

```
#include <mysql.h>
#include <NdbApi.hpp>

// Used for cout
#include <stdio.h>
#include <iostream>

#define PRINT_ERROR(code,msg) \
  std::cout << "Error in " << __FILE__ << ", line: " << __LINE__ \
            << ", code: " << code \
            << ", msg: " << msg << "." << std::endl
#define MYSQLERROR(mysql) { \
  PRINT_ERROR(mysql_errno(&mysql),mysql_error(&mysql)); \
  exit(1); }
#define APIERROR(error) { \
  PRINT_ERROR(error.code,error.message); \
  exit(1); }

/* C struct representing layout of data from table
 * api_s_i_ndbrecord in memory
 * This can make it easier to work with rows in the application,
 * but is not necessary - NdbRecord can map columns to any
 * pattern of offsets.
 * In this program, the same row offsets are used for columns
 * specified as part of a key, and as part of an attribute or
 * result.  This makes the example simpler, but is not
 * essential.
 */
struct MyTableRow
{
  unsigned int attr1;
  unsigned int attr2;
};

int main(int argc, char** argv)
{
  if (argc != 3)
    {
    std::cout << "Arguments are <socket mysqld> <connect_string cluster>.\n";
    exit(1);
  }
  char * mysqld_sock  = argv[1];
  const char *connection_string = argv[2];
  ndb_init();
  MYSQL mysql;

  /**************************************************************
   * Connect to mysql server and create table                   *
   **************************************************************/
  {
    if ( !mysql_init(&mysql) ) {
      std::cout << "mysql_init failed\n";
      exit(1);
    }
    if ( !mysql_real_connect(&mysql, "localhost", "root", "", "",
                             0, mysqld_sock, 0) )
      MYSQLERROR(mysql);

    mysql_query(&mysql, "CREATE DATABASE ndb_examples");
    if (mysql_query(&mysql, "USE ndb_examples") != 0)
      MYSQLERROR(mysql);

    mysql_query(&mysql, "DROP TABLE api_s_i_ndbrecord");
    if (mysql_query(&mysql,
                    "CREATE TABLE"
                    "  api_s_i_ndbrecord"
                    "    (ATTR1 INT UNSIGNED,"
                    "     ATTR2 INT UNSIGNED NOT NULL,"
                    "     PRIMARY KEY USING HASH (ATTR1),"
                    "     UNIQUE MYINDEXNAME USING HASH (ATTR2))"
                    "  ENGINE=NDB"))
      MYSQLERROR(mysql);
```

```
}

/********************************************************
 * Connect to ndb cluster                               *
 ********************************************************/

Ndb_cluster_connection *cluster_connection=
  new Ndb_cluster_connection(connection_string); // Object representing the cluster

if (cluster_connection->connect(5,3,1))
{
  std::cout << "Connect to cluster management server failed.\n";
  exit(1);
}

if (cluster_connection->wait_until_ready(30,30))
{
  std::cout << "Cluster was not ready within 30 secs.\n";
  exit(1);
}

Ndb* myNdb = new Ndb( cluster_connection,
                      "ndb_examples" );  // Object representing the database
if (myNdb->init() == -1) {
  APIERROR(myNdb->getNdbError());
  exit(1);
}

NdbDictionary::Dictionary* myDict= myNdb->getDictionary();
const NdbDictionary::Table *myTable= myDict->getTable("api_s_i_ndbrecord");
if (myTable == NULL)
  APIERROR(myDict->getNdbError());
const NdbDictionary::Index *myIndex= myDict->getIndex("MYINDEXNAME$unique","api_s_i_ndbrecord");
if (myIndex == NULL)
  APIERROR(myDict->getNdbError());

/* Create NdbRecord descriptors. */
const NdbDictionary::Column *col1= myTable->getColumn("ATTR1");
if (col1 == NULL)
  APIERROR(myDict->getNdbError());
const NdbDictionary::Column *col2= myTable->getColumn("ATTR2");
if (col2 == NULL)
  APIERROR(myDict->getNdbError());

/* NdbRecord for primary key lookup. */
NdbDictionary::RecordSpecification spec[2];
spec[0].column= col1;
spec[0].offset= offsetof(MyTableRow, attr1);
  // So that it goes nicely into the struct
spec[0].nullbit_byte_offset= 0;
spec[0].nullbit_bit_in_byte= 0;
const NdbRecord *pk_record=
  myDict->createRecord(myTable, spec, 1, sizeof(spec[0]));
if (pk_record == NULL)
  APIERROR(myDict->getNdbError());

/* NdbRecord for all table attributes (insert/read). */
spec[0].column= col1;
spec[0].offset= offsetof(MyTableRow, attr1);
spec[0].nullbit_byte_offset= 0;
spec[0].nullbit_bit_in_byte= 0;
spec[1].column= col2;
spec[1].offset= offsetof(MyTableRow, attr2);
spec[1].nullbit_byte_offset= 0;
spec[1].nullbit_bit_in_byte= 0;
const NdbRecord *attr_record=
  myDict->createRecord(myTable, spec, 2, sizeof(spec[0]));
if (attr_record == NULL)
  APIERROR(myDict->getNdbError());

/* NdbRecord for unique key lookup. */
spec[0].column= col2;
```

```
  spec[0].offset= offsetof(MyTableRow, attr2);
  spec[0].nullbit_byte_offset= 0;
  spec[0].nullbit_bit_in_byte= 0;
  const NdbRecord *key_record=
    myDict->createRecord(myIndex, spec, 1, sizeof(spec[0]));
  if (key_record == NULL)
    APIERROR(myDict->getNdbError());

  MyTableRow row;

  /**************************************************************************
   * Using 5 transactions, insert 10 tuples in table: (0,0),(1,1),...,(9,9) *
   **************************************************************************/
  for (int i = 0; i < 5; i++) {
    NdbTransaction *myTransaction= myNdb->startTransaction();
    if (myTransaction == NULL) APIERROR(myNdb->getNdbError());

    /*
      We initialise the row data and pass to each insertTuple operation
      The data is copied in the call to insertTuple and so the original
      row object can be reused for the two operations.
    */
    row.attr1= row.attr2= i;

    const NdbOperation *myOperation=
      myTransaction->insertTuple(attr_record, (const char*)&row);
    if (myOperation == NULL)
      APIERROR(myTransaction->getNdbError());

    row.attr1= row.attr2= i+5;
    myOperation=
      myTransaction->insertTuple(attr_record, (const char*)&row);
    if (myOperation == NULL)
      APIERROR(myTransaction->getNdbError());

    if (myTransaction->execute( NdbTransaction::Commit ) == -1)
      APIERROR(myTransaction->getNdbError());

    myNdb->closeTransaction(myTransaction);
  }

  /***************************************
   * Read and print all tuples using index *
   ***************************************/
  std::cout << "ATTR1 ATTR2" << std::endl;

  for (int i = 0; i < 10; i++) {
    NdbTransaction *myTransaction= myNdb->startTransaction();
    if (myTransaction == NULL)
      APIERROR(myNdb->getNdbError());

    /* The optional OperationOptions parameter to NdbRecord methods
     * can be used to specify extra reads of columns which are not in
     * the NdbRecord specification, which need to be stored somewhere
     * other than specified in the NdbRecord specification, or
     * which cannot be specified as part of an NdbRecord (pseudo
     * columns)
     */
    Uint32 frag;
    NdbOperation::GetValueSpec getSpec[1];
    getSpec[0].column=NdbDictionary::Column::FRAGMENT;
    getSpec[0].appStorage=&frag;

    NdbOperation::OperationOptions options;
    options.optionsPresent = NdbOperation::OperationOptions::OO_GETVALUE;
    options.extraGetValues = &getSpec[0];
    options.numExtraGetValues = 1;

    /* We're going to read using the secondary unique hash index
     * Set the value of its column
     */
    row.attr2= i;
```

```
  MyTableRow resultRow;

  unsigned char mask[1]= { 0x01 };           // Only read ATTR1 into resultRow
  const NdbOperation *myOperation=
    myTransaction->readTuple(key_record, (const char*) &row,
                             attr_record, (char*) &resultRow,
                             NdbOperation::LM_Read, mask,
                             &options,
                             sizeof(NdbOperation::OperationOptions));
  if (myOperation == NULL)
    APIERROR(myTransaction->getNdbError());

  if (myTransaction->execute( NdbTransaction::Commit,
                              NdbOperation::AbortOnError ) != -1)
  {
    printf(" %2d    %2d   (frag=%u)\n", resultRow.attr1, i, frag);
  }

  myNdb->closeTransaction(myTransaction);
}

/*************************************************************
 * Update the second attribute in half of the tuples (adding 10) *
 *************************************************************/
for (int i = 0; i < 10; i+=2) {
  NdbTransaction *myTransaction= myNdb->startTransaction();
  if (myTransaction == NULL)
    APIERROR(myNdb->getNdbError());

  /* Specify key column to lookup in secondary index */
  row.attr2= i;

  /* Specify new column value to set */
  MyTableRow newRowData;
  newRowData.attr2= i+10;
  unsigned char mask[1]= { 0x02 };           // Only update ATTR2

  const NdbOperation *myOperation=
    myTransaction->updateTuple(key_record, (const char*)&row,
                               attr_record,(char*) &newRowData, mask);
  if (myOperation == NULL)
    APIERROR(myTransaction->getNdbError());

  if ( myTransaction->execute( NdbTransaction::Commit ) == -1 )
    APIERROR(myTransaction->getNdbError());

  myNdb->closeTransaction(myTransaction);
}

/*********************************************
 * Delete one tuple (the one with unique key 3) *
 *********************************************/
{
  NdbTransaction *myTransaction= myNdb->startTransaction();
  if (myTransaction == NULL)
    APIERROR(myNdb->getNdbError());

  row.attr2= 3;
  const NdbOperation *myOperation=
    myTransaction->deleteTuple(key_record, (const char*) &row,
                               attr_record);
  if (myOperation == NULL)
    APIERROR(myTransaction->getNdbError());

  if (myTransaction->execute(NdbTransaction::Commit) == -1)
    APIERROR(myTransaction->getNdbError());

  myNdb->closeTransaction(myTransaction);
}

/***************************
```

```
   * Read and print all tuples *
   ****************************/
  {
    std::cout << "ATTR1 ATTR2" << std::endl;

    for (int i = 0; i < 10; i++) {
      NdbTransaction *myTransaction= myNdb->startTransaction();
      if (myTransaction == NULL)
        APIERROR(myNdb->getNdbError());

      row.attr1= i;

      /* Read using pk.  Note the same row space is used as
       * key and result storage space
       */
      const NdbOperation *myOperation=
        myTransaction->readTuple(pk_record, (const char*) &row,
                                 attr_record, (char*) &row);
      if (myOperation == NULL)
        APIERROR(myTransaction->getNdbError());

      if (myTransaction->execute( NdbTransaction::Commit,
                                  NdbOperation::AbortOnError ) == -1)
        if (i == 3) {
          std::cout << "Detected that deleted tuple doesn't exist!\n";
        } else {
          APIERROR(myTransaction->getNdbError());
        }

      if (i != 3)
        printf(" %2d    %2d\n", row.attr1, row.attr2);

      myNdb->closeTransaction(myTransaction);
    }
  }

  delete myNdb;
  delete cluster_connection;

  ndb_end(0);
  return 0;
}
```

2.5.7 NDB API Example Comparing RecAttr and NdbRecord

This example illustrates the key differences between the old-style `NdbRecAttr` API and the newer approach using `NdbRecord` when performing some common tasks in an NDB API application.

The source code can be found can be found in the file `storage/ndb/ndbapi-examples/ndbapi_recattr_vs_record/main.cpp` in the NDB Cluster source tree.

```
#include <mysql.h>
#include <NdbApi.hpp>

// Used for cout
#include <stdio.h>
#include <iostream>

// Do we use old-style (NdbRecAttr?) or new style (NdbRecord?)
enum ApiType {api_attr, api_record};

static void run_application(MYSQL &, Ndb_cluster_connection &, ApiType);

#define PRINT_ERROR(code,msg) \
  std::cout << "Error in " << __FILE__ \
            << ", line: " << __LINE__ \
            << ", code: " << code \
            << ", msg: " << msg << "." << std::endl
#define MYSQLERROR(mysql) { \
  PRINT_ERROR(mysql_errno(&mysql),mysql_error(&mysql)); \
```

```
  exit(-1); }
#define APIERROR(error) { \
  PRINT_ERROR(error.code,error.message); \
  exit(-1); }

int main(int argc, char** argv)
{
  if (argc != 4)
  {
    std::cout << "Arguments are <socket mysqld> "
              << "<connect_string cluster> <attr|record>.\n";
    exit(-1);
  }
  // ndb_init must be called first
  ndb_init();

  // connect to mysql server and cluster and run application
  {
    char * mysqld_sock  = argv[1];
    const char *connection_string = argv[2];
    ApiType accessType=api_attr;

    // Object representing the cluster
    Ndb_cluster_connection cluster_connection(connection_string);

    // Connect to cluster management server (ndb_mgmd)
    if (cluster_connection.connect(4 /* retries               */,
       5 /* delay between retries */,
       1 /* verbose               */))
    {
      std::cout << "Management server not ready within 30 sec.\n";
      exit(-1);
    }

    // Optionally connect and wait for the storage nodes (ndbd's)
    if (cluster_connection.wait_until_ready(30,0) < 0)
    {
      std::cout << "Cluster not ready within 30 sec.\n";
      exit(-1);
    }

    // connect to mysql server
    MYSQL mysql;
    if ( !mysql_init(&mysql) ) {
      std::cout << "mysql_init failed\n";
      exit(-1);
    }
    if ( !mysql_real_connect(&mysql, "localhost", "root", "", "",
        0, mysqld_sock, 0) )
      MYSQLERROR(mysql);

    if (0==strncmp("attr", argv[3], 4))
    {
      accessType=api_attr;
    }
    else if (0==strncmp("record", argv[3], 6))
    {
      accessType=api_record;
    }
    else
    {
      std::cout << "Bad access type argument : "
                << argv[3] << "\n";
      exit(-1);
    }

    // run the application code
    run_application(mysql, cluster_connection, accessType);
  }

  ndb_end(0);
```

```
  return 0;
}

static void init_ndbrecord_info(Ndb &);
static void create_table(MYSQL &);
static void do_insert(Ndb &, ApiType);
static void do_update(Ndb &, ApiType);
static void do_delete(Ndb &, ApiType);
static void do_read(Ndb &, ApiType);
static void do_mixed_read(Ndb &);
static void do_mixed_update(Ndb &);
static void do_scan(Ndb &, ApiType);
static void do_mixed_scan(Ndb &);
static void do_indexScan(Ndb &, ApiType);
static void do_mixed_indexScan(Ndb&);
static void do_read_and_delete(Ndb &);
static void do_scan_update(Ndb&, ApiType);
static void do_scan_delete(Ndb&, ApiType);
static void do_scan_lock_reread(Ndb&, ApiType);
static void do_all_extras_read(Ndb &myNdb);
static void do_secondary_indexScan(Ndb &myNdb, ApiType accessType);
static void do_secondary_indexScanEqual(Ndb &myNdb, ApiType accessType);
static void do_interpreted_update(Ndb &myNdb, ApiType accessType);
static void do_interpreted_scan(Ndb &myNdb, ApiType accessType);
static void do_read_using_default(Ndb &myNdb);

/* This structure is used describe how we want data read using
 * NDBRecord to be placed into memory.  This can make it easier
 * to work with data, but is not essential.
 */
struct RowData
{
  int attr1;
  int attr2;
  int attr3;
};


/* Handy struct for representing the data in the
 * secondary index
 */
struct IndexRow
{
  unsigned int attr3;
  unsigned int attr2;
};

static void run_application(MYSQL &mysql,
       Ndb_cluster_connection &cluster_connection,
                           ApiType accessType)
{
  /********************************************
   * Connect to database via mysql-c          *
   ********************************************/
  mysql_query(&mysql, "CREATE DATABASE ndb_examples");
  if (mysql_query(&mysql, "USE ndb_examples") != 0) MYSQLERROR(mysql);
  create_table(mysql);

  /********************************************
   * Connect to database via NDB API          *
   ********************************************/
  // Object representing the database
  Ndb myNdb( &cluster_connection, "ndb_examples" );
  if (myNdb.init()) APIERROR(myNdb.getNdbError());

  init_ndbrecord_info(myNdb);
  /*
   * Do different operations on database
   */
  do_insert(myNdb, accessType);
  do_update(myNdb, accessType);
  do_delete(myNdb, accessType);
```

```
  do_read(myNdb, accessType);
  do_mixed_read(myNdb);
  do_mixed_update(myNdb);
  do_read(myNdb, accessType);
  do_scan(myNdb, accessType);
  do_mixed_scan(myNdb);
  do_indexScan(myNdb, accessType);
  do_mixed_indexScan(myNdb);
  do_read_and_delete(myNdb);
  do_scan_update(myNdb, accessType);
  do_scan_delete(myNdb, accessType);
  do_scan_lock_reread(myNdb, accessType);
  do_all_extras_read(myNdb);
  do_secondary_indexScan(myNdb, accessType);
  do_secondary_indexScanEqual(myNdb, accessType);
  do_scan(myNdb, accessType);
  do_interpreted_update(myNdb, accessType);
  do_interpreted_scan(myNdb, accessType);
  do_read_using_default(myNdb);
  do_scan(myNdb, accessType);
}

/*****************************************************
 * Create a table named api_recattr_vs_record if it does not exist *
 *****************************************************/
static void create_table(MYSQL &mysql)
{
  if (mysql_query(&mysql,
    "DROP TABLE IF EXISTS"
    "  api_recattr_vs_record"))
    MYSQLERROR(mysql);

  if (mysql_query(&mysql,
    "CREATE TABLE"
    "  api_recattr_vs_record"
    "    (ATTR1 INT UNSIGNED NOT NULL PRIMARY KEY,"
    "     ATTR2 INT UNSIGNED NOT NULL,"
                "     ATTR3 INT UNSIGNED NOT NULL)"
    "  ENGINE=NDB"))
    MYSQLERROR(mysql);

  /* Add ordered secondary index on 2 attributes, in reverse order */
  if (mysql_query(&mysql,
                "CREATE INDEX"
                "  MYINDEXNAME"
                "  ON api_recattr_vs_record"
                "  (ATTR3, ATTR2)"))
    MYSQLERROR(mysql);
}


/* Clunky statics for shared NdbRecord stuff */
static const NdbDictionary::Column *pattr1Col;
static const NdbDictionary::Column *pattr2Col;
static const NdbDictionary::Column *pattr3Col;

static const NdbRecord *pkeyColumnRecord;
static const NdbRecord *pallColsRecord;
static const NdbRecord *pkeyIndexRecord;
static const NdbRecord *psecondaryIndexRecord;

static int attr1ColNum;
static int attr2ColNum;
static int attr3ColNum;

/*************************************************************
 * Initialise NdbRecord structures for table and index access *
 *************************************************************/
static void init_ndbrecord_info(Ndb &myNdb)
{
  /* Here we create various NdbRecord structures for accessing
   * data using the tables and indexes on api_recattr_vs_record
```

```
 * We could use the default NdbRecord structures, but then
 * we wouldn't have the nice ability to read and write rows
 * to and from the RowData and IndexRow structs
 */
NdbDictionary::Dictionary* myDict= myNdb.getDictionary();
const NdbDictionary::Table *myTable=
                 myDict->getTable("api_recattr_vs_record");

NdbDictionary::RecordSpecification recordSpec[3];

if (myTable == NULL)
  APIERROR(myDict->getNdbError());

pattr1Col = myTable->getColumn("ATTR1");
if (pattr1Col == NULL) APIERROR(myDict->getNdbError());
pattr2Col = myTable->getColumn("ATTR2");
if (pattr2Col == NULL) APIERROR(myDict->getNdbError());
pattr3Col = myTable->getColumn("ATTR3");
if (pattr3Col == NULL) APIERROR(myDict->getNdbError());

attr1ColNum = pattr1Col->getColumnNo();
attr2ColNum = pattr2Col->getColumnNo();
attr3ColNum = pattr3Col->getColumnNo();

// ATTR 1
recordSpec[0].column = pattr1Col;
recordSpec[0].offset = offsetof(RowData, attr1);
recordSpec[0].nullbit_byte_offset = 0; // Not nullable
recordSpec[0].nullbit_bit_in_byte = 0;

// ATTR 2
recordSpec[1].column = pattr2Col;
recordSpec[1].offset = offsetof(RowData, attr2);
recordSpec[1].nullbit_byte_offset = 0;   // Not nullable
recordSpec[1].nullbit_bit_in_byte = 0;

// ATTR 3
recordSpec[2].column = pattr3Col;
recordSpec[2].offset = offsetof(RowData, attr3);
recordSpec[2].nullbit_byte_offset = 0;   // Not nullable
recordSpec[2].nullbit_bit_in_byte = 0;

/* Create table record with just the primary key column */
pkeyColumnRecord =
  myDict->createRecord(myTable, recordSpec, 1, sizeof(recordSpec[0]));

if (pkeyColumnRecord == NULL) APIERROR(myDict->getNdbError());

/* Create table record with all the columns */
pallColsRecord =
  myDict->createRecord(myTable, recordSpec, 3, sizeof(recordSpec[0]));

if (pallColsRecord == NULL) APIERROR(myDict->getNdbError());

/* Create NdbRecord for primary index access */
const NdbDictionary::Index *myPIndex=
                 myDict->getIndex("PRIMARY", "api_recattr_vs_record");

if (myPIndex == NULL)
  APIERROR(myDict->getNdbError());

pkeyIndexRecord =
  myDict->createRecord(myPIndex, recordSpec, 1, sizeof(recordSpec[0]));

if (pkeyIndexRecord == NULL) APIERROR(myDict->getNdbError());

/* Create Index NdbRecord for secondary index access
 * Note that we use the columns from the table to define the index
 * access record
 */
const NdbDictionary::Index *mySIndex=
             myDict->getIndex("MYINDEXNAME", "api_recattr_vs_record");
```

```
  recordSpec[0].column= pattr3Col;
  recordSpec[0].offset= offsetof(IndexRow, attr3);
  recordSpec[0].nullbit_byte_offset=0;
  recordSpec[0].nullbit_bit_in_byte=0;

  recordSpec[1].column= pattr2Col;
  recordSpec[1].offset= offsetof(IndexRow, attr2);
  recordSpec[1].nullbit_byte_offset=0;
  recordSpec[1].nullbit_bit_in_byte=1;

  /* Create NdbRecord for accessing via secondary index */
  psecondaryIndexRecord =
    myDict->createRecord(mySIndex,
                         recordSpec,
                         2,
                         sizeof(recordSpec[0]));


  if (psecondaryIndexRecord == NULL)
    APIERROR(myDict->getNdbError());

}


/**************************************************************************
 * Using 5 transactions, insert 10 tuples in table: (0,0),(1,1),...,(9,9) *
 **************************************************************************/
static void do_insert(Ndb &myNdb, ApiType accessType)
{
  NdbDictionary::Dictionary* myDict= myNdb.getDictionary();
  const NdbDictionary::Table *myTable=
                           myDict->getTable("api_recattr_vs_record");

  std::cout << "Running do_insert\n";

  if (myTable == NULL)
    APIERROR(myDict->getNdbError());

  for (int i = 0; i < 5; i++) {
    NdbTransaction *myTransaction= myNdb.startTransaction();
    if (myTransaction == NULL) APIERROR(myNdb.getNdbError());

    switch (accessType)
    {
    case api_attr :
      {
        NdbOperation *myOperation= myTransaction->getNdbOperation(myTable);
        if (myOperation == NULL) APIERROR(myTransaction->getNdbError());

        myOperation->insertTuple();
        myOperation->equal("ATTR1", i);
        myOperation->setValue("ATTR2", i);
        myOperation->setValue("ATTR3", i);

        myOperation= myTransaction->getNdbOperation(myTable);

        if (myOperation == NULL) APIERROR(myTransaction->getNdbError());
        myOperation->insertTuple();
        myOperation->equal("ATTR1", i+5);
        myOperation->setValue("ATTR2", i+5);
        myOperation->setValue("ATTR3", i+5);
        break;
      }
    case api_record :
      {
        RowData row;

        row.attr1= row.attr2= row.attr3= i;

        const NdbOperation *pop1=
          myTransaction->insertTuple(pallColsRecord, (char *) &row);
```

```
        if (pop1 == NULL) APIERROR(myTransaction->getNdbError());

        row.attr1= row.attr2= row.attr3= i+5;

        const NdbOperation *pop2=
          myTransaction->insertTuple(pallColsRecord, (char *) &row);
        if (pop2 == NULL) APIERROR(myTransaction->getNdbError());

        break;
      }
    default :
      {
        std::cout << "Bad branch : " << accessType << "\n";
        exit(-1);
      }
    }

    if (myTransaction->execute( NdbTransaction::Commit ) == -1)
      APIERROR(myTransaction->getNdbError());

    myNdb.closeTransaction(myTransaction);
  }

  std::cout << "-------\n";
}

/*************************************************************
 * Update the second attribute in half of the tuples (adding 10) *
 *************************************************************/
static void do_update(Ndb &myNdb, ApiType accessType)
{
  NdbDictionary::Dictionary* myDict= myNdb.getDictionary();
  const NdbDictionary::Table *myTable=
                           myDict->getTable("api_recattr_vs_record");

  std::cout << "Running do_update\n";

  for (int i = 0; i < 10; i+=2) {
    NdbTransaction *myTransaction= myNdb.startTransaction();
    if (myTransaction == NULL) APIERROR(myNdb.getNdbError());

    switch (accessType)
    {
      case api_attr :
      {
        NdbOperation *myOperation= myTransaction->getNdbOperation(myTable);
        if (myOperation == NULL) APIERROR(myTransaction->getNdbError());

        myOperation->updateTuple();
        myOperation->equal( "ATTR1", i );
        myOperation->setValue( "ATTR2", i+10);
        myOperation->setValue( "ATTR3", i+20);
        break;
      }
      case api_record :
      {
        RowData row;
        row.attr1=i;
        row.attr2=i+10;
        row.attr3=i+20;

        /* Since we're using an NdbRecord with all columns in it to
         * specify the updated columns, we need to create a mask to
         * indicate that we are only updating attr2 and attr3.
         */
        unsigned char attrMask=(1<<attr2ColNum) | (1<<attr3ColNum);

        const NdbOperation *pop =
          myTransaction->updateTuple(pkeyColumnRecord, (char*) &row,
                                     pallColsRecord, (char*) &row,
                                     &attrMask);
```

```
        if (pop==NULL) APIERROR(myTransaction->getNdbError());
        break;
      }
    default :
      {
        std::cout << "Bad branch : " << accessType << "\n";
        exit(-1);
      }
    }

    if( myTransaction->execute( NdbTransaction::Commit ) == -1 )
      APIERROR(myTransaction->getNdbError());

    myNdb.closeTransaction(myTransaction);
  }

  std::cout << "-------\n";
};

/*****************************************
 * Delete one tuple (the one with primary key 3) *
 *****************************************/
static void do_delete(Ndb &myNdb, ApiType accessType)
{
  NdbDictionary::Dictionary* myDict= myNdb.getDictionary();
  const NdbDictionary::Table *myTable=
                          myDict->getTable("api_recattr_vs_record");

  std::cout << "Running do_delete\n";

  if (myTable == NULL)
    APIERROR(myDict->getNdbError());

  NdbTransaction *myTransaction= myNdb.startTransaction();
  if (myTransaction == NULL) APIERROR(myNdb.getNdbError());

  switch (accessType)
  {
  case api_attr :
    {
      NdbOperation *myOperation= myTransaction->getNdbOperation(myTable);
      if (myOperation == NULL) APIERROR(myTransaction->getNdbError());

      myOperation->deleteTuple();
      myOperation->equal( "ATTR1", 3 );
      break;
    }
  case api_record :
    {
      RowData keyInfo;
      keyInfo.attr1=3;

      const NdbOperation *pop=
        myTransaction->deleteTuple(pkeyColumnRecord,
                                   (char*) &keyInfo,
                                   pallColsRecord);

      if (pop==NULL) APIERROR(myTransaction->getNdbError());
      break;
    }
  default :
    {
      std::cout << "Bad branch : " << accessType << "\n";
      exit(-1);
    }
  }


  if (myTransaction->execute(NdbTransaction::Commit) == -1)
    APIERROR(myTransaction->getNdbError());

  myNdb.closeTransaction(myTransaction);
```

```
  std::cout << "-------\n";
}


/**************************************************************
 * Update the second attribute in half of the tuples (adding 10) *
 **************************************************************/
static void do_mixed_update(Ndb &myNdb)
{
  /* This method performs an update using a mix of NdbRecord
   * supplied attributes, and extra setvalues provided by
   * the OperationOptions structure.
   */
  std::cout << "Running do_mixed_update (NdbRecord only)\n";

  for (int i = 0; i < 10; i+=2) {
    NdbTransaction *myTransaction= myNdb.startTransaction();
    if (myTransaction == NULL) APIERROR(myNdb.getNdbError());

    RowData row;
    row.attr1=i;
    row.attr2=i+30;

    /* Only attr2 is updated vian NDBRecord */
    unsigned char attrMask= (1<<attr2ColNum);

    NdbOperation::SetValueSpec setvalspecs[1];

    /* Value to set attr3 to */
    Uint32 dataSource= i + 40;

    setvalspecs[0].column = pattr3Col;
    setvalspecs[0].value = &dataSource;

    NdbOperation::OperationOptions opts;
    opts.optionsPresent= NdbOperation::OperationOptions::OO_SETVALUE;
    opts.extraSetValues= &setvalspecs[0];
    opts.numExtraSetValues= 1;


    // Define mixed operation in one call to NDBAPI
    const NdbOperation *pop =
      myTransaction->updateTuple(pkeyColumnRecord, (char*) &row,
                                 pallColsRecord, (char*) &row,
                                 &attrMask,
                                 &opts);

    if (pop==NULL) APIERROR(myTransaction->getNdbError());

    if( myTransaction->execute( NdbTransaction::Commit ) == -1 )
      APIERROR(myTransaction->getNdbError());

    myNdb.closeTransaction(myTransaction);
  }

  std::cout << "-------\n";
}


/*****************************************
 * Read and print all tuples using PK access *
 *****************************************/
static void do_read(Ndb &myNdb, ApiType accessType)
{
  NdbDictionary::Dictionary* myDict= myNdb.getDictionary();
  const NdbDictionary::Table *myTable=
                            myDict->getTable("api_recattr_vs_record");

  std::cout << "Running do_read\n";

  if (myTable == NULL)
```

```
   APIERROR(myDict->getNdbError());

 std::cout << "ATTR1 ATTR2 ATTR3" << std::endl;

 for (int i = 0; i < 10; i++) {
   NdbTransaction *myTransaction= myNdb.startTransaction();
   if (myTransaction == NULL) APIERROR(myNdb.getNdbError());

   RowData rowData;
   NdbRecAttr *myRecAttr;
   NdbRecAttr *myRecAttr2;

   switch (accessType)
   {
     case api_attr :
     {
       NdbOperation *myOperation= myTransaction->getNdbOperation(myTable);
       if (myOperation == NULL) APIERROR(myTransaction->getNdbError());

       myOperation->readTuple(NdbOperation::LM_Read);
       myOperation->equal("ATTR1", i);

       myRecAttr= myOperation->getValue("ATTR2", NULL);
       if (myRecAttr == NULL) APIERROR(myTransaction->getNdbError());

       myRecAttr2=myOperation->getValue("ATTR3", NULL);
       if (myRecAttr2 == NULL) APIERROR(myTransaction->getNdbError());

       break;
     }
     case api_record :
     {
       rowData.attr1=i;
       const NdbOperation *pop=
         myTransaction->readTuple(pkeyColumnRecord,
                                  (char*) &rowData,
                                  pallColsRecord,  // Read PK+ATTR2+ATTR3
                                  (char*) &rowData);
       if (pop==NULL) APIERROR(myTransaction->getNdbError());

       break;
     }
   default :
   {
     std::cout << "Bad branch : " << accessType << "\n";
     exit(-1);
   }
   }

   if(myTransaction->execute( NdbTransaction::Commit ) == -1)
     APIERROR(myTransaction->getNdbError());

   if (myTransaction->getNdbError().classification == NdbError::NoDataFound)
     if (i == 3)
       std::cout << "Deleted tuple does not exist." << std::endl;
     else
APIERROR(myTransaction->getNdbError());

   switch (accessType)
   {
     case api_attr :
     {
       if (i != 3) {
         printf(" %2d    %2d    %2d\n",
                i,
                myRecAttr->u_32_value(),
                myRecAttr2->u_32_value());
       }
       break;
     }
     case api_record :
     {
```

```
        if (i !=3) {
          printf(" %2d    %2d    %2d\n",
                 i,
                 rowData.attr2,
                 rowData.attr3);
        }
        break;
      }
      default :
      {
        std::cout << "Bad branch : " << accessType << "\n";
        exit(-1);
      }
    }

    myNdb.closeTransaction(myTransaction);
  }

  std::cout << "-------\n";
}

/*****************************
 * Read and print all tuples *
 *****************************/
static void do_mixed_read(Ndb &myNdb)
{
  std::cout << "Running do_mixed_read (NdbRecord only)\n";

  std::cout << "ATTR1 ATTR2 ATTR3 COMMIT_COUNT" << std::endl;

  for (int i = 0; i < 10; i++) {
    NdbTransaction *myTransaction= myNdb.startTransaction();
    if (myTransaction == NULL) APIERROR(myNdb.getNdbError());

    RowData rowData;
    NdbRecAttr *myRecAttr3, *myRecAttrCC;

    /* Start with NdbRecord read of ATTR2, and then add
     * getValue NdbRecAttr read of ATTR3 and Commit count
     */
    NdbOperation::GetValueSpec extraCols[2];

    extraCols[0].column=pattr3Col;
    extraCols[0].appStorage=NULL;
    extraCols[0].recAttr=NULL;

    extraCols[1].column=NdbDictionary::Column::COMMIT_COUNT;
    extraCols[1].appStorage=NULL;
    extraCols[1].recAttr=NULL;

    NdbOperation::OperationOptions opts;
    opts.optionsPresent = NdbOperation::OperationOptions::OO_GETVALUE;

    opts.extraGetValues= &extraCols[0];
    opts.numExtraGetValues= 2;

    /* We only read attr2 using the normal NdbRecord access */
    unsigned char attrMask= (1<<attr2ColNum);

    // Set PK search criteria
    rowData.attr1= i;

    const NdbOperation *pop=
      myTransaction->readTuple(pkeyColumnRecord,
                               (char*) &rowData,
                               pallColsRecord,  // Read all with mask
                               (char*) &rowData,
                               NdbOperation::LM_Read,
                               &attrMask, // result_mask
                               &opts);
    if (pop==NULL) APIERROR(myTransaction->getNdbError());
```

```
    myRecAttr3= extraCols[0].recAttr;
    myRecAttrCC= extraCols[1].recAttr;

    if (myRecAttr3 == NULL) APIERROR(myTransaction->getNdbError());
    if (myRecAttrCC == NULL) APIERROR(myTransaction->getNdbError());


    if(myTransaction->execute( NdbTransaction::Commit ) == -1)
      APIERROR(myTransaction->getNdbError());

    if (myTransaction->getNdbError().classification == NdbError::NoDataFound)
      if (i == 3)
        std::cout << "Deleted tuple does not exist." << std::endl;
      else
 APIERROR(myTransaction->getNdbError());


    if (i !=3) {
      printf(" %2d    %2d    %2d    %d\n",
             rowData.attr1,
             rowData.attr2,
             myRecAttr3->u_32_value(),
             myRecAttrCC->u_32_value()
             );
    }

    myNdb.closeTransaction(myTransaction);
  }

  std::cout << "-------\n";
}

/*****************************************
 * Read and print all tuples via table scan *
 *****************************************/
static void do_scan(Ndb &myNdb, ApiType accessType)
{
  NdbDictionary::Dictionary* myDict= myNdb.getDictionary();
  const NdbDictionary::Table *myTable=
                           myDict->getTable("api_recattr_vs_record");

  std::cout << "Running do_scan\n";

  if (myTable == NULL)
    APIERROR(myDict->getNdbError());

  std::cout << "ATTR1 ATTR2 ATTR3" << std::endl;

  NdbTransaction *myTransaction=myNdb.startTransaction();
  if (myTransaction == NULL) APIERROR(myNdb.getNdbError());

  NdbScanOperation *psop;
  NdbRecAttr *recAttrAttr1;
  NdbRecAttr *recAttrAttr2;
  NdbRecAttr *recAttrAttr3;

  switch (accessType)
  {
    case api_attr :
    {
      psop=myTransaction->getNdbScanOperation(myTable);

      if (psop == NULL) APIERROR(myTransaction->getNdbError());

      if (psop->readTuples(NdbOperation::LM_Read) != 0)
        APIERROR (myTransaction->getNdbError());

      recAttrAttr1=psop->getValue("ATTR1");
      recAttrAttr2=psop->getValue("ATTR2");
      recAttrAttr3=psop->getValue("ATTR3");

      break;
```

```
  }
  case api_record :
  {
    /* Note that no row ptr is passed to the NdbRecord scan operation
     * The scan will fetch a batch and give the user a series of pointers
     * to rows in the batch in nextResult() below
     */
    psop=myTransaction->scanTable(pallColsRecord,
                                  NdbOperation::LM_Read);

    if (psop == NULL) APIERROR(myTransaction->getNdbError());

    break;
  }
default :
{
  std::cout << "Bad branch : " << accessType << "\n";
  exit(-1);
}
}

if(myTransaction->execute( NdbTransaction::NoCommit ) != 0)
  APIERROR(myTransaction->getNdbError());

switch (accessType)
{
case api_attr :
{
  while (psop->nextResult(true) == 0)
  {
    printf(" %2d    %2d    %2d\n",
           recAttrAttr1->u_32_value(),
           recAttrAttr2->u_32_value(),
           recAttrAttr3->u_32_value());
  }

  psop->close();

  break;
}
case api_record :
{
  RowData *prowData; // Ptr to point to our data

  int rc=0;

  /* Ask nextResult to update out ptr to point to the next
   * row from the scan
   */
  while ((rc = psop->nextResult((const char**) &prowData,
                                true,
                                false)) == 0)
  {
    printf(" %2d    %2d    %2d\n",
           prowData->attr1,
           prowData->attr2,
           prowData->attr3);
  }

  if (rc != 1)  APIERROR(myTransaction->getNdbError());

  psop->close(true);

  break;
}
default :
{
  std::cout << "Bad branch : " << accessType << "\n";
  exit(-1);
}
}
```

```
  if(myTransaction->execute( NdbTransaction::Commit ) !=0)
    APIERROR(myTransaction->getNdbError());

  myNdb.closeTransaction(myTransaction);

  std::cout << "-------\n";
}

/*********************************************************
 * Read and print all tuples via table scan and mixed read *
 *********************************************************/
static void do_mixed_scan(Ndb &myNdb)
{
  std::cout << "Running do_mixed_scan(NdbRecord only)\n";

  std::cout << "ATTR1 ATTR2 ATTR3" << std::endl;

  NdbTransaction *myTransaction=myNdb.startTransaction();
  if (myTransaction == NULL) APIERROR(myNdb.getNdbError());

  NdbScanOperation *psop;
  NdbRecAttr *recAttrAttr3;

  /* Set mask so that NdbRecord scan reads attr1 and attr2 only */
  unsigned char attrMask=((1<<attr1ColNum) | (1<<attr2ColNum));

  /* Define extra get value to get attr3 */
  NdbOperation::GetValueSpec extraGets[1];
  extraGets[0].column = pattr3Col;
  extraGets[0].appStorage= 0;
  extraGets[0].recAttr= 0;

  NdbScanOperation::ScanOptions options;
  options.optionsPresent= NdbScanOperation::ScanOptions::SO_GETVALUE;
  options.extraGetValues= &extraGets[0];
  options.numExtraGetValues= 1;

  psop=myTransaction->scanTable(pallColsRecord,
                                NdbOperation::LM_Read,
                                &attrMask,
                                &options,
                                sizeof(NdbScanOperation::ScanOptions));
  if (psop == NULL) APIERROR(myTransaction->getNdbError());

  /* RecAttr for the extra get has been set by the operation definition */
  recAttrAttr3 = extraGets[0].recAttr;

  if (recAttrAttr3 == NULL) APIERROR(myTransaction->getNdbError());

  if(myTransaction->execute( NdbTransaction::NoCommit ) != 0)
    APIERROR(myTransaction->getNdbError());

  RowData *prowData; // Ptr to point to our data

  int rc=0;

  while ((rc = psop->nextResult((const char**) &prowData,
                                true,
                                false)) == 0)
  {
    printf(" %2d    %2d    %2d\n",
           prowData->attr1,
           prowData->attr2,
           recAttrAttr3->u_32_value());
  }

  if (rc != 1)  APIERROR(myTransaction->getNdbError());

  psop->close(true);

  if(myTransaction->execute( NdbTransaction::Commit ) !=0)
    APIERROR(myTransaction->getNdbError());
```

```
  myNdb.closeTransaction(myTransaction);

  std::cout << "-------\n";
}



/*********************************************************
 * Read and print all tuples via primary ordered index scan *
 *********************************************************/
static void do_indexScan(Ndb &myNdb, ApiType accessType)
{
  NdbDictionary::Dictionary* myDict= myNdb.getDictionary();
  const NdbDictionary::Index *myPIndex=
                myDict->getIndex("PRIMARY", "api_recattr_vs_record");

  std::cout << "Running do_indexScan\n";

  if (myPIndex == NULL)
    APIERROR(myDict->getNdbError());

  std::cout << "ATTR1 ATTR2 ATTR3" << std::endl;

  NdbTransaction *myTransaction=myNdb.startTransaction();
  if (myTransaction == NULL) APIERROR(myNdb.getNdbError());

  NdbIndexScanOperation *psop;

  /* RecAttrs for NdbRecAttr Api */
  NdbRecAttr *recAttrAttr1;
  NdbRecAttr *recAttrAttr2;
  NdbRecAttr *recAttrAttr3;

  switch (accessType)
  {
    case api_attr :
    {
      psop=myTransaction->getNdbIndexScanOperation(myPIndex);

      if (psop == NULL) APIERROR(myTransaction->getNdbError());

      /* Multi read range is not supported for the NdbRecAttr scan
       * API, so we just read one range.
       */
      Uint32 scanFlags=
        NdbScanOperation::SF_OrderBy |
        NdbScanOperation::SF_MultiRange |
        NdbScanOperation::SF_ReadRangeNo;

      if (psop->readTuples(NdbOperation::LM_Read,
                           scanFlags,
                           (Uint32) 0,          // batch
                           (Uint32) 0) != 0)    // parallel
        APIERROR (myTransaction->getNdbError());

      /* Add a bound
       * Tuples where ATTR1 >=2 and < 4
       * 2,[3 deleted]
       */
      Uint32 low=2;
      Uint32 high=4;

      if (psop->setBound("ATTR1",
                         NdbIndexScanOperation::BoundLE, (char*)&low))
        APIERROR(myTransaction->getNdbError());

      if (psop->setBound("ATTR1",
                         NdbIndexScanOperation::BoundGT, (char*)&high))
        APIERROR(myTransaction->getNdbError());

      if (psop->end_of_bound(0))
```

```
      APIERROR(psop->getNdbError());

    /* Second bound
     * Tuples where ATTR1 > 5 and <=9
     * 6,7,8,9
     */
    low=5;
    high=9;
    if (psop->setBound("ATTR1",
                       NdbIndexScanOperation::BoundLT, (char*)&low))
      APIERROR(myTransaction->getNdbError());

    if (psop->setBound("ATTR1",
                       NdbIndexScanOperation::BoundGE, (char*)&high))
      APIERROR(myTransaction->getNdbError());

    if (psop->end_of_bound(1))
      APIERROR(psop->getNdbError());

    /* Read all columns */
    recAttrAttr1=psop->getValue("ATTR1");
    recAttrAttr2=psop->getValue("ATTR2");
    recAttrAttr3=psop->getValue("ATTR3");

    break;
  }
  case api_record :
  {
    /* NdbRecord supports scanning multiple ranges using a
     * single index scan operation
     */
    Uint32 scanFlags =
      NdbScanOperation::SF_OrderBy |
      NdbScanOperation::SF_MultiRange |
      NdbScanOperation::SF_ReadRangeNo;

    NdbScanOperation::ScanOptions options;
    options.optionsPresent=NdbScanOperation::ScanOptions::SO_SCANFLAGS;
    options.scan_flags=scanFlags;

    psop=myTransaction->scanIndex(pkeyIndexRecord,
                                  pallColsRecord,
                                  NdbOperation::LM_Read,
                                  NULL, // no mask; read all columns
                                        // in result record
                                  NULL, // bound defined later
                                  &options,
                                  sizeof(NdbScanOperation::ScanOptions));

    if (psop == NULL) APIERROR(myTransaction->getNdbError());

    /* Add a bound
     * Tuples where ATTR1 >=2 and < 4
     * 2,[3 deleted]
     */
    Uint32 low=2;
    Uint32 high=4;

    NdbIndexScanOperation::IndexBound bound;
    bound.low_key=(char*)&low;
    bound.low_key_count=1;
    bound.low_inclusive=true;
    bound.high_key=(char*)&high;
    bound.high_key_count=1;
    bound.high_inclusive=false;
    bound.range_no=0;

    if (psop->setBound(pkeyIndexRecord, bound))
      APIERROR(myTransaction->getNdbError());

    /* Second bound
     * Tuples where ATTR1 > 5 and <=9
```

```
     * 6,7,8,9
     */
    low=5;
    high=9;

    bound.low_key=(char*)&low;
    bound.low_key_count=1;
    bound.low_inclusive=false;
    bound.high_key=(char*)&high;
    bound.high_key_count=1;
    bound.high_inclusive=true;
    bound.range_no=1;

    if (psop->setBound(pkeyIndexRecord, bound))
      APIERROR(myTransaction->getNdbError());

    break;
  }
default :
  {
    std::cout << "Bad branch : " << accessType << "\n";
    exit(-1);
  }
}

if(myTransaction->execute( NdbTransaction::NoCommit ) != 0)
  APIERROR(myTransaction->getNdbError());

if (myTransaction->getNdbError().code != 0)
  APIERROR(myTransaction->getNdbError());

switch (accessType)
{
  case api_attr :
  {
    while (psop->nextResult(true) == 0)
    {
      printf(" %2d    %2d    %2d    Range no : %2d\n",
             recAttrAttr1->u_32_value(),
             recAttrAttr2->u_32_value(),
             recAttrAttr3->u_32_value(),
             psop->get_range_no());
    }

    psop->close();

    break;
  }
  case api_record :
  {
    RowData *prowData; // Ptr to point to our data

    int rc=0;

    while ((rc = psop->nextResult((const char**) &prowData,
                                  true,
                                  false)) == 0)
    {
      // printf(" PTR : %d\n", (int) prowData);
      printf(" %2d    %2d    %2d    Range no : %2d\n",
             prowData->attr1,
             prowData->attr2,
             prowData->attr3,
             psop->get_range_no());
    }

    if (rc != 1)  APIERROR(myTransaction->getNdbError());

    psop->close(true);

    break;
  }
```

```
    default :
    {
      std::cout << "Bad branch : " << accessType << "\n";
      exit(-1);
    }
  }

  if(myTransaction->execute( NdbTransaction::Commit ) !=0)
    APIERROR(myTransaction->getNdbError());

  myNdb.closeTransaction(myTransaction);

  std::cout << "-------\n";
}



/*************************************************************************
 * Read and print all tuples via index scan using mixed NdbRecord access *
 *************************************************************************/
static void do_mixed_indexScan(Ndb &myNdb)
{
  NdbDictionary::Dictionary* myDict= myNdb.getDictionary();
  const NdbDictionary::Index *myPIndex=
                   myDict->getIndex("PRIMARY", "api_recattr_vs_record");

  std::cout << "Running do_mixed_indexScan\n";

  if (myPIndex == NULL)
    APIERROR(myDict->getNdbError());

  std::cout << "ATTR1 ATTR2 ATTR3" << std::endl;

  NdbTransaction *myTransaction=myNdb.startTransaction();
  if (myTransaction == NULL) APIERROR(myNdb.getNdbError());

  NdbIndexScanOperation *psop;
  NdbRecAttr *recAttrAttr3;

  Uint32 scanFlags =
    NdbScanOperation::SF_OrderBy |
    NdbScanOperation::SF_MultiRange |
    NdbScanOperation::SF_ReadRangeNo;

  /* We'll get Attr3 via ScanOptions */
  unsigned char attrMask=((1<<attr1ColNum) | (1<<attr2ColNum));

  NdbOperation::GetValueSpec extraGets[1];
  extraGets[0].column= pattr3Col;
  extraGets[0].appStorage= NULL;
  extraGets[0].recAttr= NULL;

  NdbScanOperation::ScanOptions options;
  options.optionsPresent=
    NdbScanOperation::ScanOptions::SO_SCANFLAGS |
    NdbScanOperation::ScanOptions::SO_GETVALUE;
  options.scan_flags= scanFlags;
  options.extraGetValues= &extraGets[0];
  options.numExtraGetValues= 1;

  psop=myTransaction->scanIndex(pkeyIndexRecord,
                                pallColsRecord,
                                NdbOperation::LM_Read,
                                &attrMask, // mask
                                NULL, // bound defined below
                                &options,
                                sizeof(NdbScanOperation::ScanOptions));

  if (psop == NULL) APIERROR(myTransaction->getNdbError());

  /* Grab RecAttr now */
  recAttrAttr3= extraGets[0].recAttr;
```

```
  /* Add a bound
   * ATTR1 >= 2, < 4
   * 2,[3 deleted]
   */
  Uint32 low=2;
  Uint32 high=4;

  NdbIndexScanOperation::IndexBound bound;
  bound.low_key=(char*)&low;
  bound.low_key_count=1;
  bound.low_inclusive=true;
  bound.high_key=(char*)&high;
  bound.high_key_count=1;
  bound.high_inclusive=false;
  bound.range_no=0;

  if (psop->setBound(pkeyIndexRecord, bound))
    APIERROR(myTransaction->getNdbError());

  /* Second bound
   * ATTR1 > 5, <= 9
   * 6,7,8,9
   */
  low=5;
  high=9;

  bound.low_key=(char*)&low;
  bound.low_key_count=1;
  bound.low_inclusive=false;
  bound.high_key=(char*)&high;
  bound.high_key_count=1;
  bound.high_inclusive=true;
  bound.range_no=1;

  if (psop->setBound(pkeyIndexRecord, bound))
    APIERROR(myTransaction->getNdbError());

  if(myTransaction->execute( NdbTransaction::NoCommit ) != 0)
    APIERROR(myTransaction->getNdbError());


  RowData *prowData; // Ptr to point to our data

  int rc=0;

  while ((rc = psop->nextResult((const char**) &prowData,
                                true,
                                false)) == 0)
  {
    printf(" %2d    %2d    %2d    Range no : %2d\n",
           prowData->attr1,
           prowData->attr2,
           recAttrAttr3->u_32_value(),
           psop->get_range_no());
  }

  if (rc != 1)  APIERROR(myTransaction->getNdbError());

  psop->close(true);

  if(myTransaction->execute( NdbTransaction::Commit ) !=0)
    APIERROR(myTransaction->getNdbError());

  myNdb.closeTransaction(myTransaction);

  std::cout << "-------\n";
}


/*****************************************************
 * Read + Delete one tuple (the one with primary key 8) *
```

```
 ****************************************************/
static void do_read_and_delete(Ndb &myNdb)
{
  /* This procedure performs a single operation, single round
   * trip read and then delete of a tuple, specified by
   * primary key
   */
  std::cout << "Running do_read_and_delete (NdbRecord only)\n";

  NdbTransaction *myTransaction= myNdb.startTransaction();
  if (myTransaction == NULL) APIERROR(myNdb.getNdbError());

  RowData row;
  row.attr1=8;
  row.attr2=0; // Don't care
  row.attr3=0; // Don't care

  /* We'll also read some extra columns while we're
   * reading + deleting
   */
  NdbOperation::OperationOptions options;
  NdbOperation::GetValueSpec extraGets[2];
  extraGets[0].column = pattr3Col;
  extraGets[0].appStorage = NULL;
  extraGets[0].recAttr = NULL;
  extraGets[1].column = NdbDictionary::Column::COMMIT_COUNT;
  extraGets[1].appStorage = NULL;
  extraGets[1].recAttr = NULL;

  options.optionsPresent= NdbOperation::OperationOptions::OO_GETVALUE;
  options.extraGetValues= &extraGets[0];
  options.numExtraGetValues= 2;

  unsigned char attrMask = (1<<attr2ColNum); // Only read Col2 into row

  const NdbOperation *pop=
    myTransaction->deleteTuple(pkeyColumnRecord, // Spec of key used
                               (char*) &row, // Key information
                               pallColsRecord, // Spec of columns to read
                               (char*) &row, // Row to read values into
                               &attrMask, // Cols to read as part of delete
                               &options,
                               sizeof(NdbOperation::OperationOptions));

  if (pop==NULL) APIERROR(myTransaction->getNdbError());

  if (myTransaction->execute(NdbTransaction::Commit) == -1)
    APIERROR(myTransaction->getNdbError());

  std::cout << "ATTR1 ATTR2 ATTR3 COMMITS" << std::endl;
  printf(" %2d    %2d    %2d    %2d\n",
         row.attr1,
         row.attr2,
         extraGets[0].recAttr->u_32_value(),
         extraGets[1].recAttr->u_32_value());

  myNdb.closeTransaction(myTransaction);

  std::cout << "-------\n";
}

/* Some handy consts for scan control */
static const int GOT_ROW= 0;
static const int NO_MORE_ROWS= 1;
static const int NEED_TO_FETCH_ROWS= 2;

/****************************************
 * Read and update all tuples via table scan *
 ****************************************/
static void do_scan_update(Ndb &myNdb, ApiType accessType)
{
  NdbDictionary::Dictionary* myDict= myNdb.getDictionary();
```

```
const NdbDictionary::Table *myTable=
                           myDict->getTable("api_recattr_vs_record");

if (myTable == NULL)
  APIERROR(myDict->getNdbError());

std::cout << "Running do_scan_update\n";

NdbTransaction *myTransaction=myNdb.startTransaction();
if (myTransaction == NULL) APIERROR(myNdb.getNdbError());

NdbScanOperation *psop;
NdbRecAttr *recAttrAttr1;
NdbRecAttr *recAttrAttr2;
NdbRecAttr *recAttrAttr3;

switch (accessType)
{
  case api_attr :
  {
    psop=myTransaction->getNdbScanOperation(myTable);

    if (psop == NULL) APIERROR(myTransaction->getNdbError());

    /* When we want to operate on the tuples returned from a
     * scan, we need to request the tuple's keyinfo is
     * returned, with SF_KeyInfo
     */
    if (psop->readTuples(NdbOperation::LM_Read,
                         NdbScanOperation::SF_KeyInfo) != 0)
      APIERROR (myTransaction->getNdbError());

    recAttrAttr1=psop->getValue("ATTR1");
    recAttrAttr2=psop->getValue("ATTR2");
    recAttrAttr3=psop->getValue("ATTR3");

    break;
  }
  case api_record :
  {
    NdbScanOperation::ScanOptions options;
    options.optionsPresent= NdbScanOperation::ScanOptions::SO_SCANFLAGS;
    options.scan_flags= NdbScanOperation::SF_KeyInfo;

    psop=myTransaction->scanTable(pallColsRecord,
                                  NdbOperation::LM_Read,
                                  NULL,  // mask - read all columns
                                  &options,
                                  sizeof(NdbScanOperation::ScanOptions));

    if (psop == NULL) APIERROR(myTransaction->getNdbError());

    break;
  }
default :
  {
    std::cout << "Bad branch : " << accessType << "\n";
    exit(-1);
  }
}

if(myTransaction->execute( NdbTransaction::NoCommit ) != 0)
  APIERROR(myTransaction->getNdbError());

switch (accessType)
{
  case api_attr :
  {


    int result= NEED_TO_FETCH_ROWS;
    Uint32 processed= 0;
```

```
  while (result == NEED_TO_FETCH_ROWS)
  {
    bool fetch=true;
    while ((result = psop->nextResult(fetch)) == GOT_ROW)
    {
      fetch= false;
      Uint32 col2Value=recAttrAttr2->u_32_value();

      NdbOperation *op=psop->updateCurrentTuple();
      if (op==NULL)
        APIERROR(myTransaction->getNdbError());
      op->setValue("ATTR2", (10*col2Value));

      processed++;
    }
    if (result < 0)
      APIERROR(myTransaction->getNdbError());

    if (processed !=0)
    {
      // Need to execute

      if(myTransaction->execute( NdbTransaction::NoCommit ) != 0)
        APIERROR(myTransaction->getNdbError());
      processed=0;
    }
  }

  psop->close();

  break;
}
case api_record :
{
  RowData *prowData; // Ptr to point to our data

  int result= NEED_TO_FETCH_ROWS;
  Uint32 processed=0;

  while (result == NEED_TO_FETCH_ROWS)
  {
    bool fetch= true;
    while ((result = psop->nextResult((const char**) &prowData,
                                      fetch, false)) == GOT_ROW)
    {
      fetch= false;

      /* Copy row into a stack variable */
      RowData r= *prowData;

      /* Modify attr2 */
      r.attr2*= 10;

      /* Update it */
      const NdbOperation *op = psop->updateCurrentTuple(myTransaction,
                                                        pallColsRecord,
                                                        (char*) &r);

      if (op==NULL)
        APIERROR(myTransaction->getNdbError());

      processed ++;
    }

    if (result < 0)
      APIERROR(myTransaction->getNdbError());


    if (processed !=0)
    {
      /* To get here, there are no more cached scan results,
```

```
           * and some row updates that we've not sent yet.
           * Send them before we try to get another batch, or
           * finish.
           */
          if (myTransaction->execute( NdbTransaction::NoCommit ) != 0)
            APIERROR(myTransaction->getNdbError());
          processed=0;
        }
      }

      psop->close(true);

      break;
    }
    default :
    {
      std::cout << "Bad branch : " << accessType << "\n";
      exit(-1);
    }
  }

  if(myTransaction->execute( NdbTransaction::Commit ) !=0)
    APIERROR(myTransaction->getNdbError());

  myNdb.closeTransaction(myTransaction);

  std::cout << "-------\n";
}

/*************************************************
 * Read all and delete some tuples via table scan *
 *************************************************/
static void do_scan_delete(Ndb &myNdb, ApiType accessType)
{
  NdbDictionary::Dictionary* myDict= myNdb.getDictionary();
  const NdbDictionary::Table *myTable=
                             myDict->getTable("api_recattr_vs_record");

  if (myTable == NULL)
    APIERROR(myDict->getNdbError());

  std::cout << "Running do_scan_delete\n";

  NdbTransaction *myTransaction=myNdb.startTransaction();
  if (myTransaction == NULL) APIERROR(myNdb.getNdbError());

  NdbScanOperation *psop;
  NdbRecAttr *recAttrAttr1;

  /* Scan, retrieving first column.
   * Delete particular records, based on first column
   * Read third column as part of delete
   */
  switch (accessType)
  {
    case api_attr :
    {
      psop=myTransaction->getNdbScanOperation(myTable);

      if (psop == NULL) APIERROR(myTransaction->getNdbError());

      /* Need KeyInfo when performing scanning delete */
      if (psop->readTuples(NdbOperation::LM_Read,
                           NdbScanOperation::SF_KeyInfo) != 0)
        APIERROR (myTransaction->getNdbError());

      recAttrAttr1=psop->getValue("ATTR1");

      break;
    }
    case api_record :
    {
```

```
    NdbScanOperation::ScanOptions options;
    options.optionsPresent=NdbScanOperation::ScanOptions::SO_SCANFLAGS;
    /* Need KeyInfo when performing scanning delete */
    options.scan_flags=NdbScanOperation::SF_KeyInfo;

    psop=myTransaction->scanTable(pkeyColumnRecord,
                                  NdbOperation::LM_Read,
                                  NULL,  // mask
                                  &options,
                                  sizeof(NdbScanOperation::ScanOptions));

    if (psop == NULL) APIERROR(myTransaction->getNdbError());

    break;
  }
default :
  {
    std::cout << "Bad branch : " << accessType << "\n";
    exit(-1);
  }
}

if(myTransaction->execute( NdbTransaction::NoCommit ) != 0)
  APIERROR(myTransaction->getNdbError());

switch (accessType)
{
  case api_attr :
  {
    int result= NEED_TO_FETCH_ROWS;
    Uint32 processed=0;

    while (result == NEED_TO_FETCH_ROWS)
    {
      bool fetch=true;
      while ((result = psop->nextResult(fetch)) == GOT_ROW)
      {
        fetch= false;
        Uint32 col1Value=recAttrAttr1->u_32_value();

        if (col1Value == 2)
        {
          /* Note : We cannot do a delete pre-read via
           * the NdbRecAttr interface.  We can only
           * delete here.
           */
          if (psop->deleteCurrentTuple())
            APIERROR(myTransaction->getNdbError());
          processed++;
        }
      }
      if (result < 0)
        APIERROR(myTransaction->getNdbError());

      if (processed !=0)
      {
        /* To get here, there are no more cached scan results,
         * and some row deletes that we've not sent yet.
         * Send them before we try to get another batch, or
         * finish.
         */
        if(myTransaction->execute( NdbTransaction::NoCommit ) != 0)
          APIERROR(myTransaction->getNdbError());
        processed=0;
      }
    }

    psop->close();

    break;
```

```
  }
  case api_record :
  {
    RowData *prowData; // Ptr to point to our data

    int result= NEED_TO_FETCH_ROWS;
    Uint32 processed=0;

    while (result == NEED_TO_FETCH_ROWS)
    {
      bool fetch=true;

      const NdbOperation* theDeleteOp;
      RowData readRow;
      NdbRecAttr* attr3;
      NdbRecAttr* commitCount;

      while ((result = psop->nextResult((const char**) &prowData,
                                        fetch,
                                        false)) == GOT_ROW)
      {
        fetch = false;

        /* Copy latest row to a stack local */
        RowData r;
        r= *prowData;

        if (r.attr1 == 2)
        {
          /* We're going to perform a read+delete on this
           * row.  We'll read attr1 and attr2 vian NDBRecord
           * and Attr3 and the commit count via extra
           * get values.
           */
          NdbOperation::OperationOptions options;
          NdbOperation::GetValueSpec extraGets[2];
          extraGets[0].column = pattr3Col;
          extraGets[0].appStorage = NULL;
          extraGets[0].recAttr = NULL;
          extraGets[1].column = NdbDictionary::Column::COMMIT_COUNT;
          extraGets[1].appStorage = NULL;
          extraGets[1].recAttr = NULL;

          options.optionsPresent= NdbOperation::OperationOptions::OO_GETVALUE;
          options.extraGetValues= &extraGets[0];
          options.numExtraGetValues= 2;

          // Read cols 1 + 2 vian NDBRecord
          unsigned char attrMask =
                  (1<<attr1ColNum) | (1<<attr2ColNum);

          theDeleteOp =
            psop->deleteCurrentTuple(myTransaction,
                                     pallColsRecord,
                                     (char*) &readRow,
                                     &attrMask,
                                     &options,
                                     sizeof(NdbOperation::OperationOptions)
                                    );

          if (theDeleteOp==NULL)
            APIERROR(myTransaction->getNdbError());

          /* Store extra Get RecAttrs */
          attr3= extraGets[0].recAttr;
          commitCount= extraGets[1].recAttr;

          processed ++;
        }
      }

      if (result < 0)
```

```
          APIERROR(myTransaction->getNdbError());


        if (processed !=0)
        {
          /* To get here, there are no more cached scan results,
           * and some row deletes that we've not sent yet.
           * Send them before we try to get another batch, or
           * finish.
           */
          if (myTransaction->execute( NdbTransaction::NoCommit ) != 0)
            APIERROR(myTransaction->getNdbError());
          processed=0;

          // Let's look at the data just read
          printf("Deleted data\n");
          printf("ATTR1  ATTR2  ATTR3 COMMITS\n");
          printf("  %2d    %2d    %2d    %2d\n",
                 readRow.attr1,
                 readRow.attr2,
                 attr3->u_32_value(),
                 commitCount->u_32_value());
        }
      }

      psop->close(true);

      break;
    }
    default :
    {
      std::cout << "Bad branch : " << accessType << "\n";
      exit(-1);
    }
  }

  if(myTransaction->execute( NdbTransaction::Commit ) !=0)
    APIERROR(myTransaction->getNdbError());

  myNdb.closeTransaction(myTransaction);

  std::cout << "-------\n";
}



/*************************************************************
 * Read all tuples via scan, reread one with lock takeover *
 *************************************************************/
static void do_scan_lock_reread(Ndb &myNdb, ApiType accessType)
{
  NdbDictionary::Dictionary* myDict= myNdb.getDictionary();
  const NdbDictionary::Table *myTable=
                              myDict->getTable("api_recattr_vs_record");

  if (myTable == NULL)
    APIERROR(myDict->getNdbError());

  std::cout << "Running do_scan_lock_reread\n";

  NdbTransaction *myTransaction=myNdb.startTransaction();
  if (myTransaction == NULL) APIERROR(myNdb.getNdbError());

  NdbScanOperation *psop;
  NdbRecAttr *recAttrAttr1;

  switch (accessType)
  {
    case api_attr :
    {
      psop=myTransaction->getNdbScanOperation(myTable);
```

```
    if (psop == NULL) APIERROR(myTransaction->getNdbError());

    /* Need KeyInfo for lock takeover */
    if (psop->readTuples(NdbOperation::LM_Read,
                         NdbScanOperation::SF_KeyInfo) != 0)
      APIERROR (myTransaction->getNdbError());

    recAttrAttr1=psop->getValue("ATTR1");

    break;
  }
  case api_record :
  {
    NdbScanOperation::ScanOptions options;
    options.optionsPresent= NdbScanOperation::ScanOptions::SO_SCANFLAGS;
    /* Need KeyInfo for lock takeover */
    options.scan_flags= NdbScanOperation::SF_KeyInfo;

    psop=myTransaction->scanTable(pkeyColumnRecord,
                                  NdbOperation::LM_Read,
                                  NULL,  // mask
                                  &options,
                                  sizeof(NdbScanOperation::ScanOptions));

    if (psop == NULL) APIERROR(myTransaction->getNdbError());

    break;
  }
default :
  {
    std::cout << "Bad branch : " << accessType << "\n";
    exit(-1);
  }
}

if(myTransaction->execute( NdbTransaction::NoCommit ) != 0)
  APIERROR(myTransaction->getNdbError());

switch (accessType)
{
  case api_attr :
  {
    int result= NEED_TO_FETCH_ROWS;
    Uint32 processed=0;
    NdbRecAttr *attr1, *attr2, *attr3, *commitCount;

    while (result == NEED_TO_FETCH_ROWS)
    {
      bool fetch=true;
      while ((result = psop->nextResult(fetch)) == GOT_ROW)
      {
        fetch= false;
        Uint32 col1Value=recAttrAttr1->u_32_value();

        if (col1Value == 9)
        {
          /* Let's read the rest of the info for it with
           * a separate operation
           */
          NdbOperation *op= psop->lockCurrentTuple();

          if (op==NULL)
            APIERROR(myTransaction->getNdbError());
          attr1=op->getValue("ATTR1");
          attr2=op->getValue("ATTR2");
          attr3=op->getValue("ATTR3");
          commitCount=op->getValue(NdbDictionary::Column::COMMIT_COUNT);
          processed++;
        }
      }
      if (result < 0)
        APIERROR(myTransaction->getNdbError());
```

```
      if (processed !=0)
      {
        // Need to execute

        if(myTransaction->execute( NdbTransaction::NoCommit ) != 0)
          APIERROR(myTransaction->getNdbError());
        processed=0;

        // Let's look at the whole row...
        printf("Locked and re-read data:\n");
        printf("ATTR1  ATTR2  ATTR3 COMMITS\n");
        printf("  %2d    %2d    %2d    %2d\n",
               attr1->u_32_value(),
               attr2->u_32_value(),
               attr3->u_32_value(),
               commitCount->u_32_value());
      }
    }

    psop->close();

    break;
  }
  case api_record :
  {
    RowData *prowData; // Ptr to point to our data

    int result= NEED_TO_FETCH_ROWS;
    Uint32 processed=0;
    RowData rereadData;
    NdbRecAttr *attr3, *commitCount;

    while (result == NEED_TO_FETCH_ROWS)
    {
      bool fetch=true;
      while ((result = psop->nextResult((const char**) &prowData,
                                        fetch,
                                        false)) == GOT_ROW)
      {
        fetch = false;

        /* Copy row to stack local */
        RowData r;
        r=*prowData;

        if (r.attr1 == 9)
        {
          /* Perform extra read of this row via lockCurrentTuple
           * Read all columns using NdbRecord for attr1 + attr2,
           * and extra get values for attr3 and the commit count
           */
          NdbOperation::OperationOptions options;
          NdbOperation::GetValueSpec extraGets[2];
          extraGets[0].column = pattr3Col;
          extraGets[0].appStorage = NULL;
          extraGets[0].recAttr = NULL;
          extraGets[1].column = NdbDictionary::Column::COMMIT_COUNT;
          extraGets[1].appStorage = NULL;
          extraGets[1].recAttr = NULL;

          options.optionsPresent=NdbOperation::OperationOptions::OO_GETVALUE;
          options.extraGetValues=&extraGets[0];
          options.numExtraGetValues=2;

          // Read cols 1 + 2 vian NDBRecord
          unsigned char attrMask =
                      (1<<attr1ColNum) | (1<<attr2ColNum);

          const NdbOperation *lockOp =
              psop->lockCurrentTuple(myTransaction,
                                        pallColsRecord,
```

```
                                        (char *) &rereadData,
                                        &attrMask,
                                        &options,
                                        sizeof(NdbOperation::OperationOptions)
                                       );
            if (lockOp == NULL)
              APIERROR(myTransaction->getNdbError());

            attr3= extraGets[0].recAttr;
            commitCount= extraGets[1].recAttr;

            processed++;
          }
        }

        if (result < 0)
          APIERROR(myTransaction->getNdbError());


        if (processed !=0)
        {
          // Need to execute

          if (myTransaction->execute( NdbTransaction::NoCommit ) != 0)
            APIERROR(myTransaction->getNdbError());
          processed=0;

          // Let's look at the whole row...
          printf("Locked and re-read data:\n");
          printf("ATTR1  ATTR2  ATTR3 COMMITS\n");
          printf("  %2d    %2d    %2d    %2d\n",
                 rereadData.attr1,
                 rereadData.attr2,
                 attr3->u_32_value(),
                 commitCount->u_32_value());

        }
      }

      psop->close(true);

      break;
    }
    default :
    {
      std::cout << "Bad branch : " << accessType << "\n";
      exit(-1);
    }
  }

  if(myTransaction->execute( NdbTransaction::Commit ) !=0)
    APIERROR(myTransaction->getNdbError());

  myNdb.closeTransaction(myTransaction);

  std::cout << "-------\n";
}

/*************************************************************
 * Read all tuples via primary key, using only extra getValues *
 *************************************************************/
static void do_all_extras_read(Ndb &myNdb)
{
  std::cout << "Running do_all_extras_read(NdbRecord only)\n";
  std::cout << "ATTR1 ATTR2 ATTR3 COMMIT_COUNT" << std::endl;

  for (int i = 0; i < 10; i++) {
    NdbTransaction *myTransaction= myNdb.startTransaction();
    if (myTransaction == NULL) APIERROR(myNdb.getNdbError());

    RowData rowData;
    NdbRecAttr *myRecAttr1, *myRecAttr2, *myRecAttr3, *myRecAttrCC;
```

```
  /* We read nothing vian NDBRecord, and everything via
   * 'extra' reads
   */
  NdbOperation::GetValueSpec extraCols[4];

  extraCols[0].column=pattr1Col;
  extraCols[0].appStorage=NULL;
  extraCols[0].recAttr=NULL;

  extraCols[1].column=pattr2Col;
  extraCols[1].appStorage=NULL;
  extraCols[1].recAttr=NULL;

  extraCols[2].column=pattr3Col;
  extraCols[2].appStorage=NULL;
  extraCols[2].recAttr=NULL;

  extraCols[3].column=NdbDictionary::Column::COMMIT_COUNT;
  extraCols[3].appStorage=NULL;
  extraCols[3].recAttr=NULL;

  NdbOperation::OperationOptions opts;
  opts.optionsPresent = NdbOperation::OperationOptions::OO_GETVALUE;

  opts.extraGetValues=&extraCols[0];
  opts.numExtraGetValues=4;

  unsigned char attrMask= 0; // No row results required.

  // Set PK search criteria
  rowData.attr1= i;

  const NdbOperation *pop=
    myTransaction->readTuple(pkeyColumnRecord,
                             (char*) &rowData,
                             pkeyColumnRecord,
                             NULL, // null result row
                             NdbOperation::LM_Read,
                             &attrMask,
                             &opts);
  if (pop==NULL) APIERROR(myTransaction->getNdbError());

  myRecAttr1=extraCols[0].recAttr;
  myRecAttr2=extraCols[1].recAttr;
  myRecAttr3=extraCols[2].recAttr;
  myRecAttrCC=extraCols[3].recAttr;

  if (myRecAttr1 == NULL) APIERROR(myTransaction->getNdbError());
  if (myRecAttr2 == NULL) APIERROR(myTransaction->getNdbError());
  if (myRecAttr3 == NULL) APIERROR(myTransaction->getNdbError());
  if (myRecAttrCC == NULL) APIERROR(myTransaction->getNdbError());

  if(myTransaction->execute( NdbTransaction::Commit ) == -1)
    APIERROR(myTransaction->getNdbError());

  bool deleted= (myTransaction->getNdbError().classification ==
                 NdbError::NoDataFound);
  if (deleted)
    printf("Detected that deleted tuple %d doesn't exist!\n", i);
  else
  {
    printf(" %2d    %2d    %2d    %d\n",
           myRecAttr1->u_32_value(),
           myRecAttr2->u_32_value(),
           myRecAttr3->u_32_value(),
           myRecAttrCC->u_32_value()
           );
  }

  myNdb.closeTransaction(myTransaction);
}
```

```
  std::cout << "-------\n";
}


/*****************************************************************
 * Read and print some tuples via bounded scan of secondary index *
 *****************************************************************/
static void do_secondary_indexScan(Ndb &myNdb, ApiType accessType)
{
  NdbDictionary::Dictionary* myDict= myNdb.getDictionary();
  const NdbDictionary::Index *mySIndex=
              myDict->getIndex("MYINDEXNAME", "api_recattr_vs_record");

  std::cout << "Running do_secondary_indexScan\n";
  std::cout << "ATTR1 ATTR2 ATTR3" << std::endl;

  NdbTransaction *myTransaction=myNdb.startTransaction();
  if (myTransaction == NULL) APIERROR(myNdb.getNdbError());

  NdbIndexScanOperation *psop;
  NdbRecAttr *recAttrAttr1;
  NdbRecAttr *recAttrAttr2;
  NdbRecAttr *recAttrAttr3;

  Uint32 scanFlags =
    NdbScanOperation::SF_OrderBy |
    NdbScanOperation::SF_Descending |
    NdbScanOperation::SF_MultiRange |
    NdbScanOperation::SF_ReadRangeNo;

  switch (accessType)
  {
    case api_attr :
    {
      psop=myTransaction->getNdbIndexScanOperation(mySIndex);

      if (psop == NULL) APIERROR(myTransaction->getNdbError());

      if (psop->readTuples(NdbOperation::LM_Read,
                           scanFlags,
                           (Uint32) 0,          // batch
                           (Uint32) 0) != 0)    // parallel
        APIERROR (myTransaction->getNdbError());

      /* Bounds :
       * > ATTR3=6
       * < ATTR3=42
       */
      Uint32 low=6;
      Uint32 high=42;

      if (psop->setBound("ATTR3",
                         NdbIndexScanOperation::BoundLT, (char*)&low))
        APIERROR(psop->getNdbError());

      if (psop->setBound("ATTR3",
                         NdbIndexScanOperation::BoundGT, (char*)&high))
        APIERROR(psop->getNdbError());

      recAttrAttr1=psop->getValue("ATTR1");
      recAttrAttr2=psop->getValue("ATTR2");
      recAttrAttr3=psop->getValue("ATTR3");

      break;
    }
    case api_record :
    {

      NdbScanOperation::ScanOptions options;
      options.optionsPresent=NdbScanOperation::ScanOptions::SO_SCANFLAGS;
      options.scan_flags=scanFlags;
```

```
    psop=myTransaction->scanIndex(psecondaryIndexRecord,
                                  pallColsRecord,
                                  NdbOperation::LM_Read,
                                  NULL, // mask
                                  NULL, // bound
                                  &options,
                                  sizeof(NdbScanOperation::ScanOptions));

    if (psop == NULL) APIERROR(myTransaction->getNdbError());

    /* Bounds :
     * > ATTR3=6
     * < ATTR3=42
     */
    Uint32 low=6;
    Uint32 high=42;

    NdbIndexScanOperation::IndexBound bound;
    bound.low_key=(char*)&low;
    bound.low_key_count=1;
    bound.low_inclusive=false;
    bound.high_key=(char*)&high;
    bound.high_key_count=1;
    bound.high_inclusive=false;
    bound.range_no=0;

    if (psop->setBound(psecondaryIndexRecord, bound))
      APIERROR(myTransaction->getNdbError());

    break;
  }
default :
  {
    std::cout << "Bad branch : " << accessType << "\n";
    exit(-1);
  }
}

if(myTransaction->execute( NdbTransaction::NoCommit ) != 0)
  APIERROR(myTransaction->getNdbError());

// Check rc anyway
if (myTransaction->getNdbError().status != NdbError::Success)
  APIERROR(myTransaction->getNdbError());

switch (accessType)
{
  case api_attr :
  {
    while (psop->nextResult(true) == 0)
    {
      printf(" %2d    %2d    %2d    Range no : %2d\n",
             recAttrAttr1->u_32_value(),
             recAttrAttr2->u_32_value(),
             recAttrAttr3->u_32_value(),
             psop->get_range_no());
    }

    psop->close();

    break;
  }
  case api_record :
  {
    RowData *prowData; // Ptr to point to our data

    int rc=0;

    while ((rc = psop->nextResult((const char**) &prowData,
                                  true,
                                  false)) == 0)
```

```
      {
        // printf(" PTR : %d\n", (int) prowData);
        printf(" %2d    %2d    %2d    Range no : %2d\n",
               prowData->attr1,
               prowData->attr2,
               prowData->attr3,
               psop->get_range_no());
      }

      if (rc != 1)  APIERROR(myTransaction->getNdbError());

      psop->close(true);

      break;
    }
    default :
    {
      std::cout << "Bad branch : " << accessType << "\n";
      exit(-1);
    }
  }

  if(myTransaction->execute( NdbTransaction::Commit ) !=0)
    APIERROR(myTransaction->getNdbError());

  myNdb.closeTransaction(myTransaction);

  std::cout << "-------\n";
}


/**************************************************************************
 * Index scan to read tuples from secondary index using equality bound *
 **************************************************************************/
static void do_secondary_indexScanEqual(Ndb &myNdb, ApiType accessType)
{
  NdbDictionary::Dictionary* myDict= myNdb.getDictionary();
  const NdbDictionary::Index *mySIndex=
               myDict->getIndex("MYINDEXNAME", "api_recattr_vs_record");

  std::cout << "Running do_secondary_indexScanEqual\n";
  std::cout << "ATTR1 ATTR2 ATTR3" << std::endl;

  NdbTransaction *myTransaction=myNdb.startTransaction();
  if (myTransaction == NULL) APIERROR(myNdb.getNdbError());

  NdbIndexScanOperation *psop;
  NdbRecAttr *recAttrAttr1;
  NdbRecAttr *recAttrAttr2;
  NdbRecAttr *recAttrAttr3;

  Uint32 scanFlags = NdbScanOperation::SF_OrderBy;

  Uint32 attr3Eq= 44;

  switch (accessType)
  {
    case api_attr :
    {
      psop=myTransaction->getNdbIndexScanOperation(mySIndex);

      if (psop == NULL) APIERROR(myTransaction->getNdbError());

      if (psop->readTuples(NdbOperation::LM_Read,
                           scanFlags,
                           (Uint32) 0,          // batch
                           (Uint32) 0) != 0)    // parallel
        APIERROR (myTransaction->getNdbError());

      if (psop->setBound("ATTR3",
                         NdbIndexScanOperation::BoundEQ, (char*)&attr3Eq))
        APIERROR(myTransaction->getNdbError());
```

```
    recAttrAttr1=psop->getValue("ATTR1");
    recAttrAttr2=psop->getValue("ATTR2");
    recAttrAttr3=psop->getValue("ATTR3");

    break;
  }
  case api_record :
  {

    NdbScanOperation::ScanOptions options;
    options.optionsPresent= NdbScanOperation::ScanOptions::SO_SCANFLAGS;
    options.scan_flags=scanFlags;

    psop=myTransaction->scanIndex(psecondaryIndexRecord,
                                  pallColsRecord, // Read all table rows back
                                  NdbOperation::LM_Read,
                                  NULL, // mask
                                  NULL, // bound specified below
                                  &options,
                                  sizeof(NdbScanOperation::ScanOptions));

    if (psop == NULL) APIERROR(myTransaction->getNdbError());

    /* Set equality bound via two inclusive bounds */
    NdbIndexScanOperation::IndexBound bound;
    bound.low_key= (char*)&attr3Eq;
    bound.low_key_count= 1;
    bound.low_inclusive= true;
    bound.high_key= (char*)&attr3Eq;
    bound.high_key_count= 1;
    bound.high_inclusive= true;
    bound.range_no= 0;

    if (psop->setBound(psecondaryIndexRecord, bound))
      APIERROR(myTransaction->getNdbError());

    break;
  }
default :
  {
    std::cout << "Bad branch : " << accessType << "\n";
    exit(-1);
  }
}

if(myTransaction->execute( NdbTransaction::NoCommit ) != 0)
  APIERROR(myTransaction->getNdbError());

// Check rc anyway
if (myTransaction->getNdbError().status != NdbError::Success)
  APIERROR(myTransaction->getNdbError());

switch (accessType)
{
  case api_attr :
  {
    int res;

    while ((res= psop->nextResult(true)) == GOT_ROW)
    {
      printf(" %2d    %2d    %2d\n",
             recAttrAttr1->u_32_value(),
             recAttrAttr2->u_32_value(),
             recAttrAttr3->u_32_value());
    }

    if (res != NO_MORE_ROWS)
      APIERROR(psop->getNdbError());

    psop->close();
```

```
      break;
    }
    case api_record :
    {
      RowData *prowData; // Ptr to point to our data

      int rc=0;

      while ((rc = psop->nextResult((const char**) &prowData,
                                    true,   // fetch
                                    false)) // forceSend
             == GOT_ROW)
      {
        printf(" %2d    %2d    %2d\n",
               prowData->attr1,
               prowData->attr2,
               prowData->attr3);
      }

      if (rc != NO_MORE_ROWS)
        APIERROR(myTransaction->getNdbError());

      psop->close(true);

      break;
    }
    default :
    {
      std::cout << "Bad branch : " << accessType << "\n";
      exit(-1);
    }
  }

  if(myTransaction->execute( NdbTransaction::Commit ) !=0)
    APIERROR(myTransaction->getNdbError());

  myNdb.closeTransaction(myTransaction);

  std::cout << "-------\n";
}


/**********************
 * Interpreted update *
 **********************/
static void do_interpreted_update(Ndb &myNdb, ApiType accessType)
{
  NdbDictionary::Dictionary* myDict= myNdb.getDictionary();

  const NdbDictionary::Table *myTable=
                              myDict->getTable("api_recattr_vs_record");

  const NdbDictionary::Index *myPIndex=
                       myDict->getIndex("PRIMARY", "api_recattr_vs_record");

  std::cout << "Running do_interpreted_update\n";

  if (myTable == NULL)
    APIERROR(myDict->getNdbError());
  if (myPIndex == NULL)
    APIERROR(myDict->getNdbError());

  std::cout << "ATTR1 ATTR2 ATTR3" << std::endl;

  NdbTransaction *myTransaction=myNdb.startTransaction();
  if (myTransaction == NULL) APIERROR(myNdb.getNdbError());

  NdbRecAttr *recAttrAttr1;
  NdbRecAttr *recAttrAttr2;
  NdbRecAttr *recAttrAttr3;
  NdbRecAttr *recAttrAttr11;
  NdbRecAttr *recAttrAttr12;
```

```
NdbRecAttr *recAttrAttr13;
RowData rowData;
RowData rowData2;

/* Register aliases */
const Uint32 R1=1, R2=2, R3=3, R4=4, R5=5, R6=6;

switch (accessType)
{
  case api_attr :
  {
    NdbOperation *pop;
    pop=myTransaction->getNdbOperation(myTable);

    if (pop == NULL) APIERROR(myTransaction->getNdbError());

    if (pop->interpretedUpdateTuple())
      APIERROR (pop->getNdbError());

    /* Interpreted update on row where ATTR1 == 4 */
    if (pop->equal("ATTR1", 4) != 0)
      APIERROR (pop->getNdbError());

    /* First, read the values of all attributes in the normal way */
    recAttrAttr1=pop->getValue("ATTR1");
    recAttrAttr2=pop->getValue("ATTR2");
    recAttrAttr3=pop->getValue("ATTR3");

    /* Now define interpreted program which will run after the
     * values have been read
     * This program is rather tortuous and doesn't achieve much other
     * than demonstrating control flow, register and some column
     * operations
     */
    // R5= 3
    if (pop->load_const_u32(R5, 3) != 0)
      APIERROR (pop->getNdbError());

    // R1= *ATTR1; R2= *ATTR2; R3= *ATTR3
    if (pop->read_attr("ATTR1", R1) != 0)
      APIERROR (pop->getNdbError());
    if (pop->read_attr("ATTR2", R2) != 0)
      APIERROR (pop->getNdbError());
    if (pop->read_attr("ATTR3", R3) != 0)
      APIERROR (pop->getNdbError());

    // R3= R3-R5
    if (pop->sub_reg(R3, R5, R3) != 0)
      APIERROR (pop->getNdbError());

    // R2= R1+R2
    if (pop->add_reg(R1, R2, R2) != 0)
      APIERROR (pop->getNdbError());

    // *ATTR2= R2
    if (pop->write_attr("ATTR2", R2) != 0)
      APIERROR (pop->getNdbError());

    // *ATTR3= R3
    if (pop->write_attr("ATTR3", R3) != 0)
      APIERROR (pop->getNdbError());

    // *ATTR3 = *ATTR3 - 30
    if (pop->subValue("ATTR3", (Uint32)30) != 0)
      APIERROR (pop->getNdbError());

    Uint32 comparisonValue= 10;

    // if *ATTR3 > comparisonValue, goto Label 0
    if (pop->branch_col_lt(pattr3Col->getColumnNo(),
                           &comparisonValue,
                           sizeof(Uint32),
```

```
                        false,
                        0) != 0)
    APIERROR (pop->getNdbError());

  // assert(false)
  // Fail the operation with error 627 if we get here.
  if (pop->interpret_exit_nok(627) != 0)
    APIERROR (pop->getNdbError());

  // Label 0
  if (pop->def_label(0) != 0)
    APIERROR (pop->getNdbError());

  Uint32 comparisonValue2= 344;

  // if *ATTR2 == comparisonValue, goto Label 1
  if (pop->branch_col_eq(pattr2Col->getColumnNo(),
                        &comparisonValue2,
                        sizeof(Uint32),
                        false,
                        1) != 0)
    APIERROR (pop->getNdbError());

  // assert(false)
  // Fail the operation with error 628 if we get here
  if (pop->interpret_exit_nok(628) != 0)
    APIERROR (pop->getNdbError());

  // Label 1
  if (pop->def_label(1) != 1)
    APIERROR (pop->getNdbError());

  // Optional infinite loop
  //if (pop->branch_label(0) != 0)
  //  APIERROR (pop->getNdbError());

  // R1 = 10
  if (pop->load_const_u32(R1, 10) != 0)
    APIERROR (pop->getNdbError());

  // R3 = 2
  if (pop->load_const_u32(R3, 2) != 0)
    APIERROR (pop->getNdbError());

  // Now call subroutine 0
  if (pop->call_sub(0) != 0)
    APIERROR (pop->getNdbError());

  // *ATTR2= R2
  if (pop->write_attr("ATTR2", R2) != 0)
    APIERROR (pop->getNdbError());

  // Return ok, we'll move onto an update.
  if (pop->interpret_exit_ok() != 0)
    APIERROR (pop->getNdbError());

  /* Define a final read of the columns after the update */
  recAttrAttr11= pop->getValue("ATTR1");
  recAttrAttr12= pop->getValue("ATTR2");
  recAttrAttr13= pop->getValue("ATTR3");

  // Define any subroutines called by the 'main' program
  // Subroutine 0
  if (pop->def_subroutine(0) != 0)
    APIERROR (pop->getNdbError());

  // R4= 1
  if (pop->load_const_u32(R4, 1) != 0)
    APIERROR (pop->getNdbError());

  // Label 2
  if (pop->def_label(2) != 2)
```

```
      APIERROR (pop->getNdbError());

    // R3= R3-R4
    if (pop->sub_reg(R3, R4, R3) != 0)
      APIERROR (pop->getNdbError());

    // R2= R2 + R1
    if (pop->add_reg(R2, R1, R2) != 0)
      APIERROR (pop->getNdbError());

    // Optional infinite loop
    // if (pop->branch_label(2) != 0)
    //  APIERROR (pop->getNdbError());

    // Loop, subtracting 1 from R4 until R4 < 1
    if (pop->branch_ge(R4, R3, 2) != 0)
      APIERROR (pop->getNdbError());

    // Jump to label 3
    if (pop->branch_label(3) != 0)
      APIERROR (pop->getNdbError());

    // assert(false)
    // Fail operation with error 629
    if (pop->interpret_exit_nok(629) != 0)
      APIERROR (pop->getNdbError());

    // Label 3
    if (pop->def_label(3) != 3)
      APIERROR (pop->getNdbError());

    // Nested subroutine call to sub 2
    if (pop->call_sub(2) != 0)
      APIERROR (pop->getNdbError());

    // Return from subroutine 0
    if (pop->ret_sub() !=0)
      APIERROR (pop->getNdbError());

    // Subroutine 1
    if (pop->def_subroutine(1) != 1)
      APIERROR (pop->getNdbError());

    // R6= R1+R2
    if (pop->add_reg(R1, R2, R6) != 0)
      APIERROR (pop->getNdbError());

    // Return from subrouine 1
    if (pop->ret_sub() !=0)
      APIERROR (pop->getNdbError());

    // Subroutine 2
    if (pop->def_subroutine(2) != 2)
      APIERROR (pop->getNdbError());

    // Call backward to subroutine 1
    if (pop->call_sub(1) != 0)
      APIERROR (pop->getNdbError());

    // Return from subroutine 2
    if (pop->ret_sub() !=0)
      APIERROR (pop->getNdbError());

    break;
  }
  case api_record :
  {
    const NdbOperation *pop;
    rowData.attr1= 4;

    /* NdbRecord does not support an updateTuple pre-read or post-read, so
     * we use separate operations for these.
```

```
      * Note that this assumes that a operations are executed in
      * the order they are defined by NDBAPI, which is not guaranteed.  To
      * ensure execution order, the application should perform a NoCommit
      * execute between operations.
      */

     const NdbOperation *op0= myTransaction->readTuple(pkeyColumnRecord,
                                                       (char*) &rowData,
                                                       pallColsRecord,
                                                       (char*) &rowData);
     if (op0 == NULL)
       APIERROR (myTransaction->getNdbError());

     /* Allocate some space to define an Interpreted program */
     const Uint32 numWords= 64;
     Uint32 space[numWords];

     NdbInterpretedCode stackCode(myTable,
                                  &space[0],
                                  numWords);

     NdbInterpretedCode *code= &stackCode;

     /* Similar program as above, with tortuous control flow and little
      * purpose.  Note that for NdbInterpretedCode, some instruction
      * arguments are in different orders
      */

     // R5= 3
     if (code->load_const_u32(R5, 3) != 0)
       APIERROR(code->getNdbError());

     // R1= *ATTR1; R2= *ATTR2; R3= *ATTR3
     if (code->read_attr(R1, pattr1Col) != 0)
       APIERROR (code->getNdbError());
     if (code->read_attr(R2, pattr2Col) != 0)
       APIERROR (code->getNdbError());
     if (code->read_attr(R3, pattr3Col) != 0)
       APIERROR (code->getNdbError());

     // R3= R3-R5
     if (code->sub_reg(R3, R3, R5) != 0)
       APIERROR (code->getNdbError());

     // R2= R1+R2
     if (code->add_reg(R2, R1, R2) != 0)
       APIERROR (code->getNdbError());

     // *ATTR2= R2
     if (code->write_attr(pattr2Col, R2) != 0)
       APIERROR (code->getNdbError());

     // *ATTR3= R3
     if (code->write_attr(pattr3Col, R3) != 0)
       APIERROR (code->getNdbError());

     // *ATTR3 = *ATTR3 - 30
     if (code->sub_val(pattr3Col->getColumnNo(), (Uint32)30) != 0)
       APIERROR (code->getNdbError());

     Uint32 comparisonValue= 10;

     // if comparisonValue < *ATTR3, goto Label 0
     if (code->branch_col_lt(&comparisonValue,
                             sizeof(Uint32),
                             pattr3Col->getColumnNo(),
                             0) != 0)
       APIERROR (code->getNdbError());

     // assert(false)
     // Fail operation with error 627
     if (code->interpret_exit_nok(627) != 0)
```

```
      APIERROR (code->getNdbError());

    // Label 0
    if (code->def_label(0) != 0)
      APIERROR (code->getNdbError());

    Uint32 comparisonValue2= 344;

    // if *ATTR2 == comparisonValue, goto Label 1
    if (code->branch_col_eq(&comparisonValue2,
                            sizeof(Uint32),
                            pattr2Col->getColumnNo(),
                            1) != 0)
      APIERROR (code->getNdbError());

    // assert(false)
    // Fail operation with error 628
    if (code->interpret_exit_nok(628) != 0)
      APIERROR (code->getNdbError());

    // Label 1
    if (code->def_label(1) != 0)
      APIERROR (code->getNdbError());

    // R1= 10
    if (code->load_const_u32(R1, 10) != 0)
      APIERROR (code->getNdbError());

    // R3= 2
    if (code->load_const_u32(R3, 2) != 0)
      APIERROR (code->getNdbError());

    // Call subroutine 0 to effect
    // R2 = R2 + (R1*R3)
    if (code->call_sub(0) != 0)
      APIERROR (code->getNdbError());

    // *ATTR2= R2
    if (code->write_attr(pattr2Col, R2) != 0)
      APIERROR (code->getNdbError());

    // Return ok
    if (code->interpret_exit_ok() != 0)
      APIERROR (code->getNdbError());

    // Subroutine 0
    if (code->def_sub(0) != 0)
      APIERROR (code->getNdbError());

    // R4= 1
    if (code->load_const_u32(R4, 1) != 0)
      APIERROR (code->getNdbError());

    // Label 2
    if (code->def_label(2) != 0)
      APIERROR (code->getNdbError());

    // R3= R3-R4
    if (code->sub_reg(R3, R3, R4) != 0)
      APIERROR (code->getNdbError());

    // R2= R2+R1
    if (code->add_reg(R2, R2, R1) != 0)
      APIERROR (code->getNdbError());

    // Loop, subtracting 1 from R4 until R4>1
    if (code->branch_ge(R3, R4, 2) != 0)
      APIERROR (code->getNdbError());

    // Jump to label 3
    if (code->branch_label(3) != 0)
      APIERROR (code->getNdbError());
```

```
// Fail operation with error 629
if (code->interpret_exit_nok(629) != 0)
  APIERROR (code->getNdbError());

// Label 3
if (code->def_label(3) != 0)
  APIERROR (code->getNdbError());

// Call sub 2
if (code->call_sub(2) != 0)
  APIERROR (code->getNdbError());

// Return from sub 0
if (code->ret_sub() != 0)
  APIERROR (code->getNdbError());

// Subroutine 1
if (code->def_sub(1) != 0)
  APIERROR (code->getNdbError());

// R6= R1+R2
if (code->add_reg(R6, R1, R2) != 0)
  APIERROR (code->getNdbError());

// Return from subroutine 1
if (code->ret_sub() !=0)
  APIERROR (code->getNdbError());

// Subroutine 2
if (code->def_sub(2) != 0)
  APIERROR (code->getNdbError());

// Call backward to subroutine 1
if (code->call_sub(1) != 0)
  APIERROR (code->getNdbError());

// Return from subroutine 2
if (code->ret_sub() !=0)
  APIERROR (code->getNdbError());

/* Finalise code object
 * This step is essential for NdbInterpretedCode objects
 * and must be done before they can be used.
 */
if (code->finalise() !=0)
  APIERROR (code->getNdbError());

/* Time to define the update operation to use the
 * InterpretedCode object.  The same finalised object
 * could be used with multiple operations or even
 * multiple threads
 */
NdbOperation::OperationOptions oo;
oo.optionsPresent=
  NdbOperation::OperationOptions::OO_INTERPRETED;
oo.interpretedCode= code;

unsigned char mask= 0;

pop= myTransaction->updateTuple(pkeyColumnRecord,
                                (char*) &rowData,
                                pallColsRecord,
                                (char*) &rowData,
                                (const unsigned char *) &mask,
                                          // mask - update nothing
                                &oo,
                                sizeof(NdbOperation::OperationOptions));
if (pop == NULL)
  APIERROR (myTransaction->getNdbError());

// NoCommit execute so we can read the 'after' data.
```

```
    if (myTransaction->execute( NdbTransaction::NoCommit ) != 0)
      APIERROR(myTransaction->getNdbError());

    /* Second read op as we can't currently do a 'read after
     * 'interpreted code' read as part of NdbRecord.
     * We are assuming that the order of op definition == order
     * of execution on a single row, which is not guaranteed.
     */
    const NdbOperation *pop2=
      myTransaction->readTuple(pkeyColumnRecord,
                               (char*) &rowData,
                               pallColsRecord,
                               (char*) &rowData2);
    if (pop2 == NULL)
      APIERROR (myTransaction->getNdbError());

    break;
  }
default :
  {
    std::cout << "Bad branch : " << accessType << "\n";
    exit(-1);
  }
}

if(myTransaction->execute( NdbTransaction::NoCommit ) != 0)
  APIERROR(myTransaction->getNdbError());

// Check return code
if (myTransaction->getNdbError().status != NdbError::Success)
  APIERROR(myTransaction->getNdbError());

switch (accessType)
{
  case api_attr :
  {
    printf(" %2d    %2d    %2d  Before\n"
           " %2d    %2d    %2d  After\n",
           recAttrAttr1->u_32_value(),
           recAttrAttr2->u_32_value(),
           recAttrAttr3->u_32_value(),
           recAttrAttr11->u_32_value(),
           recAttrAttr12->u_32_value(),
           recAttrAttr13->u_32_value());
    break;
  }

  case api_record :
  {
    printf(" %2d    %2d    %2d  Before\n"
           " %2d    %2d    %2d  After\n",
           rowData.attr1,
           rowData.attr2,
           rowData.attr3,
           rowData2.attr1,
           rowData2.attr2,
           rowData2.attr3);
    break;
  }
  default :
  {
    std::cout << "Bad branch : " << accessType << "\n";
    exit(-1);
  }
}

if(myTransaction->execute( NdbTransaction::Commit ) !=0)
  APIERROR(myTransaction->getNdbError());

myNdb.closeTransaction(myTransaction);

std::cout << "-------\n";
```

```
}


/*****************************************************
 * Read and print selected rows with interpreted code *
 *****************************************************/
static void do_interpreted_scan(Ndb &myNdb, ApiType accessType)
{
  NdbDictionary::Dictionary* myDict= myNdb.getDictionary();
  const NdbDictionary::Table *myTable=
                             myDict->getTable("api_recattr_vs_record");

  std::cout << "Running do_interpreted_scan\n";

  if (myTable == NULL)
    APIERROR(myDict->getNdbError());

  std::cout << "ATTR1 ATTR2 ATTR3" << std::endl;

  NdbTransaction *myTransaction=myNdb.startTransaction();
  if (myTransaction == NULL) APIERROR(myNdb.getNdbError());

  NdbScanOperation *psop;
  NdbRecAttr *recAttrAttr1;
  NdbRecAttr *recAttrAttr2;
  NdbRecAttr *recAttrAttr3;

  /* Create some space on the stack for the program */
  const Uint32 numWords= 64;
  Uint32 space[numWords];

  NdbInterpretedCode stackCode(myTable,
                               &space[0],
                               numWords);

  NdbInterpretedCode *code= &stackCode;

  /* RecAttr and NdbRecord scans both use NdbInterpretedCode
   * Let's define a small scan filter of sorts
   */
  Uint32 comparisonValue= 10;

  // Return rows where 10 > ATTR3 (ATTR3 <10)
  if (code->branch_col_gt(&comparisonValue,
                          sizeof(Uint32),
                          pattr3Col->getColumnNo(),
                          0) != 0)
    APIERROR (myTransaction->getNdbError());

  /* If we get here then we don't return this row */
  if (code->interpret_exit_nok() != 0)
    APIERROR (myTransaction->getNdbError());

  /* Label 0 */
  if (code->def_label(0) != 0)
    APIERROR (myTransaction->getNdbError());

  /* Return this row */
  if (code->interpret_exit_ok() != 0)
    APIERROR (myTransaction->getNdbError());

  /* Finalise the Interpreted Program */
  if (code->finalise() != 0)
    APIERROR (myTransaction->getNdbError());

  switch (accessType)
  {
    case api_attr :
    {
      psop=myTransaction->getNdbScanOperation(myTable);

      if (psop == NULL)
```

```
      APIERROR(myTransaction->getNdbError());

    if (psop->readTuples(NdbOperation::LM_Read) != 0)
      APIERROR (myTransaction->getNdbError());

    if (psop->setInterpretedCode(code) != 0)
      APIERROR (myTransaction->getNdbError());

    recAttrAttr1=psop->getValue("ATTR1");
    recAttrAttr2=psop->getValue("ATTR2");
    recAttrAttr3=psop->getValue("ATTR3");

    break;
  }
  case api_record :
  {
    NdbScanOperation::ScanOptions so;

    so.optionsPresent = NdbScanOperation::ScanOptions::SO_INTERPRETED;
    so.interpretedCode= code;

    psop=myTransaction->scanTable(pallColsRecord,
                                  NdbOperation::LM_Read,
                                  NULL, // mask
                                  &so,
                                  sizeof(NdbScanOperation::ScanOptions));

    if (psop == NULL) APIERROR(myTransaction->getNdbError());

    break;
  }
default :
  {
    std::cout << "Bad branch : " << accessType << "\n";
    exit(-1);
  }
}

if(myTransaction->execute( NdbTransaction::NoCommit ) != 0)
  APIERROR(myTransaction->getNdbError());

switch (accessType)
{
  case api_attr :
  {
    while (psop->nextResult(true) == 0)
    {
      printf(" %2d    %2d    %2d\n",
             recAttrAttr1->u_32_value(),
             recAttrAttr2->u_32_value(),
             recAttrAttr3->u_32_value());
    }

    psop->close();

    break;
  }
  case api_record :
  {
    RowData *prowData; // Ptr to point to our data

    int rc=0;

    while ((rc = psop->nextResult((const char**) &prowData,
                                  true,
                                  false)) == GOT_ROW)
    {
      printf(" %2d    %2d    %2d\n",
             prowData->attr1,
             prowData->attr2,
             prowData->attr3);
    }
```

```
      if (rc != NO_MORE_ROWS)  APIERROR(myTransaction->getNdbError());

      psop->close(true);

      break;
    }
    default :
    {
      std::cout << "Bad branch : " << accessType << "\n";
      exit(-1);
    }
  }

  if(myTransaction->execute( NdbTransaction::Commit ) !=0)
    APIERROR(myTransaction->getNdbError());

  myNdb.closeTransaction(myTransaction);

  std::cout << "-------\n";
}

/*****************************************************
 * Read some data using the default NdbRecord objects *
 *****************************************************/
static void do_read_using_default(Ndb &myNdb)
{
  NdbDictionary::Dictionary* myDict= myNdb.getDictionary();

  const NdbDictionary::Table *myTable=
                             myDict->getTable("api_recattr_vs_record");

  const NdbRecord* tableRec= myTable->getDefaultRecord();

  if (myTable == NULL)
    APIERROR(myDict->getNdbError());

  std::cout << "Running do_read_using_default_record (NdbRecord only)\n";
  std::cout << "ATTR1 ATTR2 ATTR3" << std::endl;

  /* Allocate some space for the rows to be read into */
  char* buffer= (char*)malloc(NdbDictionary::getRecordRowLength(tableRec));

  if (buffer== NULL)
  {
    printf("Allocation failed\n");
    exit(-1);
  }

  for (int i = 0; i < 10; i++) {
    NdbTransaction *myTransaction= myNdb.startTransaction();
    if (myTransaction == NULL) APIERROR(myNdb.getNdbError());

    char* attr1= NdbDictionary::getValuePtr(tableRec,
                                            buffer,
                                            attr1ColNum);
    *((unsigned int*)attr1)= i;

    const NdbOperation *pop=
      myTransaction->readTuple(tableRec,
                               buffer,
                               tableRec, // Read everything
                               buffer);
    if (pop==NULL) APIERROR(myTransaction->getNdbError());

    if(myTransaction->execute( NdbTransaction::Commit ) == -1)
      APIERROR(myTransaction->getNdbError());

    NdbError err= myTransaction->getNdbError();
    if (err.code != 0)
    {
      if (err.classification == NdbError::NoDataFound)
```

```
        std::cout << "Tuple " << i
                  << " does not exist." << std::endl;
      else
        APIERROR(myTransaction->getNdbError());
    }
    else
    {
      printf(" %2d    %2d    %2d\n",
             i,
             *((unsigned int*) NdbDictionary::getValuePtr(tableRec,
                                                          buffer,
                                                          attr2ColNum)),
             *((unsigned int*) NdbDictionary::getValuePtr(tableRec,
                                                          buffer,
                                                          attr3ColNum)));
    }

    myNdb.closeTransaction(myTransaction);
  }

  free(buffer);

  std::cout << "-------\n";
}
```

2.5.8 NDB API Event Handling Example

This example demonstrates NDB API event handling.

The source code for this program may be found in the NDB Cluster source tree, in the file `storage/ndb/ndbapi-examples/ndbapi_event/ndbapi_event.cpp`.

```
#include <NdbApi.hpp>

// Used for cout
#include <stdio.h>
#include <iostream>
#include <unistd.h>
#ifdef VM_TRACE
#include <my_global.h>
#endif
#ifndef assert
#include <assert.h>
#endif


/**
 * Assume that there is a table which is being updated by
 * another process (e.g. flexBench -l 0 -stdtables).
 * We want to monitor what happens with column values.
 *
 * Or using the mysql client:
 *
 * shell> mysql -u root
 * mysql> create database ndb_examples;
 * mysql> use ndb_examples;
 * mysql> create table t0
          (c0 int, c1 int, c2 char(4), c3 char(4), c4 text,
          primary key(c0, c2)) engine ndb charset latin1;
 *
 * In another window start ndbapi_event, wait until properly started

   insert into t0 values (1, 2, 'a', 'b', null);
   insert into t0 values (3, 4, 'c', 'd', null);
   update t0 set c3 = 'e' where c0 = 1 and c2 = 'a'; -- use pk
   update t0 set c3 = 'f'; -- use scan
   update t0 set c3 = 'F'; -- use scan update to 'same'
   update t0 set c2 = 'g' where c0 = 1; -- update pk part
   update t0 set c2 = 'G' where c0 = 1; -- update pk part to 'same'
   update t0 set c0 = 5, c2 = 'H' where c0 = 3; -- update full PK
```

```
   delete from t0;

   insert ...; update ...; -- see events w/ same pk merged (if -m option)
   delete ...; insert ...; -- there are 5 combinations ID IU DI UD UU
   update ...; update ...;

   -- text requires -m flag
   set @a = repeat('a',256); -- inline size
   set @b = repeat('b',2000); -- part size
   set @c = repeat('c',2000*30); -- 30 parts

   -- update the text field using combinations of @a, @b, @c ...

 * you should see the data popping up in the example window
 *
 */

#define APIERROR(error) \
  { std::cout << "Error in " << __FILE__ << ", line:" << __LINE__ << ", code:" \
              << error.code << ", msg: " << error.message << "." << std::endl; \
    exit(-1); }

int myCreateEvent(Ndb* myNdb,
    const char *eventName,
    const char *eventTableName,
    const char **eventColumnName,
    const int noEventColumnName,
                bool merge_events);

int main(int argc, char** argv)
{
  if (argc < 3)
  {
    std::cout << "Arguments are <connect_string cluster> <timeout> [m(merge events)|d(debug)].\n";
    exit(-1);
  }
  const char *connection_string = argv[1];
  int timeout = atoi(argv[2]);
  ndb_init();
  bool merge_events = argc > 3 && strchr(argv[3], 'm') != 0;
#ifdef VM_TRACE
  bool dbug = argc > 3 && strchr(argv[3], 'd') != 0;
  if (dbug) DBUG_PUSH("d:t:");
  if (dbug) putenv("API_SIGNAL_LOG=-");
#endif

  Ndb_cluster_connection *cluster_connection=
    new Ndb_cluster_connection(connection_string); // Object representing the cluster

  int r= cluster_connection->connect(5 /* retries               */,
        3 /* delay between retries */,
        1 /* verbose               */);
  if (r > 0)
  {
    std::cout
      << "Cluster connect failed, possibly resolved with more retries.\n";
    exit(-1);
  }
  else if (r < 0)
  {
    std::cout
      << "Cluster connect failed.\n";
    exit(-1);
  }

  if (cluster_connection->wait_until_ready(30,30))
  {
    std::cout << "Cluster was not ready within 30 secs." << std::endl;
    exit(-1);
  }

  Ndb* myNdb= new Ndb(cluster_connection,
```

```
       "ndb_examples");  // Object representing the database

 if (myNdb->init() == -1) APIERROR(myNdb->getNdbError());

 const char *eventName= "CHNG_IN_t0";
 const char *eventTableName= "t0";
 const int noEventColumnName= 5;
 const char *eventColumnName[noEventColumnName]=
   {"c0",
    "c1",
    "c2",
    "c3",
    "c4"
   };

 // Create events
 myCreateEvent(myNdb,
 eventName,
 eventTableName,
 eventColumnName,
 noEventColumnName,
              merge_events);

 // Normal values and blobs are unfortunately handled differently..
 typedef union { NdbRecAttr* ra; NdbBlob* bh; } RA_BH;

 int i, j, k, l;
 j = 0;
 while (j < timeout) {

   // Start "transaction" for handling events
   NdbEventOperation* op;
   printf("create EventOperation\n");
   if ((op = myNdb->createEventOperation(eventName)) == NULL)
     APIERROR(myNdb->getNdbError());
   op->mergeEvents(merge_events);

   printf("get values\n");
   RA_BH recAttr[noEventColumnName];
   RA_BH recAttrPre[noEventColumnName];
   // primary keys should always be a part of the result
   for (i = 0; i < noEventColumnName; i++) {
     if (i < 4) {
       recAttr[i].ra    = op->getValue(eventColumnName[i]);
       recAttrPre[i].ra = op->getPreValue(eventColumnName[i]);
     } else if (merge_events) {
       recAttr[i].bh    = op->getBlobHandle(eventColumnName[i]);
       recAttrPre[i].bh = op->getPreBlobHandle(eventColumnName[i]);
     }
   }

   // set up the callbacks
   printf("execute\n");
   // This starts changes to "start flowing"
   if (op->execute())
     APIERROR(op->getNdbError());

   NdbEventOperation* the_op = op;

   i= 0;
   while (i < timeout) {
     // printf("now waiting for event...\n");
     int r = myNdb->pollEvents(1000); // wait for event or 1000 ms
     if (r > 0) {
// printf("got data! %d\n", r);
while ((op= myNdb->nextEvent())) {
         assert(the_op == op);
  i++;
  switch (op->getEventType()) {
  case NdbDictionary::Event::TE_INSERT:
    printf("%u INSERT", i);
    break;
```

```
  case NdbDictionary::Event::TE_DELETE:
    printf("%u DELETE", i);
    break;
  case NdbDictionary::Event::TE_UPDATE:
    printf("%u UPDATE", i);
    break;
  default:
    abort(); // should not happen
  }
        printf(" gci=%d\n", (int)op->getGCI());
        for (k = 0; k <= 1; k++) {
          printf(k == 0 ? "post: " : "pre : ");
          for (l = 0; l < noEventColumnName; l++) {
            if (l < 4) {
              NdbRecAttr* ra = k == 0 ? recAttr[l].ra : recAttrPre[l].ra;
              if (ra->isNULL() >= 0) { // we have a value
                if (ra->isNULL() == 0) { // we have a non-null value
                  if (l < 2)
                    printf("%-5u", ra->u_32_value());
                  else
                    printf("%-5.4s", ra->aRef());
                } else
                  printf("%-5s", "NULL");
              } else
                printf("%-5s", "-"); // no value
            } else if (merge_events) {
              int isNull;
              NdbBlob* bh = k == 0 ? recAttr[l].bh : recAttrPre[l].bh;
              bh->getDefined(isNull);
              if (isNull >= 0) { // we have a value
                if (! isNull) { // we have a non-null value
                  Uint64 length = 0;
                  bh->getLength(length);
                  // read into buffer
                  unsigned char* buf = new unsigned char [length];
                  memset(buf, 'X', length);
                  Uint32 n = length;
                  bh->readData(buf, n); // n is in/out
                  assert(n == length);
                  // pretty-print
                  bool first = true;
                  Uint32 i = 0;
                  while (i < n) {
                    unsigned char c = buf[i++];
                    Uint32 m = 1;
                    while (i < n && buf[i] == c)
                      i++, m++;
                    if (! first)
                      printf("+");
                    printf("%u%c", m, c);
                    first = false;
                  }
                  printf("[%u]", n);
                  delete [] buf;
                } else
                  printf("%-5s", "NULL");
              } else
                printf("%-5s", "-"); // no value
            }
          }
          printf("\n");
        }
}
     } // else printf("timed out (%i)\n", timeout);
   }
   // don't want to listen to events anymore
   if (myNdb->dropEventOperation(the_op)) APIERROR(myNdb->getNdbError());
   the_op = 0;

   j++;
 }
```

```
  {
    NdbDictionary::Dictionary *myDict = myNdb->getDictionary();
    if (!myDict) APIERROR(myNdb->getNdbError());
    // remove event from database
    if (myDict->dropEvent(eventName)) APIERROR(myDict->getNdbError());
  }

  delete myNdb;
  delete cluster_connection;
  ndb_end(0);
  return 0;
}

int myCreateEvent(Ndb* myNdb,
    const char *eventName,
    const char *eventTableName,
    const char **eventColumnNames,
    const int noEventColumnNames,
                bool merge_events)
{
  NdbDictionary::Dictionary *myDict= myNdb->getDictionary();
  if (!myDict) APIERROR(myNdb->getNdbError());

  const NdbDictionary::Table *table= myDict->getTable(eventTableName);
  if (!table) APIERROR(myDict->getNdbError());

  NdbDictionary::Event myEvent(eventName, *table);
  myEvent.addTableEvent(NdbDictionary::Event::TE_ALL);
  //  myEvent.addTableEvent(NdbDictionary::Event::TE_INSERT);
  //  myEvent.addTableEvent(NdbDictionary::Event::TE_UPDATE);
  //  myEvent.addTableEvent(NdbDictionary::Event::TE_DELETE);

  myEvent.addEventColumns(noEventColumnNames, eventColumnNames);
  myEvent.mergeEvents(merge_events);

  // Add event to database
  if (myDict->createEvent(myEvent) == 0)
    myEvent.print();
  else if (myDict->getNdbError().classification ==
    NdbError::SchemaObjectExists) {
    printf("Event creation failed, event exists\n");
    printf("dropping Event...\n");
    if (myDict->dropEvent(eventName)) APIERROR(myDict->getNdbError());
    // try again
    // Add event to database
    if ( myDict->createEvent(myEvent)) APIERROR(myDict->getNdbError());
  } else
    APIERROR(myDict->getNdbError());

  return 0;
}
```

2.5.9 NDB API Example: Basic BLOB Handling

This example illustrates the manipulation of a `BLOB` column in the `NDB` API. It demonstrates how to perform insert, read, and update operations, using both inline value buffers as well as read and write methods.

The source code can be found can be found in the file `storage/ndb/ndbapi-examples/ndbapi_blob/ndbapi_blob.cpp` in the NDB Cluster source tree.

Note

While the MySQL data type used in the example is actually `TEXT`, the same principles apply

```
/*
   Copyright (c) 2007, 2016, Oracle and/or its affiliates. All rights reserved.
```

```
   This program is free software; you can redistribute it and/or modify
   it under the terms of the GNU General Public License as published by
   the Free Software Foundation; version 2 of the License.

   This program is distributed in the hope that it will be useful,
   but WITHOUT ANY WARRANTY; without even the implied warranty of
   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
   GNU General Public License for more details.

   You should have received a copy of the GNU General Public License
   along with this program; if not, write to the Free Software
   Foundation, Inc., 51 Franklin St, Fifth Floor, Boston, MA 02110-1301  USA
*/


/*
  ndbapi_blob.cpp:

  Illustrates the manipulation of BLOB (actually TEXT in this example).

  Shows insert, read, and update, using both inline value buffer and
  read/write methods.
 */

#ifdef _WIN32
#include <winsock2.h>
#endif
#include <mysql.h>
#include <mysqld_error.h>
#include <NdbApi.hpp>
#include <stdlib.h>
#include <string.h>
/* Used for cout. */
#include <iostream>
#include <stdio.h>
#include <ctype.h>


/**
 * Helper debugging macros
 */
#define PRINT_ERROR(code,msg) \
  std::cout << "Error in " << __FILE__ << ", line: " << __LINE__ \
            << ", code: " << code \
            << ", msg: " << msg << "." << std::endl
#define MYSQLERROR(mysql) { \
  PRINT_ERROR(mysql_errno(&mysql),mysql_error(&mysql)); \
  exit(-1); }
#define APIERROR(error) { \
  PRINT_ERROR(error.code,error.message); \
  exit(-1); }

/* Quote taken from Project Gutenberg. */
const char *text_quote=
"Just at this moment, somehow or other, they began to run.\n"
"\n"
"  Alice never could quite make out, in thinking it over\n"
"afterwards, how it was that they began:  all she remembers is,\n"
"that they were running hand in hand, and the Queen went so fast\n"
"that it was all she could do to keep up with her:  and still the\n"
"Queen kept crying 'Faster! Faster!' but Alice felt she COULD NOT\n"
"go faster, though she had not breath left to say so.\n"
"\n"
"  The most curious part of the thing was, that the trees and the\n"
"other things round them never changed their places at all:\n"
"however fast they went, they never seemed to pass anything.  'I\n"
"wonder if all the things move along with us?' thought poor\n"
"puzzled Alice.  And the Queen seemed to guess her thoughts, for\n"
"she cried, 'Faster!  Don't try to talk!'\n"
"\n"
"  Not that Alice had any idea of doing THAT.  She felt as if she\n"
"would never be able to talk again, she was getting so much out of\n"
```

```
"breath:  and still the Queen cried 'Faster! Faster!' and dragged\n"
"her along.  'Are we nearly there?'  Alice managed to pant out at\n"
"last.\n"
"\n"
"  'Nearly there!' the Queen repeated.  'Why, we passed it ten\n"
"minutes ago!  Faster!'  And they ran on for a time in silence,\n"
"with the wind whistling in Alice's ears, and almost blowing her\n"
"hair off her head, she fancied.\n"
"\n"
"  'Now!  Now!' cried the Queen.  'Faster!  Faster!'  And they\n"
"went so fast that at last they seemed to skim through the air,\n"
"hardly touching the ground with their feet, till suddenly, just\n"
"as Alice was getting quite exhausted, they stopped, and she found\n"
"herself sitting on the ground, breathless and giddy.\n"
"\n"
"  The Queen propped her up against a tree, and said kindly, 'You\n"
"may rest a little now.'\n"
"\n"
"  Alice looked round her in great surprise.  'Why, I do believe\n"
"we've been under this tree the whole time!  Everything's just as\n"
"it was!'\n"
"\n"
"  'Of course it is,' said the Queen, 'what would you have it?'\n"
"\n"
"  'Well, in OUR country,' said Alice, still panting a little,\n"
"'you'd generally get to somewhere else--if you ran very fast\n"
"for a long time, as we've been doing.'\n"
"\n"
"  'A slow sort of country!' said the Queen.  'Now, HERE, you see,\n"
"it takes all the running YOU can do, to keep in the same place.\n"
"If you want to get somewhere else, you must run at least twice as\n"
"fast as that!'\n"
"\n"
"  'I'd rather not try, please!' said Alice.  'I'm quite content\n"
"to stay here--only I AM so hot and thirsty!'\n"
"\n"
" -- Lewis Carroll, 'Through the Looking-Glass'.";

/*
  Function to drop table.
*/
void drop_table(MYSQL &mysql)
{
  if (mysql_query(&mysql, "DROP TABLE api_blob"))
    MYSQLERROR(mysql);
}


/*
  Functions to create table.
*/
int try_create_table(MYSQL &mysql)
{
  return mysql_query(&mysql,
                     "CREATE TABLE"
                     "  api_blob"
                     "    (my_id INT UNSIGNED NOT NULL,"
                     "     my_text TEXT NOT NULL,"
                     "     PRIMARY KEY USING HASH (my_id))"
                     "  ENGINE=NDB");
}

void create_table(MYSQL &mysql)
{
  if (try_create_table(mysql))
  {
    if (mysql_errno(&mysql) != ER_TABLE_EXISTS_ERROR)
      MYSQLERROR(mysql);
    std::cout << "NDB Cluster already has example table: api_blob. "
              << "Dropping it..." << std::endl;
    /*****************
     * Recreate table *
```

```
    ******************/
    drop_table(mysql);
    if (try_create_table(mysql))
      MYSQLERROR(mysql);
  }
}

int populate(Ndb *myNdb)
{
  const NdbDictionary::Dictionary *myDict= myNdb->getDictionary();
  const NdbDictionary::Table *myTable= myDict->getTable("api_blob");
  if (myTable == NULL)
    APIERROR(myDict->getNdbError());

  NdbTransaction *myTrans= myNdb->startTransaction();
  if (myTrans == NULL)
    APIERROR(myNdb->getNdbError());

  NdbOperation *myNdbOperation= myTrans->getNdbOperation(myTable);
  if (myNdbOperation == NULL)
    APIERROR(myTrans->getNdbError());
  myNdbOperation->insertTuple();
  myNdbOperation->equal("my_id", 1);
  NdbBlob *myBlobHandle= myNdbOperation->getBlobHandle("my_text");
  if (myBlobHandle == NULL)
    APIERROR(myNdbOperation->getNdbError());
  myBlobHandle->setValue(text_quote, strlen(text_quote));

  int check= myTrans->execute(NdbTransaction::Commit);
  myTrans->close();
  return check != -1;
}


int update_key(Ndb *myNdb)
{
  /*
    Uppercase all characters in TEXT field, using primary key operation.
    Use piece-wise read/write to avoid loading entire data into memory
    at once.
  */
  const NdbDictionary::Dictionary *myDict= myNdb->getDictionary();
  const NdbDictionary::Table *myTable= myDict->getTable("api_blob");
  if (myTable == NULL)
    APIERROR(myDict->getNdbError());

  NdbTransaction *myTrans= myNdb->startTransaction();
  if (myTrans == NULL)
    APIERROR(myNdb->getNdbError());

  NdbOperation *myNdbOperation= myTrans->getNdbOperation(myTable);
  if (myNdbOperation == NULL)
    APIERROR(myTrans->getNdbError());
  myNdbOperation->updateTuple();
  myNdbOperation->equal("my_id", 1);
  NdbBlob *myBlobHandle= myNdbOperation->getBlobHandle("my_text");
  if (myBlobHandle == NULL)
    APIERROR(myNdbOperation->getNdbError());

  /* Execute NoCommit to make the blob handle active. */
  if (-1 == myTrans->execute(NdbTransaction::NoCommit))
    APIERROR(myTrans->getNdbError());

  Uint64 length= 0;
  if (-1 == myBlobHandle->getLength(length))
    APIERROR(myBlobHandle->getNdbError());

  /*
    A real application should use a much larger chunk size for
    efficiency, preferably much larger than the part size, which
    defaults to 2000. 64000 might be a good value.
  */
```

```
#define CHUNK_SIZE 100
  int chunk;
  char buffer[CHUNK_SIZE];
  for (chunk= (length-1)/CHUNK_SIZE; chunk >=0; chunk--)
  {
    Uint64 pos= chunk*CHUNK_SIZE;
    Uint32 chunk_length= CHUNK_SIZE;
    if (pos + chunk_length > length)
      chunk_length= length - pos;

    /* Read from the end back, to illustrate seeking. */
    if (-1 == myBlobHandle->setPos(pos))
      APIERROR(myBlobHandle->getNdbError());
    if (-1 == myBlobHandle->readData(buffer, chunk_length))
      APIERROR(myBlobHandle->getNdbError());
    int res= myTrans->execute(NdbTransaction::NoCommit);
    if (-1 == res)
      APIERROR(myTrans->getNdbError());

    /* Uppercase everything. */
    for (Uint64 j= 0; j < chunk_length; j++)
      buffer[j]= toupper(buffer[j]);

    if (-1 == myBlobHandle->setPos(pos))
      APIERROR(myBlobHandle->getNdbError());
    if (-1 == myBlobHandle->writeData(buffer, chunk_length))
      APIERROR(myBlobHandle->getNdbError());
    /* Commit on the final update. */
    if (-1 == myTrans->execute(chunk ?
                               NdbTransaction::NoCommit :
                               NdbTransaction::Commit))
      APIERROR(myTrans->getNdbError());
  }

  myNdb->closeTransaction(myTrans);

  return 1;
}


int update_scan(Ndb *myNdb)
{
  /*
    Lowercase all characters in TEXT field, using a scan with
    updateCurrentTuple().
  */
  char buffer[10000];

  const NdbDictionary::Dictionary *myDict= myNdb->getDictionary();
  const NdbDictionary::Table *myTable= myDict->getTable("api_blob");
  if (myTable == NULL)
    APIERROR(myDict->getNdbError());

  NdbTransaction *myTrans= myNdb->startTransaction();
  if (myTrans == NULL)
    APIERROR(myNdb->getNdbError());

  NdbScanOperation *myScanOp= myTrans->getNdbScanOperation(myTable);
  if (myScanOp == NULL)
    APIERROR(myTrans->getNdbError());
  myScanOp->readTuples(NdbOperation::LM_Exclusive);
  NdbBlob *myBlobHandle= myScanOp->getBlobHandle("my_text");
  if (myBlobHandle == NULL)
    APIERROR(myScanOp->getNdbError());
  if (myBlobHandle->getValue(buffer, sizeof(buffer)))
    APIERROR(myBlobHandle->getNdbError());

  /* Start the scan. */
  if (-1 == myTrans->execute(NdbTransaction::NoCommit))
    APIERROR(myTrans->getNdbError());

  int res;
```

```
  for (;;)
  {
    res= myScanOp->nextResult(true);
    if (res==1)
      break;                                    // Scan done.
    else if (res)
      APIERROR(myScanOp->getNdbError());

    Uint64 length= 0;
    if (myBlobHandle->getLength(length) == -1)
      APIERROR(myBlobHandle->getNdbError());

    /* Lowercase everything. */
    for (Uint64 j= 0; j < length; j++)
      buffer[j]= tolower(buffer[j]);

    NdbOperation *myUpdateOp= myScanOp->updateCurrentTuple();
    if (myUpdateOp == NULL)
      APIERROR(myTrans->getNdbError());
    NdbBlob *myBlobHandle2= myUpdateOp->getBlobHandle("my_text");
    if (myBlobHandle2 == NULL)
      APIERROR(myUpdateOp->getNdbError());
    if (myBlobHandle2->setValue(buffer, length))
      APIERROR(myBlobHandle2->getNdbError());

    if (-1 == myTrans->execute(NdbTransaction::NoCommit))
      APIERROR(myTrans->getNdbError());
  }

  if (-1 == myTrans->execute(NdbTransaction::Commit))
    APIERROR(myTrans->getNdbError());

  myNdb->closeTransaction(myTrans);

  return 1;
}


struct ActiveHookData {
  char buffer[10000];
  Uint32 readLength;
};

int myFetchHook(NdbBlob* myBlobHandle, void* arg)
{
  ActiveHookData *ahd= (ActiveHookData *)arg;

  ahd->readLength= sizeof(ahd->buffer) - 1;
  return myBlobHandle->readData(ahd->buffer, ahd->readLength);
}

int fetch_key(Ndb *myNdb)
{
  /*
    Fetch and show the blob field, using setActiveHook().
  */
  const NdbDictionary::Dictionary *myDict= myNdb->getDictionary();
  const NdbDictionary::Table *myTable= myDict->getTable("api_blob");
  if (myTable == NULL)
    APIERROR(myDict->getNdbError());

  NdbTransaction *myTrans= myNdb->startTransaction();
  if (myTrans == NULL)
    APIERROR(myNdb->getNdbError());

  NdbOperation *myNdbOperation= myTrans->getNdbOperation(myTable);
  if (myNdbOperation == NULL)
    APIERROR(myTrans->getNdbError());
  myNdbOperation->readTuple();
  myNdbOperation->equal("my_id", 1);
  NdbBlob *myBlobHandle= myNdbOperation->getBlobHandle("my_text");
  if (myBlobHandle == NULL)
```

```
    APIERROR(myNdbOperation->getNdbError());
  struct ActiveHookData ahd;
  if (myBlobHandle->setActiveHook(myFetchHook, &ahd) == -1)
    APIERROR(myBlobHandle->getNdbError());

  /*
    Execute Commit, but calling our callback set up in setActiveHook()
    before actually committing.
  */
  if (-1 == myTrans->execute(NdbTransaction::Commit))
    APIERROR(myTrans->getNdbError());
  myNdb->closeTransaction(myTrans);

  /* Our fetch callback will have been called during the execute(). */

  ahd.buffer[ahd.readLength]= '\0';
  std::cout << "Fetched data:" << std::endl << ahd.buffer << std::endl;

  return 1;
}


int update2_key(Ndb *myNdb)
{
  char buffer[10000];

  /* Simple setValue() update. */
  const NdbDictionary::Dictionary *myDict= myNdb->getDictionary();
  const NdbDictionary::Table *myTable= myDict->getTable("api_blob");
  if (myTable == NULL)
    APIERROR(myDict->getNdbError());

  NdbTransaction *myTrans= myNdb->startTransaction();
  if (myTrans == NULL)
    APIERROR(myNdb->getNdbError());

  NdbOperation *myNdbOperation= myTrans->getNdbOperation(myTable);
  if (myNdbOperation == NULL)
    APIERROR(myTrans->getNdbError());
  myNdbOperation->updateTuple();
  myNdbOperation->equal("my_id", 1);
  NdbBlob *myBlobHandle= myNdbOperation->getBlobHandle("my_text");
  if (myBlobHandle == NULL)
    APIERROR(myNdbOperation->getNdbError());
  memset(buffer, ' ', sizeof(buffer));
  if (myBlobHandle->setValue(buffer, sizeof(buffer)) == -1)
    APIERROR(myBlobHandle->getNdbError());

  if (-1 == myTrans->execute(NdbTransaction::Commit))
    APIERROR(myTrans->getNdbError());
  myNdb->closeTransaction(myTrans);

  return 1;
}


int delete_key(Ndb *myNdb)
{
  /* Deletion of blob row. */
  const NdbDictionary::Dictionary *myDict= myNdb->getDictionary();
  const NdbDictionary::Table *myTable= myDict->getTable("api_blob");
  if (myTable == NULL)
    APIERROR(myDict->getNdbError());

  NdbTransaction *myTrans= myNdb->startTransaction();
  if (myTrans == NULL)
    APIERROR(myNdb->getNdbError());

  NdbOperation *myNdbOperation= myTrans->getNdbOperation(myTable);
  if (myNdbOperation == NULL)
    APIERROR(myTrans->getNdbError());
  myNdbOperation->deleteTuple();
```

```
  myNdbOperation->equal("my_id", 1);

  if (-1 == myTrans->execute(NdbTransaction::Commit))
    APIERROR(myTrans->getNdbError());
  myNdb->closeTransaction(myTrans);

  return 1;
}

void mysql_connect_and_create(const char *socket)
{
  MYSQL mysql;
  bool ok;

  mysql_init(&mysql);

  ok = mysql_real_connect(&mysql, "localhost", "root", "", "", 0, socket, 0);
  if(ok) {
    mysql_query(&mysql, "CREATE DATABASE ndb_examples");
    ok = ! mysql_select_db(&mysql, "ndb_examples");
  }
  if(ok) {
    create_table(mysql);
  }
  mysql_close(&mysql);

  if(! ok) MYSQLERROR(mysql);
}

void ndb_run_blob_operations(const char *connectstring)
{
  /* Connect to ndb cluster. */
  Ndb_cluster_connection cluster_connection(connectstring);
  if (cluster_connection.connect(4, 5, 1))
  {
    std::cout << "Unable to connect to cluster within 30 secs." << std::endl;
    exit(-1);
  }
  /* Optionally connect and wait for the storage nodes (ndbd's). */
  if (cluster_connection.wait_until_ready(30,0) < 0)
  {
    std::cout << "Cluster was not ready within 30 secs.\n";
    exit(-1);
  }

  Ndb myNdb(&cluster_connection,"ndb_examples");
  if (myNdb.init(1024) == -1) {      // Set max 1024 parallel transactions
    APIERROR(myNdb.getNdbError());
    exit(-1);
  }

  if(populate(&myNdb) > 0)
    std::cout << "populate: Success!" << std::endl;

  if(update_key(&myNdb) > 0)
    std::cout << "update_key: Success!" << std::endl;

  if(update_scan(&myNdb) > 0)
    std::cout << "update_scan: Success!" << std::endl;

  if(fetch_key(&myNdb) > 0)
    std::cout << "fetch_key: Success!" << std::endl;

  if(update2_key(&myNdb) > 0)
    std::cout << "update2_key: Success!" << std::endl;

  if(delete_key(&myNdb) > 0)
    std::cout << "delete_key: Success!" << std::endl;
}

int main(int argc, char**argv)
{
```

```
  if (argc != 3)
  {
    std::cout << "Arguments are <socket mysqld> <connect_string cluster>.\n";
    exit(-1);
  }
  char *mysqld_sock  = argv[1];
  const char *connectstring = argv[2];

  mysql_connect_and_create(mysqld_sock);

  ndb_init();
  ndb_run_blob_operations(connectstring);
  ndb_end(0);

  return 0;
}
```

2.5.10 NDB API Example: Handling BLOB Columns and Values Using NdbRecord

This example illustrates the manipulation of a `BLOB` column in the NDB API using the `NdbRecord` interface. It demonstrates how to perform insert, read, and update operations, using both inline value buffers as well as read and write methods. It can be found in the file `storage/ndb/ndbapi-examples/ndbapi_blob_ndbrecord/main.cpp` in the NDB Cluster source trees.

Note

While the MySQL data type used in the example is actually `TEXT`, the same principles apply

```
/*
   Copyright (c) 2007, 2016, Oracle and/or its affiliates. All rights reserved.

   This program is free software; you can redistribute it and/or modify
   it under the terms of the GNU General Public License as published by
   the Free Software Foundation; version 2 of the License.

   This program is distributed in the hope that it will be useful,
   but WITHOUT ANY WARRANTY; without even the implied warranty of
   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
   GNU General Public License for more details.

   You should have received a copy of the GNU General Public License
   along with this program; if not, write to the Free Software
   Foundation, Inc., 51 Franklin St, Fifth Floor, Boston, MA 02110-1301  USA
*/


/*
  ndbapi_blob_ndbrecord

  Illustrates the manipulation of BLOB (actually TEXT in this example).
  This example uses the NdbRecord style way of accessing tuples.

  Shows insert, read, and update, using both inline value buffer and
  read/write methods.
 */

#ifdef _WIN32
#include <winsock2.h>
#endif
#include <mysql.h>
#include <mysqld_error.h>
#include <NdbApi.hpp>
/* Used for cout. */
#include <iostream>
#include <stdio.h>
#include <ctype.h>
```

```
#include <stdlib.h>
#include <stddef.h>
#include <string.h>

/**
 * Helper debugging macros
 */
#define PRINT_ERROR(code,msg) \
  std::cout << "Error in " << __FILE__ << ", line: " << __LINE__ \
            << ", code: " << code \
            << ", msg: " << msg << "." << std::endl
#define MYSQLERROR(mysql) { \
  PRINT_ERROR(mysql_errno(&mysql),mysql_error(&mysql)); \
  exit(-1); }
#define APIERROR(error) { \
  PRINT_ERROR(error.code,error.message); \
  exit(-1); }

/* Quote taken from Project Gutenberg. */
const char *text_quote=
"Just at this moment, somehow or other, they began to run.\n"
"\n"
"  Alice never could quite make out, in thinking it over\n"
"afterwards, how it was that they began:  all she remembers is,\n"
"that they were running hand in hand, and the Queen went so fast\n"
"that it was all she could do to keep up with her:  and still the\n"
"Queen kept crying 'Faster! Faster!' but Alice felt she COULD NOT\n"
"go faster, though she had not breath left to say so.\n"
"\n"
"  The most curious part of the thing was, that the trees and the\n"
"other things round them never changed their places at all:\n"
"however fast they went, they never seemed to pass anything.  'I\n"
"wonder if all the things move along with us?' thought poor\n"
"puzzled Alice.  And the Queen seemed to guess her thoughts, for\n"
"she cried, 'Faster!  Don't try to talk!'\n"
"\n"
"  Not that Alice had any idea of doing THAT.  She felt as if she\n"
"would never be able to talk again, she was getting so much out of\n"
"breath:  and still the Queen cried 'Faster! Faster!' and dragged\n"
"her along.  'Are we nearly there?'  Alice managed to pant out at\n"
"last.\n"
"\n"
"  'Nearly there!' the Queen repeated.  'Why, we passed it ten\n"
"minutes ago!  Faster!'  And they ran on for a time in silence,\n"
"with the wind whistling in Alice's ears, and almost blowing her\n"
"hair off her head, she fancied.\n"
"\n"
"  'Now!  Now!' cried the Queen.  'Faster!  Faster!'  And they\n"
"went so fast that at last they seemed to skim through the air,\n"
"hardly touching the ground with their feet, till suddenly, just\n"
"as Alice was getting quite exhausted, they stopped, and she found\n"
"herself sitting on the ground, breathless and giddy.\n"
"\n"
"  The Queen propped her up against a tree, and said kindly, 'You\n"
"may rest a little now.'\n"
"\n"
"  Alice looked round her in great surprise.  'Why, I do believe\n"
"we've been under this tree the whole time!  Everything's just as\n"
"it was!'\n"
"\n"
"  'Of course it is,' said the Queen, 'what would you have it?'\n"
"\n"
"  'Well, in OUR country,' said Alice, still panting a little,\n"
"'you'd generally get to somewhere else--if you ran very fast\n"
"for a long time, as we've been doing.'\n"
"\n"
"  'A slow sort of country!' said the Queen.  'Now, HERE, you see,\n"
"it takes all the running YOU can do, to keep in the same place.\n"
"If you want to get somewhere else, you must run at least twice as\n"
"fast as that!'\n"
"\n"
"  'I'd rather not try, please!' said Alice.  'I'm quite content\n"
```

```
"to stay here--only I AM so hot and thirsty!'\n"
"\n"
" -- Lewis Carroll, 'Through the Looking-Glass'.";

/* NdbRecord objects. */

const NdbRecord *key_record;                    // For specifying table key
const NdbRecord *blob_record;                   // For accessing blob
const NdbRecord *full_record;                   // All columns, for insert

/* C struct representing the row layout */
struct MyRow
{
  unsigned int myId;

  /* Pointer to Blob handle for operations on the blob column
   * Space must be left for it in the row, but a pointer to the
   * blob handle can also be obtained via calls to
   * NdbOperation::getBlobHandle()
   */
  NdbBlob* myText;
};

static void setup_records(Ndb *myNdb)
{
  NdbDictionary::RecordSpecification spec[2];

  NdbDictionary::Dictionary *myDict= myNdb->getDictionary();
  const NdbDictionary::Table *myTable= myDict->getTable("api_blob_ndbrecord");
  if (myTable == NULL)
    APIERROR(myDict->getNdbError());
  const NdbDictionary::Column *col1= myTable->getColumn("my_id");
  if (col1 == NULL)
    APIERROR(myDict->getNdbError());
  const NdbDictionary::Column *col2= myTable->getColumn("my_text");
  if (col2 == NULL)
    APIERROR(myDict->getNdbError());

  spec[0].column= col1;
  spec[0].offset= offsetof(MyRow, myId);
  spec[0].nullbit_byte_offset= 0;
  spec[0].nullbit_bit_in_byte= 0;
  spec[1].column= col2;
  spec[1].offset= offsetof(MyRow, myText);
  spec[1].nullbit_byte_offset= 0;
  spec[1].nullbit_bit_in_byte= 0;

  key_record= myDict->createRecord(myTable, &spec[0], 1, sizeof(spec[0]));
  if (key_record == NULL)
    APIERROR(myDict->getNdbError());
  blob_record= myDict->createRecord(myTable, &spec[1], 1, sizeof(spec[0]));
  if (blob_record == NULL)
    APIERROR(myDict->getNdbError());
  full_record= myDict->createRecord(myTable, &spec[0], 2, sizeof(spec[0]));
  if (full_record == NULL)
    APIERROR(myDict->getNdbError());
}

/*
  Function to drop table.
*/
void drop_table(MYSQL &mysql)
{
  if (mysql_query(&mysql, "DROP TABLE api_blob_ndbrecord"))
    MYSQLERROR(mysql);
}


/*
  Functions to create table.
*/
int try_create_table(MYSQL &mysql)
```

```
{
  return mysql_query(&mysql,
                     "CREATE TABLE"
                     "  api_blob_ndbrecord"
                     "    (my_id INT UNSIGNED NOT NULL,"
                     "     my_text TEXT NOT NULL,"
                     "     PRIMARY KEY USING HASH (my_id))"
                     "  ENGINE=NDB");
}

void create_table(MYSQL &mysql)
{
  if (try_create_table(mysql))
  {
    if (mysql_errno(&mysql) != ER_TABLE_EXISTS_ERROR)
      MYSQLERROR(mysql);
    std::cout << "NDB Cluster already has example table: api_blob_ndbrecord. "
              << "Dropping it..." << std::endl;
    /******************
     * Recreate table *
     ******************/
    drop_table(mysql);
    if (try_create_table(mysql))
      MYSQLERROR(mysql);
  }
}

int populate(Ndb *myNdb)
{
  MyRow row;

  NdbTransaction *myTrans= myNdb->startTransaction();
  if (myTrans == NULL)
    APIERROR(myNdb->getNdbError());

  row.myId= 1;
  const NdbOperation *myNdbOperation= myTrans->insertTuple(full_record, (const char*) &row);
  if (myNdbOperation == NULL)
    APIERROR(myTrans->getNdbError());

  NdbBlob *myBlobHandle= myNdbOperation->getBlobHandle("my_text");
  if (myBlobHandle == NULL)
    APIERROR(myNdbOperation->getNdbError());
  myBlobHandle->setValue(text_quote, strlen(text_quote));

  int check= myTrans->execute(NdbTransaction::Commit);
  myTrans->close();
  return check != -1;
}


int update_key(Ndb *myNdb)
{
  MyRow row;

  /*
    Uppercase all characters in TEXT field, using primary key operation.
    Use piece-wise read/write to avoid loading entire data into memory
    at once.
  */

  NdbTransaction *myTrans= myNdb->startTransaction();
  if (myTrans == NULL)
    APIERROR(myNdb->getNdbError());

  row.myId= 1;

  const NdbOperation *myNdbOperation=
    myTrans->updateTuple(key_record,
                         (const char*) &row,
                         blob_record,
                         (const char*) &row);
```

```
  if (myNdbOperation == NULL)
    APIERROR(myTrans->getNdbError());

  NdbBlob *myBlobHandle= myNdbOperation->getBlobHandle("my_text");
  if (myBlobHandle == NULL)
    APIERROR(myNdbOperation->getNdbError());

  /* Execute NoCommit to make the blob handle active so
   * that we can determine the actual Blob length
   */
  if (-1 == myTrans->execute(NdbTransaction::NoCommit))
    APIERROR(myTrans->getNdbError());

  Uint64 length= 0;
  if (-1 == myBlobHandle->getLength(length))
    APIERROR(myBlobHandle->getNdbError());

  /*
    A real application should use a much larger chunk size for
    efficiency, preferably much larger than the part size, which
    defaults to 2000. 64000 might be a good value.
  */
#define CHUNK_SIZE 100
  int chunk;
  char buffer[CHUNK_SIZE];
  for (chunk= (length-1)/CHUNK_SIZE; chunk >=0; chunk--)
  {
    Uint64 pos= chunk*CHUNK_SIZE;
    Uint32 chunk_length= CHUNK_SIZE;
    if (pos + chunk_length > length)
      chunk_length= length - pos;

    /* Read from the end back, to illustrate seeking. */
    if (-1 == myBlobHandle->setPos(pos))
      APIERROR(myBlobHandle->getNdbError());
    if (-1 == myBlobHandle->readData(buffer, chunk_length))
      APIERROR(myBlobHandle->getNdbError());
    int res= myTrans->execute(NdbTransaction::NoCommit);
    if (-1 == res)
      APIERROR(myTrans->getNdbError());

    /* Uppercase everything. */
    for (Uint64 j= 0; j < chunk_length; j++)
      buffer[j]= toupper(buffer[j]);

    if (-1 == myBlobHandle->setPos(pos))
      APIERROR(myBlobHandle->getNdbError());
    if (-1 == myBlobHandle->writeData(buffer, chunk_length))
      APIERROR(myBlobHandle->getNdbError());
    /* Commit on the final update. */
    if (-1 == myTrans->execute(chunk ?
                               NdbTransaction::NoCommit :
                               NdbTransaction::Commit))
      APIERROR(myTrans->getNdbError());
  }

  myNdb->closeTransaction(myTrans);

  return 1;
}


int update_scan(Ndb *myNdb)
{
  /*
    Lowercase all characters in TEXT field, using a scan with
    updateCurrentTuple().
  */
  char buffer[10000];

  NdbTransaction *myTrans= myNdb->startTransaction();
  if (myTrans == NULL)
```

```
    APIERROR(myNdb->getNdbError());

  NdbScanOperation *myScanOp=
    myTrans->scanTable(blob_record, NdbOperation::LM_Exclusive);
  if (myScanOp == NULL)
    APIERROR(myTrans->getNdbError());
  NdbBlob *myBlobHandle= myScanOp->getBlobHandle("my_text");
  if (myBlobHandle == NULL)
    APIERROR(myScanOp->getNdbError());
  if (myBlobHandle->getValue(buffer, sizeof(buffer)))
    APIERROR(myBlobHandle->getNdbError());

  /* Start the scan. */
  if (-1 == myTrans->execute(NdbTransaction::NoCommit))
    APIERROR(myTrans->getNdbError());

  const MyRow *out_row;
  int res;
  for (;;)
  {
    res= myScanOp->nextResult((const char**)&out_row, true, false);
    if (res==1)
      break;                                    // Scan done.
    else if (res)
      APIERROR(myScanOp->getNdbError());

    Uint64 length= 0;
    if (myBlobHandle->getLength(length) == -1)
      APIERROR(myBlobHandle->getNdbError());

    /* Lowercase everything. */
    for (Uint64 j= 0; j < length; j++)
      buffer[j]= tolower(buffer[j]);

    /* 'Take over' the row locks from the scan to a separate
     * operation for updating the tuple
     */
    const NdbOperation *myUpdateOp=
      myScanOp->updateCurrentTuple(myTrans,
                                   blob_record,
                                   (const char*)out_row);
    if (myUpdateOp == NULL)
      APIERROR(myTrans->getNdbError());
    NdbBlob *myBlobHandle2= myUpdateOp->getBlobHandle("my_text");
    if (myBlobHandle2 == NULL)
      APIERROR(myUpdateOp->getNdbError());
    if (myBlobHandle2->setValue(buffer, length))
      APIERROR(myBlobHandle2->getNdbError());

    if (-1 == myTrans->execute(NdbTransaction::NoCommit))
      APIERROR(myTrans->getNdbError());
  }

  if (-1 == myTrans->execute(NdbTransaction::Commit))
    APIERROR(myTrans->getNdbError());

  myNdb->closeTransaction(myTrans);

  return 1;
}


struct ActiveHookData {
  char buffer[10000];
  Uint32 readLength;
};

int myFetchHook(NdbBlob* myBlobHandle, void* arg)
{
  ActiveHookData *ahd= (ActiveHookData *)arg;

  ahd->readLength= sizeof(ahd->buffer) - 1;
```

```
  return myBlobHandle->readData(ahd->buffer, ahd->readLength);
}

int fetch_key(Ndb *myNdb)
{
  /* Fetch a blob without specifying how many bytes
   * to read up front, in one execution using
   * the 'ActiveHook' mechanism.
   * The supplied ActiveHook procedure is called when
   * the Blob handle becomes 'active'.  At that point
   * the length of the Blob can be obtained, and buffering
   * arranged, and the data read requested.
   */

  /* Separate rows used to specify key and hold result */
  MyRow key_row;
  MyRow out_row;

  /*
    Fetch and show the blob field, using setActiveHook().
  */

  NdbTransaction *myTrans= myNdb->startTransaction();
  if (myTrans == NULL)
    APIERROR(myNdb->getNdbError());

  key_row.myId= 1;
  out_row.myText= NULL;
  const NdbOperation *myNdbOperation=
    myTrans->readTuple(key_record,
                       (const char*) &key_row,
                       blob_record,
                       (char*) &out_row);
  if (myNdbOperation == NULL)
    APIERROR(myTrans->getNdbError());

  /* This time, we'll get the blob handle from the row, because
   * we can.  Alternatively, we could use the normal mechanism
   * of calling getBlobHandle().
   */
  NdbBlob *myBlobHandle= out_row.myText;
  if (myBlobHandle == NULL)
    APIERROR(myNdbOperation->getNdbError());
  struct ActiveHookData ahd;
  if (myBlobHandle->setActiveHook(myFetchHook, &ahd) == -1)
    APIERROR(myBlobHandle->getNdbError());

  /*
    Execute Commit, but calling our callback set up in setActiveHook()
    before actually committing.
  */
  if (-1 == myTrans->execute(NdbTransaction::Commit))
    APIERROR(myTrans->getNdbError());
  myNdb->closeTransaction(myTrans);

  /* Our fetch callback will have been called during the execute(). */

  ahd.buffer[ahd.readLength]= '\0';
  std::cout << "Fetched data:" << std::endl << ahd.buffer << std::endl;

  return 1;
}


int update2_key(Ndb *myNdb)
{
  char buffer[10000];
  MyRow row;

  /* Simple setValue() update specified before the
   * Blob handle is made active
   */
```

```
  NdbTransaction *myTrans= myNdb->startTransaction();
  if (myTrans == NULL)
    APIERROR(myNdb->getNdbError());

  row.myId= 1;
  const NdbOperation *myNdbOperation=
    myTrans->updateTuple(key_record,
                         (const char*)&row,
                         blob_record,
                         (char*) &row);
  if (myNdbOperation == NULL)
    APIERROR(myTrans->getNdbError());
  NdbBlob *myBlobHandle= myNdbOperation->getBlobHandle("my_text");
  if (myBlobHandle == NULL)
    APIERROR(myNdbOperation->getNdbError());
  memset(buffer, ' ', sizeof(buffer));
  if (myBlobHandle->setValue(buffer, sizeof(buffer)) == -1)
    APIERROR(myBlobHandle->getNdbError());

  if (-1 == myTrans->execute(NdbTransaction::Commit))
    APIERROR(myTrans->getNdbError());
  myNdb->closeTransaction(myTrans);

  return 1;
}


int delete_key(Ndb *myNdb)
{
  MyRow row;

  /* Deletion of row containing blob via primary key. */

  NdbTransaction *myTrans= myNdb->startTransaction();
  if (myTrans == NULL)
    APIERROR(myNdb->getNdbError());

  row.myId= 1;
  const NdbOperation *myNdbOperation= myTrans->deleteTuple(key_record,
                                                           (const char*)&row,
                                                           full_record);
  if (myNdbOperation == NULL)
    APIERROR(myTrans->getNdbError());

  if (-1 == myTrans->execute(NdbTransaction::Commit))
    APIERROR(myTrans->getNdbError());
  myNdb->closeTransaction(myTrans);

  return 1;
}

void mysql_connect_and_create(const char *socket)
{
  MYSQL mysql;
  bool ok;

  mysql_init(&mysql);

  ok = mysql_real_connect(&mysql, "localhost", "root", "", "", 0, socket, 0);
  if(ok) {
    mysql_query(&mysql, "CREATE DATABASE ndb_examples");
    ok = ! mysql_select_db(&mysql, "ndb_examples");
  }
  if(ok) {
    create_table(mysql);
  }
  mysql_close(&mysql);

  if(! ok) MYSQLERROR(mysql);
}
```

```
void ndb_run_ndbrecord_blob_operations(const char * connectstring)
{
  /* Connect to ndb cluster. */

  Ndb_cluster_connection cluster_connection(connectstring);
  if (cluster_connection.connect(4, 5, 1))
  {
    std::cout << "Unable to connect to cluster within 30 secs." << std::endl;
    exit(-1);
  }
  /* Optionally connect and wait for the storage nodes (ndbd's). */
  if (cluster_connection.wait_until_ready(30,0) < 0)
  {
    std::cout << "Cluster was not ready within 30 secs.\n";
    exit(-1);
  }

  Ndb myNdb(&cluster_connection,"ndb_examples");
  if (myNdb.init(1024) == -1) {      // Set max 1024 parallel transactions
    APIERROR(myNdb.getNdbError());
    exit(-1);
  }

  setup_records(&myNdb);

  if(populate(&myNdb) > 0)
    std::cout << "populate: Success!" << std::endl;

  if(update_key(&myNdb) > 0)
    std::cout << "update_key: Success!" << std::endl;

  if(update_scan(&myNdb) > 0)
    std::cout << "update_scan: Success!" << std::endl;

  if(fetch_key(&myNdb) > 0)
    std::cout << "fetch_key: Success!" << std::endl;

  if(update2_key(&myNdb) > 0)
    std::cout << "update2_key: Success!" << std::endl;

  if(delete_key(&myNdb) > 0)
    std::cout << "delete_key: Success!" << std::endl;
}

int main(int argc, char**argv)
{
  if (argc != 3)
  {
    std::cout << "Arguments are <socket mysqld> <connect_string cluster>.\n";
    exit(-1);
  }
  char *mysqld_sock  = argv[1];
  const char *connectstring = argv[2];

  mysql_connect_and_create(mysqld_sock);

  ndb_init();
  ndb_run_ndbrecord_blob_operations(connectstring);
  ndb_end(0);

  return 0;
}
```

2.5.11 NDB API Simple Array Example

This program inserts `CHAR`, `VARCHAR`, and `BINARY` column data into a table by constructing `aRef` objects using local functions. It then reads the columns back and extracts the data from them using local functions.

This example assumes you have a table named `api_array_simple`, created as follows:

```
CREATE TABLE api_array_simple (
    ATTR1 INT UNSIGNED NOT NULL PRIMARY KEY,
    ATTR2 CHAR(20) NOT NULL,
    ATTR3 VARCHAR(20) NOT NULL,
    ATTR4 VARCHAR(500) NOT NULL,
    ATTR5 BINARY(20) NOT NULL,
    ATTR6 VARBINARY(20) NOT NULL,
    ATTR7 VARBINARY(500) NOT NULL
) ENGINE NDB CHARSET LATIN1;
```

Note

This program uses a number of utilities which can be found in `storage/ndb/ndbapi-examples/common/`. See Section 2.5.13, “Common Files for NDB API Array Examples”, for listings of these.

The example file can be found as `ndbapi_array_simple/ndbapi_array_simple.cpp` in the NDB 7.3.8, NDB 7.4.3, or later NDB Cluster source distribution's `storage/ndb/ndbapi-examples` directory. (Bug #70550, Bug #17592990)

```
#include <NdbApi.hpp>
#include <iostream>
#include <vector>
#include <cstdlib>
#include <cstring>

/*
  See Section 2.5.13, “Common Files for NDB API Array Examples”,
  for listings of these utilities.
*/
#include "../common/error_handling.hpp"
#include "../common/ndb_util.hpp"
#include "../common/util.hpp"

using namespace std;

/* structure to help in insertion */
struct RowData
{
  /* id */
  int attr1;
  /* CHAR(20)- fixed length, no additional length bytes */
  char attr2[20];
  /* VARCHAR(20) - requires one additional length byte (length < 256 ) */
  char attr3[1 + 20];
  /* VARCHAR(500) - requires two additional length bytes (length > 256 ) */
  char attr4[2 + 500];
  /* BINARY(20) - fixed length, requires no additional length byte */
  char attr5[20];
  /* VARBINARY(20) - requires one additional length byte (length < 256 ) */
  char attr6[1 + 20];
  /* VARBINARY(20) - requires one additional length byte (length > 256 ) */
  char attr7[2 + 500];
};

/* extracts the length and the start byte of the data stored */
static int get_byte_array(const NdbRecAttr* attr,
                          const char*& first_byte,
                          size_t& bytes)
{
  const NdbDictionary::Column::ArrayType array_type =
    attr->getColumn()->getArrayType();
  const size_t attr_bytes = attr->get_size_in_bytes();
  const char* aRef = attr->aRef();
  string result;

  switch (array_type) {
  case NdbDictionary::Column::ArrayTypeFixed:
    /*
```

```
     No prefix length is stored in aRef. Data starts from aRef's first byte
     data might be padded with blank or null bytes to fill the whole column
     */
    first_byte = aRef;
    bytes = attr_bytes;
    return 0;
  case NdbDictionary::Column::ArrayTypeShortVar:
    /*
     First byte of aRef has the length of data stored
     Data starts from second byte of aRef
     */
    first_byte = aRef + 1;
    bytes = (size_t)(aRef[0]);
    return 0;
  case NdbDictionary::Column::ArrayTypeMediumVar:
    /*
     First two bytes of aRef has the length of data stored
     Data starts from third byte of aRef
     */
    first_byte = aRef + 2;
    bytes = (size_t)(aRef[1]) * 256 + (size_t)(aRef[0]);
    return 0;
  default:
    first_byte = NULL;
    bytes = 0;
    return -1;
  }
}

/*
 Extracts the string from given NdbRecAttr
 Uses get_byte_array internally
 */
static int get_string(const NdbRecAttr* attr, string& str)
{
  size_t attr_bytes;
  const char* data_start_ptr = NULL;

  /* get stored length and data using get_byte_array */
  if(get_byte_array(attr, data_start_ptr, attr_bytes) == 0)
  {
    /* we have length of the string and start location */
    str= string(data_start_ptr, attr_bytes);
    if(attr->getType() == NdbDictionary::Column::Char)
    {
      /* Fixed Char : remove blank spaces at the end */
      size_t endpos = str.find_last_not_of(" ");
      if( string::npos != endpos )
      {
        str = str.substr(0, endpos+1);
      }
    }
  }
  return 0;
}

/*****************************************************
 * Use one transaction and insert 21 rows in one batch *
 *****************************************************/
static void do_insert(Ndb& ndb)
{
  const NdbDictionary::Dictionary* dict = ndb.getDictionary();
  const NdbDictionary::Table *table = dict->getTable("api_array_simple");

  if (table == NULL) APIERROR(dict->getNdbError());

  NdbTransaction *transaction= ndb.startTransaction();
  if (transaction == NULL) APIERROR(ndb.getNdbError());

  /* Create and initialize sample data */
  const string meter = 50 * string("''''-,,,,|");
  const string space = 20 * string(" ");
```

```
unsigned char binary_meter[500];
for (unsigned i = 0; i < 500; i++)
{
  binary_meter[i] = (unsigned char)(i % 256);
}

vector<NdbOperation*> operations;
for (int i = 0; i <= 20; i++)
{
  RowData data;
  NdbOperation* myOperation = transaction->getNdbOperation(table);
  if (myOperation == NULL) APIERROR(transaction->getNdbError());
  data.attr1 = i;

  // Fill CHAR(20) with 'i' chars from meter
  strncpy (data.attr2, meter.c_str(), i);
  // Pad it with space up to 20 chars
  strncpy (data.attr2 + i, space.c_str(), 20 - i);

  // Fill VARCHAR(20) with 'i' chars from meter. First byte is
  // reserved for length field. No padding is needed.
  strncpy (data.attr3 + 1, meter.c_str(), i);
  // Set the length byte
  data.attr3[0] = (char)i;

  // Fill VARCHAR(500) with 20*i chars from meter. First two bytes
  // are reserved for length field. No padding is needed.
  strncpy (data.attr4 + 2, meter.c_str(), 20*i);
  // Set the length bytes
  data.attr4[0] = (char)(20*i % 256);
  data.attr4[1] = (char)(20*i / 256);

  // Fill BINARY(20) with 'i' bytes from binary_meter.
  memcpy(data.attr5, binary_meter, i);
  // Pad with 0 up to 20 bytes.
  memset(data.attr5 + i, 0, 20 - i);

  // Fill VARBINARY(20) with 'i' bytes from binary_meter. First byte
  // is reserved for length field. No padding is needed.
  memcpy(data.attr6 + 1, binary_meter, i);
  // Set the length byte
  data.attr6[0] = (char)i;

  // Fill VARBINARY(500) with 'i' bytes from binary_meter. First two
  // bytes are reserved for length filed. No padding is needed.
  memcpy(data.attr7 + 2, binary_meter, 20*i);
  // Set the length bytes
  data.attr7[0] = (char)(20*i % 256);
  data.attr7[1] = (char)(20*i / 256);

  myOperation->insertTuple();
  myOperation->equal("ATTR1", data.attr1);
  myOperation->setValue("ATTR2", data.attr2);
  myOperation->setValue("ATTR3", data.attr3);
  myOperation->setValue("ATTR4", data.attr4);
  myOperation->setValue("ATTR5", data.attr5);
  myOperation->setValue("ATTR6", data.attr6);
  myOperation->setValue("ATTR7", data.attr7);

  operations.push_back(myOperation);
}

// Now execute all operations in one batch, and check for errors.
if (transaction->execute( NdbTransaction::Commit ) != 0)
{
  for (size_t i = 0; i < operations.size(); i++)
  {
    const NdbError err= operations[i]->getNdbError();
    if(err.code != NdbError::Success)
    {
      cout << "Error inserting Row : " << i << endl;
      PRINT_ERROR(err.code, err.message);
```

```
      }
    }
    APIERROR(transaction->getNdbError());
  }
  ndb.closeTransaction(transaction);
}

/*
 Reads the row with id = 17
 Retrieves an prints value of the [VAR]CHAR/BINARY
 */
static void do_read(Ndb& ndb)
{
  const NdbDictionary::Dictionary* dict= ndb.getDictionary();
  const NdbDictionary::Table* table= dict->getTable("api_array_simple");

  if (table == NULL) APIERROR(dict->getNdbError());

  NdbTransaction *transaction= ndb.startTransaction();
  if (transaction == NULL) APIERROR(ndb.getNdbError());

  NdbOperation *operation= transaction->getNdbOperation(table);
  if (operation == NULL) APIERROR(transaction->getNdbError());

  /* create and execute a read operation */
  operation->readTuple(NdbOperation::LM_Read);
  operation->equal("ATTR1", 17);

  vector<NdbRecAttr*> attr;
  const int column_count= table->getNoOfColumns();
  attr.reserve(column_count);

  for (int i= 1; i < column_count; i++)
  {
    attr[i] = operation->getValue(i, NULL);
    if (attr[i] == NULL) APIERROR(transaction->getNdbError());
  }

  if(transaction->execute( NdbTransaction::Commit ) == -1)
    APIERROR(transaction->getNdbError());

  /* print the fetched data */
  cout << "Row ID : 17\n";
  for (int i= 1; i < column_count; i++)
  {
    if (attr[i] != NULL)
    {
      NdbDictionary::Column::Type column_type = attr[i]->getType();
      cout << "Column id: " << i << ", name: " << attr[i]->getColumn()->getName()
           << ", size: " << attr[i]->get_size_in_bytes()
           << ", type: " << column_type_to_string(attr[i]->getType());
      switch (column_type) {
      case NdbDictionary::Column::Char:
      case NdbDictionary::Column::Varchar:
      case NdbDictionary::Column::Longvarchar:
        {
          /* for char columns the actual string is printed */
          string str;
          get_string(attr[i], str);
          cout << ", stored string length: " << str.length()
               << ", value: " << str << endl;
        }
        break;
      case NdbDictionary::Column::Binary:
      case NdbDictionary::Column::Varbinary:
      case NdbDictionary::Column::Longvarbinary:
        {
          /* for binary columns the sum of all stored bytes is printed */
          const char* first;
          size_t count;
          get_byte_array(attr[i], first, count);
          int sum = 0;
```

```
          for (const char* byte = first; byte < first + count; byte++)
          {
            sum += (int)(*byte);
          }
          cout << ", stored bytes length: " << count
               << ", sum of byte array: " << sum << endl;
        }
        break;
      default:
        cout << ", column type \"" << column_type_to_string(attr[i]->getType())
             << "\" not covered by this example" << endl;
        break;
      }
    }
  }

  ndb.closeTransaction(transaction);
}

static void run_application(Ndb_cluster_connection &cluster_connection,
                            const char* database_name)
{
  /********************************************
   * Connect to database via NDB API           *
   ********************************************/
  // Object representing the database
  Ndb ndb( &cluster_connection, database_name);
  if (ndb.init()) APIERROR(ndb.getNdbError());

  /*
   * Do different operations on database
   */
  do_insert(ndb);
  do_read(ndb);
}

int main(int argc, char** argv)
{
  if (argc != 3)
  {
    std::cout << "Arguments are <connect_string cluster> <database_name>.\n";
    exit(-1);
  }
  /* ndb_init must be called first */
  ndb_init();
  {
    /* connect to cluster */
    const char *connectstring = argv[1];
    Ndb_cluster_connection cluster_connection(connectstring);
    if (cluster_connection.connect(30 /* retries */,
                                   1  /* delay between retries */,
                                   0  /* verbose */))
    {
      std::cout << "Cluster management server was not ready within 30 secs.\n";
      exit(-1);
    }

    /* Connect and wait for the storage nodes */
    if (cluster_connection.wait_until_ready(30,10) < 0)
    {
      std::cout << "Cluster was not ready within 30 secs.\n";
      exit(-1);
    }

    /* run the application code */
    const char* dbname = argv[2];
    run_application(cluster_connection, dbname);
  }
  ndb_end(0);

  return 0;
}
```

2.5.12 NDB API Simple Array Example Using Adapter

This program inserts `CHAR`, `VARCHAR`, and `BINARY` column data into a table by constructing aRef objects using array adapters of the type defined in `common/array_adapter.hpp` (see Section 2.5.13, “Common Files for NDB API Array Examples”). It then reads the columns back and extracts the data, again using array adapters.

The example uses the table shown here:

```
CREATE TABLE api_array_using_adapter (
  ATTR1 INT UNSIGNED NOT NULL PRIMARY KEY,
  ATTR2 CHAR(20) NOT NULL,
  ATTR3 VARCHAR(20) NOT NULL,
  ATTR4 VARCHAR(500) NOT NULL,
  ATTR5 BINARY(20) NOT NULL,
  ATTR6 VARBINARY(20) NOT NULL,
  ATTR7 VARBINARY(500) NOT NULL
) ENGINE NDB CHARSET LATIN1;
```

The example file can be found as `ndbapi_array_using_adapter/ndbapi_array_using_adapter.cpp` in the NDB 7.3.8, NDB 7.4.3, or later NDB Cluster source distribution's `storage/ndb/ndbapi-examples` directory. (Bug #70550, Bug #17592990)

```
#include &lt;NdbApi.hpp&gt;
#include &lt;iostream&gt;
#include &lt;vector&gt;
#include &lt;cstdlib&gt;
#include &lt;cstring&gt;

using namespace std;

/*
  See Section 2.5.13, “Common Files for NDB API Array Examples”,
  for listings of these utilities.
*/
#include "../common/error_handling.hpp"
#include "../common/array_adapter.hpp"
#include "../common/ndb_util.hpp"
#include "../common/util.hpp"


// Use one transaction and insert 21 rows in one batch.
static void do_insert(Ndb&amp; ndb)
{
  const NdbDictionary::Dictionary* dict = ndb.getDictionary();
  const NdbDictionary::Table *table = dict-&gt;getTable("api_array_using_adapter");

  if (table == NULL)
  {
    APIERROR(dict-&gt;getNdbError());
  }

  // Get a column object for each CHAR/VARCHAR/BINARY/VARBINARY column
  // to insert into.
  const NdbDictionary::Column *column2 = table-&gt;getColumn("ATTR2");
  if (column2 == NULL)
  {
    APIERROR(dict-&gt;getNdbError());
  }

  const NdbDictionary::Column *column3 = table-&gt;getColumn("ATTR3");
  if (column3 == NULL)
  {
    APIERROR(dict-&gt;getNdbError());
  }

  const NdbDictionary::Column *column4 = table-&gt;getColumn("ATTR4");
```

```
if (column4 == NULL)
{
  APIERROR(dict-&gt;getNdbError());
}

const NdbDictionary::Column *column5 = table-&gt;getColumn("ATTR5");
if (column5 == NULL)
{
  APIERROR(dict-&gt;getNdbError());
}

const NdbDictionary::Column *column6 = table-&gt;getColumn("ATTR6");
if (column6 == NULL)
{
  APIERROR(dict-&gt;getNdbError());
}

const NdbDictionary::Column *column7 = table-&gt;getColumn("ATTR7");
if (column7 == NULL)
{
  APIERROR(dict-&gt;getNdbError());
}

// Create a read/write attribute adapter to be used for all
// CHAR/VARCHAR/BINARY/VARBINARY columns.
ReadWriteArrayAdapter attr_adapter;

// Create and initialize sample data.
const string meter = 50 * string("''''-,,,,|");
unsigned char binary_meter[500];
for (unsigned i = 0; i &lt; 500; i++)
{
  binary_meter[i] = (unsigned char)(i % 256);
}

NdbTransaction *transaction= ndb.startTransaction();
if (transaction == NULL) APIERROR(ndb.getNdbError());

// Create 21 operations and put a reference to them in a vector to
// be able to find failing operations.
vector&lt;NdbOperation*&gt; operations;
for (int i = 0; i &lt;= 20; i++)
{
  NdbOperation* operation = transaction-&gt;getNdbOperation(table);
  if (operation == NULL) APIERROR(transaction-&gt;getNdbError());
  operation-&gt;insertTuple();

  operation-&gt;equal("ATTR1", i);

  /* use ReadWrite Adapter to convert string to aRefs */
  ReadWriteArrayAdapter::ErrorType error;

  char *attr2_aRef;
  attr2_aRef= attr_adapter.make_aRef(column2, meter.substr(0,i), error);
  PRINT_IF_NOT_EQUAL(error, ReadWriteArrayAdapter::Success,
                     "make_aRef failed for ATTR2");
  operation-&gt;setValue("ATTR2", attr2_aRef);

  char *attr3_aRef;
  attr3_aRef= attr_adapter.make_aRef(column3, meter.substr(0,i), error);
  PRINT_IF_NOT_EQUAL(error, ReadWriteArrayAdapter::Success,
                     "make_aRef failed for ATTR3");
  operation-&gt;setValue("ATTR3", attr3_aRef);

  char *attr4_aRef;
  attr4_aRef= attr_adapter.make_aRef(column4, meter.substr(0,20*i), error);
  PRINT_IF_NOT_EQUAL(error, ReadWriteArrayAdapter::Success,
                     "make_aRef failed for ATTR4");
  operation-&gt;setValue("ATTR4", attr4_aRef);

  char* attr5_aRef;
  char* attr5_first;
```

```
    attr_adapter.allocate_in_bytes(column5, attr5_aRef, attr5_first, i, error);
    PRINT_IF_NOT_EQUAL(error, ReadWriteArrayAdapter::Success,
                       "allocate_in_bytes failed for ATTR5");
    memcpy(attr5_first, binary_meter, i);
    operation-&gt;setValue("ATTR5", attr5_aRef);

    char* attr6_aRef;
    char* attr6_first;
    attr_adapter.allocate_in_bytes(column6, attr6_aRef, attr6_first, i, error);
    PRINT_IF_NOT_EQUAL(error, ReadWriteArrayAdapter::Success,
                       "allocate_in_bytes failed for ATTR6");
    memcpy(attr6_first, binary_meter, i);
    operation-&gt;setValue("ATTR6", attr6_aRef);

    char* attr7_aRef;
    char* attr7_first;
    attr_adapter.allocate_in_bytes(column7, attr7_aRef, attr7_first, 20*i, error);
    PRINT_IF_NOT_EQUAL(error, ReadWriteArrayAdapter::Success,
                       "allocate_in_bytes failed for ATTR7");
    memcpy(attr7_first, binary_meter, 20*i);
    operation-&gt;setValue("ATTR7", attr7_aRef);

    operations.push_back(operation);
  }

  // Now execute all operations in one batch, and check for errors.
  if (transaction-&gt;execute( NdbTransaction::Commit ) != 0)
  {
    for (size_t i = 0; i &lt; operations.size(); i++)
    {
      const NdbError err= operations[i]-&gt;getNdbError();
      if(err.code != NdbError::Success)
      {
        cout &lt;&lt; "Error inserting Row : " &lt;&lt; i &lt;&lt; endl;
        PRINT_ERROR(err.code, err.message);
      }
    }
    APIERROR(transaction-&gt;getNdbError());
  }
  ndb.closeTransaction(transaction);
}

/*
 Reads the row with id = 17
 Retrieves an prints value of the [VAR]CHAR/BINARY using array_adapter
 */
static void do_read(Ndb&amp; ndb)
{
  const NdbDictionary::Dictionary* dict= ndb.getDictionary();
  const NdbDictionary::Table* table= dict-&gt;getTable("api_array_using_adapter");

  if (table == NULL) APIERROR(dict-&gt;getNdbError());

  NdbTransaction *transaction= ndb.startTransaction();
  if (transaction == NULL) APIERROR(ndb.getNdbError());

  NdbOperation *operation= transaction-&gt;getNdbOperation(table);
  if (operation == NULL) APIERROR(transaction-&gt;getNdbError());

  operation-&gt;readTuple(NdbOperation::LM_Read);
  operation-&gt;equal("ATTR1", 17);

  vector&lt;NdbRecAttr*&gt; attr;
  const int column_count= table-&gt;getNoOfColumns();
  attr.reserve(column_count);

  for (int i= 1; i &lt; column_count; i++)
  {
    attr[i] = operation-&gt;getValue(i, NULL);
    if (attr[i] == NULL) APIERROR(transaction-&gt;getNdbError());
  }
```

```
  if(transaction-&gt;execute( NdbTransaction::Commit ) == -1)
    APIERROR(transaction-&gt;getNdbError());

  /* Now use an array adapter to read the data from columns */
  const ReadOnlyArrayAdapter attr_adapter;
  ReadOnlyArrayAdapter::ErrorType error;

  /* print the fetched data */
  cout &lt;&lt; "Row ID : 17\n";
  for (int i= 1; i &lt; column_count; i++)
  {
    if (attr[i] != NULL)
    {
      NdbDictionary::Column::Type column_type = attr[i]-&gt;getType();
      cout &lt;&lt; "Column id: " &lt;&lt; i
           &lt;&lt; ", name: " &lt;&lt; attr[i]-&gt;getColumn()-&gt;getName()
           &lt;&lt; ", size: " &lt;&lt; attr[i]-&gt;get_size_in_bytes()
           &lt;&lt; ", type: " &lt;&lt; column_type_to_string(attr[i]-&gt;getType());
      if(attr_adapter.is_binary_array_type(column_type))
      {
        /* if column is [VAR]BINARY, get the byte array and print their sum */
        const char* data_ptr;
        size_t data_length;
        attr_adapter.get_byte_array(attr[i], data_ptr,
                                    data_length, error);
        if(error == ReadOnlyArrayAdapter::Success)
        {
          int sum = 0;
          for (size_t j = 0; j &lt; data_length; j++)
            sum += (int)(data_ptr[j]);
          cout &lt;&lt; ", stored bytes length: " &lt;&lt; data_length
               &lt;&lt; ", sum of byte array: " &lt;&lt; sum &lt;&lt; endl;
        }
        else
          cout &lt;&lt; ", error fetching value." &lt;&lt; endl;
      }
      else
      {
        /* if the column is [VAR]CHAR, retrieve the string and print */
        std::string value= attr_adapter.get_string(attr[i], error);
        if(error == ReadOnlyArrayAdapter::Success)
        {
          cout &lt;&lt; ", stored string length: " &lt;&lt; value.length()
               &lt;&lt; ", value: " &lt;&lt; value
               &lt;&lt; endl;
        }
        else
          cout &lt;&lt; ", error fetching value." &lt;&lt; endl;
      }
    }
  }

  ndb.closeTransaction(transaction);
}

static void run_application(Ndb_cluster_connection &amp;cluster_connection,
                            const char* database_name)
{
  /********************************************
   * Connect to database via NDB API           *
   ********************************************/
  // Object representing the database
  Ndb ndb( &amp;cluster_connection, database_name);
  if (ndb.init()) APIERROR(ndb.getNdbError());

  /*
   * Do different operations on database
   */
  do_insert(ndb);
  do_read(ndb);
}
```

```
int main(int argc, char** argv)
{
  if (argc != 3)
  {
    std::cout &lt;&lt; "Arguments are &lt;connect_string cluster&gt; &lt;database_name&gt;.\n";
    exit(-1);
  }
  /* ndb_init must be called first */
  ndb_init();
  {
    /* connect to cluster */
    const char *connectstring = argv[1];
    Ndb_cluster_connection cluster_connection(connectstring);
    if (cluster_connection.connect(30 /* retries */,
                                   1  /* delay between retries */,
                                   0  /* verbose */))
    {
      std::cout &lt;&lt; "Cluster management server was not ready within 30 secs.\n";
      exit(-1);
    }

    /* Connect and wait for the storage nodes */
    if (cluster_connection.wait_until_ready(30,10) &lt; 0)
    {
      std::cout &lt;&lt; "Cluster was not ready within 30 secs.\n";
      exit(-1);
    }

    /* run the application code */
    const char* dbname = argv[2];
    run_application(cluster_connection, dbname);
  }
  ndb_end(0);

  return 0;
}
</programlisting>
  </section>

  <section id="ndbapi-examples-common-files">
    <title>Common Files for Examples</title>
    <para>

    </para>

<programlisting>
</programlisting>
    <para>
      common/
    </para>
<programlisting>

#ifndef ARRAY_ADAPTER_HPP
#define ARRAY_ADAPTER_HPP

#include <algorithm>
#include <assert.h>

/*
 Utility classes to convert between C++ strings/byte arrays and the
 internal format used for [VAR]CHAR/BINARY types.

 Base class that can be used for read operations. The column type is
 taken from the NdbRecAttr object, so only one object is needed to
 convert from different [VAR]CHAR/BINARY types. No additional memory
 is allocated.
 */
class ReadOnlyArrayAdapter {
public:
  ReadOnlyArrayAdapter() {}

  enum ErrorType {Success,
```

```
                  InvalidColumnType,
                  InvalidArrayType,
                  InvalidNullColumn,
                  InvalidNullAttribute,
                  InvalidNullaRef,
                  BytesOutOfRange,
                  UnknownError};

  /*
    Return a C++ string from the aRef() value of attr. This value
    will use the column and column type from attr. The advantage is
    for reading; the same ArrayAdapter can be used for multiple
    columns. The disadvantage is; passing an attribute not of
    [VAR]CHAR/BINARY type will result in a traditional exit(-1)
    */
  std::string get_string(const NdbRecAttr* attr,
                         ErrorType& error) const;

  /* Calculate the first_byte and number of bytes in aRef for attr */
  void get_byte_array(const NdbRecAttr* attr,
                      const char*& first_byte,
                      size_t& bytes,
                      ErrorType& error) const;

  /* Check if a column is of type [VAR]BINARY */
  bool is_binary_array_type(const NdbDictionary::Column::Type t) const;

  /* Check if a column is of type [VAR]BINARY or [VAR]CHAR */
  bool is_array_type(const NdbDictionary::Column::Type t) const;
private:
  /* Disable copy constructor */
  ReadOnlyArrayAdapter(const ReadOnlyArrayAdapter& a) {}
};


 /*
  Extension to ReadOnlyArrayAdapter to be used together with
  insert/write/update operations. Memory is allocated for each
  call to make_aRef or allocate_in_bytes. The memory allocated will
  be deallocated by the destructor. To save memory, the scope of an
  instance of this class should not be longer than the life time of
  the transaction. On the other hand, it must be long enough for the
  usage of all references created
  */
class ReadWriteArrayAdapter : public ReadOnlyArrayAdapter {
public:
  ReadWriteArrayAdapter() {}

  /* Destructor, the only place where memory is deallocated */
  ~ReadWriteArrayAdapter();

  /*
   Create a binary representation of the string 's' and return a
   pointer to it. This pointer can later be used as argument to for
   example setValue
   */
  char* make_aRef(const NdbDictionary::Column* column,
                  std::string s,
                  ErrorType& error);

  /*
   Allocate a number of bytes suitable for this column type. aRef
   can later be used as argument to for example setValue. first_byte
   is the first byte to store data to. bytes is the number of bytes
   to allocate
   */
  void allocate_in_bytes(const NdbDictionary::Column* column,
                         char*& aRef,
                         char*& first_byte,
                         size_t bytes,
                         ErrorType& error);
```

```
private:
  /* Disable copy constructor */
  ReadWriteArrayAdapter(const ReadWriteArrayAdapter& a)
    :ReadOnlyArrayAdapter() {}

  /* Record of allocated char arrays to delete by the destructor */
  std::vector<char*> aRef_created;
};


inline ReadWriteArrayAdapter::~ReadWriteArrayAdapter()
{
  for (std::vector<char*>::iterator i = aRef_created.begin();
       i != aRef_created.end();
       ++i) {
    delete [] *i;
  }
}


char*
ReadWriteArrayAdapter::
make_aRef(const NdbDictionary::Column* column,
          std::string input,
          ErrorType& error)
{
  char* new_ref;
  char* data_start;

  /*
   Allocate bytes and push them into the aRef_created vector.
   After this operation, new_ref has a complete aRef to use in insertion
   and data_start has ptr from which data is to be written.
   The new_aref returned is padded completely with blank spaces.
   */
  allocate_in_bytes(column, new_ref, data_start, input.length(), error);

  if(error != Success)
  {
    return NULL;
  }

  /*
   Copy the input string into aRef's data pointer
   without affecting remaining blank spaces at end.
   */
  strncpy(data_start, input.c_str(), input.length());

  return new_ref;
}


void
ReadWriteArrayAdapter::
allocate_in_bytes(const NdbDictionary::Column* column,
                  char*& aRef,
                  char*& first_byte,
                  size_t bytes,
                  ErrorType& error)
{
  bool is_binary;
  char zero_char;
  NdbDictionary::Column::ArrayType array_type;
  size_t max_length;

  /* unless there is going to be any problem */
  error = Success;

  if (column == NULL)
  {
    error = InvalidNullColumn;
    aRef = NULL;
```

```
    first_byte = NULL;
    return;
  }

  if (!is_array_type(column->getType()))
  {
    error = InvalidColumnType;
    aRef = NULL;
    first_byte = NULL;
    return;
  }

  is_binary = is_binary_array_type(column->getType());
  zero_char = (is_binary ? 0 : ' ');
  array_type = column->getArrayType();
  max_length = column->getLength();

  if (bytes > max_length)
  {
    error = BytesOutOfRange;
    aRef = NULL;
    first_byte = NULL;
    return;
  }

  switch (array_type) {
  case NdbDictionary::Column::ArrayTypeFixed:
    /* no need to store length bytes */
    aRef = new char[max_length];
    first_byte = aRef;
    /* pad the complete string with blank space (or) null bytes */
    for (size_t i=0; i < max_length; i++) {
      aRef[i] = zero_char;
    }
    break;
  case NdbDictionary::Column::ArrayTypeShortVar:
    /* byte length stored over first byte. no padding required */
    aRef = new char[1 + bytes];
    first_byte = aRef + 1;
    aRef[0] = (char)bytes;
    break;
  case NdbDictionary::Column::ArrayTypeMediumVar:
    /* byte length stored over first two bytes. no padding required */
    aRef = new char[2 + bytes];
    first_byte = aRef + 2;
    aRef[0] = (char)(bytes % 256);
    aRef[1] = (char)(bytes / 256);
    break;
  }
  aRef_created.push_back(aRef);
}


std::string ReadOnlyArrayAdapter::get_string(const NdbRecAttr* attr,
                                             ErrorType& error) const
{
  size_t attr_bytes= 0;
  const char* data_ptr= NULL;
  std::string result= "";

  /* get the beginning of data and its size.. */
  get_byte_array(attr, data_ptr, attr_bytes, error);

  if(error != Success)
  {
    return result;
  }

  /* ..and copy the  value into result */
  result = string(data_ptr, attr_bytes);

  /* special treatment for FixedArrayType to eliminate padding characters */
```

```
  if(attr->getColumn()->getArrayType() == NdbDictionary::Column::ArrayTypeFixed)
  {
    char padding_char = ' ';
    std::size_t last = result.find_last_not_of(padding_char);
    result = result.substr(0, last+1);
  }

  return result;
}


void
ReadOnlyArrayAdapter::
get_byte_array(const NdbRecAttr* attr,
               const char*& data_ptr,
               size_t& bytes,
               ErrorType& error) const
{
  /* unless there is a problem */
  error= Success;

  if (attr == NULL)
  {
    error = InvalidNullAttribute;
    return;
  }

  if (!is_array_type(attr->getType()))
  {
    error = InvalidColumnType;
    return;
  }

  const NdbDictionary::Column::ArrayType array_type =
      attr->getColumn()->getArrayType();
  const size_t attr_bytes = attr->get_size_in_bytes();
  const char* aRef = attr->aRef();

  if(aRef == NULL)
  {
    error= InvalidNullaRef;
    return;
  }

  switch (array_type) {
  case NdbDictionary::Column::ArrayTypeFixed:
    /* no length bytes stored with aRef */
    data_ptr = aRef;
    bytes = attr_bytes;
    break;
  case NdbDictionary::Column::ArrayTypeShortVar:
    /* first byte of aRef has length of the data */
    data_ptr = aRef + 1;
    bytes = (size_t)(aRef[0]);
    break;
  case NdbDictionary::Column::ArrayTypeMediumVar:
    /* first two bytes of aRef has length of the data */
    data_ptr = aRef + 2;
    bytes = (size_t)(aRef[1]) * 256 + (size_t)(aRef[0]);
    break;
  default:
    /* should never reach here */
    data_ptr = NULL;
    bytes = 0;
    error = InvalidArrayType;
    break;
  }
}


bool
ReadOnlyArrayAdapter::
```

```
is_binary_array_type(const NdbDictionary::Column::Type t) const
{
  bool is_binary;

  switch (t)
  {
  case NdbDictionary::Column::Binary:
  case NdbDictionary::Column::Varbinary:
  case NdbDictionary::Column::Longvarbinary:
    is_binary = true;
    break;
  default:
    is_binary = false;
  }
  return is_binary;
}


bool
ReadOnlyArrayAdapter::
is_array_type(const NdbDictionary::Column::Type t) const
{
  bool is_array;

  switch (t)
  {
  case NdbDictionary::Column::Binary:
  case NdbDictionary::Column::Varbinary:
  case NdbDictionary::Column::Longvarbinary:
  case NdbDictionary::Column::Char:
  case NdbDictionary::Column::Varchar:
  case NdbDictionary::Column::Longvarchar:
    is_array = true;
    break;
  default:
    is_array = false;
  }
  return is_array;
}

#endif // #ifndef ARRAY_ADAPTER_HPP
```

2.5.13 Common Files for NDB API Array Examples

In NDB 7.3.8, NDB 7.4.3, or later NDB Cluster source distribution, the `storage/ndb/ndbapi-examples` directory `storage/ndb/ndbapi-examples/common` contains four header files with utilities for use in example NDB API programs. (Bug #70550, Bug #17592990) The names of these files are listed here:

- `array_adapter.hpp`: Contains utility classes for converting between C++ style strings or byte arrays and the format used by NDB internally for `VARCHAR`, `CHAR`, and `VARBINARY` types.
- `error_handling.hpp`: Contains error handling functions.
- `ndb_util.hpp`: Defines a `column_type_to_string()` function which handles `NDB` column types.
- `util.hpp`: Provides a method for generating strings of arbitrary length.

Following in this section are source listings for each of the header files.

array_adapter.hpp

```
#ifndef ARRAY_ADAPTER_HPP
#define ARRAY_ADAPTER_HPP

#include <algorithm>
#include <assert.h>
```

```
/*
 Utility classes to convert between C++ strings/byte arrays and the
 internal format used for [VAR]CHAR/BINARY types.

 Base class that can be used for read operations. The column type is
 taken from the NdbRecAttr object, so only one object is needed to
 convert from different [VAR]CHAR/BINARY types. No additional memory
 is allocated.
 */
class ReadOnlyArrayAdapter {
public:
  ReadOnlyArrayAdapter() {}

  enum ErrorType {Success,
                  InvalidColumnType,
                  InvalidArrayType,
                  InvalidNullColumn,
                  InvalidNullAttribute,
                  InvalidNullaRef,
                  BytesOutOfRange,
                  UnknownError};

  /*
    Return a C++ string from the aRef() value of attr. This value
    will use the column and column type from attr. The advantage is
    for reading; the same ArrayAdapter can be used for multiple
    columns. The disadvantage is; passing an attribute not of
    [VAR]CHAR/BINARY type will result in a traditional exit(-1)
    */
  std::string get_string(const NdbRecAttr* attr,
                         ErrorType& error) const;

  /* Calculate the first_byte and number of bytes in aRef for attr */
  void get_byte_array(const NdbRecAttr* attr,
                      const char*& first_byte,
                      size_t& bytes,
                      ErrorType& error) const;

  /* Check if a column is of type [VAR]BINARY */
  bool is_binary_array_type(const NdbDictionary::Column::Type t) const;

  /* Check if a column is of type [VAR]BINARY or [VAR]CHAR */
  bool is_array_type(const NdbDictionary::Column::Type t) const;
private:
  /* Disable copy constructor */
  ReadOnlyArrayAdapter(const ReadOnlyArrayAdapter& a) {}
};


 /*
  Extension to ReadOnlyArrayAdapter to be used together with
  insert/write/update operations. Memory is allocated for each
  call to make_aRef or allocate_in_bytes. The memory allocated will
  be deallocated by the destructor. To save memory, the scope of an
  instance of this class should not be longer than the life time of
  the transaction. On the other hand, it must be long enough for the
  usage of all references created
  */
class ReadWriteArrayAdapter : public ReadOnlyArrayAdapter {
public:
  ReadWriteArrayAdapter() {}

  /* Destructor, the only place where memory is deallocated */
  ~ReadWriteArrayAdapter();

  /*
   Create a binary representation of the string 's' and return a
   pointer to it. This pointer can later be used as argument to for
   example setValue
   */
  char* make_aRef(const NdbDictionary::Column* column,
                  std::string s,
```

```
                  ErrorType& error);

  /*
   Allocate a number of bytes suitable for this column type. aRef
   can later be used as argument to for example setValue. first_byte
   is the first byte to store data to. bytes is the number of bytes
   to allocate
   */
  void allocate_in_bytes(const NdbDictionary::Column* column,
                         char*& aRef,
                         char*& first_byte,
                         size_t bytes,
                         ErrorType& error);

private:
  /* Disable copy constructor */
  ReadWriteArrayAdapter(const ReadWriteArrayAdapter& a)
    :ReadOnlyArrayAdapter() {}

  /* Record of allocated char arrays to delete by the destructor */
  std::vector<char*> aRef_created;
};


inline ReadWriteArrayAdapter::~ReadWriteArrayAdapter()
{
  for (std::vector<char*>::iterator i = aRef_created.begin();
       i != aRef_created.end();
       ++i) {
    delete [] *i;
  }
}


char*
ReadWriteArrayAdapter::
make_aRef(const NdbDictionary::Column* column,
          std::string input,
          ErrorType& error)
{
  char* new_ref;
  char* data_start;

  /*
   Allocate bytes and push them into the aRef_created vector.
   After this operation, new_ref has a complete aRef to use in insertion
   and data_start has ptr from which data is to be written.
   The new_aref returned is padded completely with blank spaces.
   */
  allocate_in_bytes(column, new_ref, data_start, input.length(), error);

  if(error != Success)
  {
    return NULL;
  }

  /*
   Copy the input string into aRef's data pointer
   without affecting remaining blank spaces at end.
   */
  strncpy(data_start, input.c_str(), input.length());

  return new_ref;
}


void
ReadWriteArrayAdapter::
allocate_in_bytes(const NdbDictionary::Column* column,
                  char*& aRef,
                  char*& first_byte,
                  size_t bytes,
```

```
                  ErrorType& error)
{
  bool is_binary;
  char zero_char;
  NdbDictionary::Column::ArrayType array_type;
  size_t max_length;

  /* unless there is going to be any problem */
  error = Success;

  if (column == NULL)
  {
    error = InvalidNullColumn;
    aRef = NULL;
    first_byte = NULL;
    return;
  }

  if (!is_array_type(column->getType()))
  {
    error = InvalidColumnType;
    aRef = NULL;
    first_byte = NULL;
    return;
  }

  is_binary = is_binary_array_type(column->getType());
  zero_char = (is_binary ? 0 : ' ');
  array_type = column->getArrayType();
  max_length = column->getLength();

  if (bytes > max_length)
  {
    error = BytesOutOfRange;
    aRef = NULL;
    first_byte = NULL;
    return;
  }

  switch (array_type) {
  case NdbDictionary::Column::ArrayTypeFixed:
    /* no need to store length bytes */
    aRef = new char[max_length];
    first_byte = aRef;
    /* pad the complete string with blank space (or) null bytes */
    for (size_t i=0; i < max_length; i++) {
      aRef[i] = zero_char;
    }
    break;
  case NdbDictionary::Column::ArrayTypeShortVar:
    /* byte length stored over first byte. no padding required */
    aRef = new char[1 + bytes];
    first_byte = aRef + 1;
    aRef[0] = (char)bytes;
    break;
  case NdbDictionary::Column::ArrayTypeMediumVar:
    /* byte length stored over first two bytes. no padding required */
    aRef = new char[2 + bytes];
    first_byte = aRef + 2;
    aRef[0] = (char)(bytes % 256);
    aRef[1] = (char)(bytes / 256);
    break;
  }
  aRef_created.push_back(aRef);
}


std::string ReadOnlyArrayAdapter::get_string(const NdbRecAttr* attr,
                                             ErrorType& error) const
{
  size_t attr_bytes= 0;
  const char* data_ptr= NULL;
```

```
  std::string result= "";

  /* get the beginning of data and its size.. */
  get_byte_array(attr, data_ptr, attr_bytes, error);

  if(error != Success)
  {
    return result;
  }

  /* ..and copy the  value into result */
  result = string(data_ptr, attr_bytes);

  /* special treatment for FixedArrayType to eliminate padding characters */
  if(attr->getColumn()->getArrayType() == NdbDictionary::Column::ArrayTypeFixed)
  {
    char padding_char = ' ';
    std::size_t last = result.find_last_not_of(padding_char);
    result = result.substr(0, last+1);
  }

  return result;
}


void
ReadOnlyArrayAdapter::
get_byte_array(const NdbRecAttr* attr,
               const char*& data_ptr,
               size_t& bytes,
               ErrorType& error) const
{
  /* unless there is a problem */
  error= Success;

  if (attr == NULL)
  {
    error = InvalidNullAttribute;
    return;
  }

  if (!is_array_type(attr->getType()))
  {
    error = InvalidColumnType;
    return;
  }

  const NdbDictionary::Column::ArrayType array_type =
      attr->getColumn()->getArrayType();
  const size_t attr_bytes = attr->get_size_in_bytes();
  const char* aRef = attr->aRef();

  if(aRef == NULL)
  {
    error= InvalidNullaRef;
    return;
  }

  switch (array_type) {
  case NdbDictionary::Column::ArrayTypeFixed:
    /* no length bytes stored with aRef */
    data_ptr = aRef;
    bytes = attr_bytes;
    break;
  case NdbDictionary::Column::ArrayTypeShortVar:
    /* first byte of aRef has length of the data */
    data_ptr = aRef + 1;
    bytes = (size_t)(aRef[0]);
    break;
  case NdbDictionary::Column::ArrayTypeMediumVar:
    /* first two bytes of aRef has length of the data */
    data_ptr = aRef + 2;
```

```
    bytes = (size_t)(aRef[1]) * 256 + (size_t)(aRef[0]);
    break;
  default:
    /* should never reach here */
    data_ptr = NULL;
    bytes = 0;
    error = InvalidArrayType;
    break;
  }
}


bool
ReadOnlyArrayAdapter::
is_binary_array_type(const NdbDictionary::Column::Type t) const
{
  bool is_binary;

  switch (t)
  {
  case NdbDictionary::Column::Binary:
  case NdbDictionary::Column::Varbinary:
  case NdbDictionary::Column::Longvarbinary:
    is_binary = true;
    break;
  default:
    is_binary = false;
  }
  return is_binary;
}


bool
ReadOnlyArrayAdapter::
is_array_type(const NdbDictionary::Column::Type t) const
{
  bool is_array;

  switch (t)
  {
  case NdbDictionary::Column::Binary:
  case NdbDictionary::Column::Varbinary:
  case NdbDictionary::Column::Longvarbinary:
  case NdbDictionary::Column::Char:
  case NdbDictionary::Column::Varchar:
  case NdbDictionary::Column::Longvarchar:
    is_array = true;
    break;
  default:
    is_array = false;
  }
  return is_array;
}

#endif // #ifndef ARRAY_ADAPTER_HPP
```

error_handling.hpp

```
#ifndef ERROR_HANDLING_HPP
#define ERROR_HANDLING_HPP

template <typename T>
inline static void print_if_not_equal(T got,
                                      T expected,
                                      const char* msg,
                                      const char* file,
                                      int line)
{
  std::cout << "Got value " << got << " instead of expected value " << expected
            << " in " << file << ":" << line;
}
```

```
#define PRINT_IF_NOT_EQUAL(got, expected, msg) {                        \
    if (got != expected) {                                              \
      print_if_not_equal(got, expected, msg, __FILE__, __LINE__);       \
      exit(-1);                                                         \
    }                                                                   \
  }

#define PRINT_ERROR(code,msg)                                           \
  std::cout << "Error in " << __FILE__ << ", line: " << __LINE__        \
            << ", code: " << code                                       \
            << ", msg: " << msg << "." << std::endl

#define APIERROR(error) {                \
  PRINT_ERROR(error.code,error.message); \
  exit(-1); }

#endif
```

ndb_util.hpp

```
#ifndef NDB_UTIL_HPP
#define NDB_UTIL_HPP

#include <NdbApi.hpp>
#include <string>
#include <sstream>

static const std::string column_type_to_string(NdbDictionary::Column::Type type)
{
  switch (type)
  {
  case NdbDictionary::Column::Undefined:
    return "Undefined";
  case NdbDictionary::Column::Tinyint:
    return "Tinyint";
  case NdbDictionary::Column::Tinyunsigned:
    return "Tinyunsigned";
  case NdbDictionary::Column::Smallint:
    return "Smallint";
  case NdbDictionary::Column::Smallunsigned:
    return "Smallunsigned";
  case NdbDictionary::Column::Mediumint:
    return "Mediumint";
  case NdbDictionary::Column::Mediumunsigned:
    return "Mediumunsigned";
  case NdbDictionary::Column::Int:
    return "Int";
  case NdbDictionary::Column::Unsigned:
    return "Unsigned";
  case NdbDictionary::Column::Bigint:
    return "Bigint";
  case NdbDictionary::Column::Bigunsigned:
    return "Bigunsigned";
  case NdbDictionary::Column::Float:
    return "Float";
  case NdbDictionary::Column::Double:
    return "Double";
  case NdbDictionary::Column::Olddecimal:
    return "Olddecimal";
  case NdbDictionary::Column::Olddecimalunsigned:
    return "Olddecimalunsigned";
  case NdbDictionary::Column::Decimal:
    return "Decimal";
  case NdbDictionary::Column::Decimalunsigned:
    return "Decimalunsigned";
  case NdbDictionary::Column::Char:
    return "Char";
  case NdbDictionary::Column::Varchar:
    return "Varchar";
  case NdbDictionary::Column::Binary:
```

```
    return "Binary";
  case NdbDictionary::Column::Varbinary:
    return "Varbinary";
  case NdbDictionary::Column::Datetime:
    return "Datetime";
  case NdbDictionary::Column::Date:
    return "Date";
  case NdbDictionary::Column::Blob:
    return "Blob";
  case NdbDictionary::Column::Text:
    return "Text";
  case NdbDictionary::Column::Bit:
    return "Bit";
  case NdbDictionary::Column::Longvarchar:
    return "Longvarchar";
  case NdbDictionary::Column::Longvarbinary:
    return "Longvarbinary";
  case NdbDictionary::Column::Time:
    return "Time";
  case NdbDictionary::Column::Year:
    return "Year";
  case NdbDictionary::Column::Timestamp:
    return "Timestamp";
  case NdbDictionary::Column::Time2:
    return "Time2";
  case NdbDictionary::Column::Datetime2:
    return "Datetime2";
  case NdbDictionary::Column::Timestamp2:
    return "Timestamp2";
  default:
    {
      std::string str;
      std::stringstream s(str);
      s << "Unknown type: " << type;
      return s.str();
    }
  }
}

#endif
```

util.hpp

```
#include <string>

/* Return a string containing 'n' copies of the string 's'. */
static std::string operator * (unsigned n, const std::string& s)
{
  std::string result;
  result.reserve(n * s.length());
  for (unsigned i = 0; i < n; i++)
  {
    result.append(s);
  }
  return result;
}

#endif // #ifndef UTIL_HPP
```

Chapter 3 The MGM API

Table of Contents

3.1 MGM API Concepts 483
3.1.1 Working with Log Events 484
3.1.2 Structured Log Events 484
3.2 MGM API Function Listing 485
3.2.1 Log Event Functions 485
3.2.2 MGM API Error Handling Functions 488
3.2.3 Management Server Handle Functions 490
3.2.4 Management Server Connection Functions 491
3.2.5 Cluster Status Functions 496
3.2.6 Functions for Starting & Stopping Nodes 498
3.2.7 Cluster Log Functions 503
3.2.8 Backup Functions 506
3.2.9 Single-User Mode Functions 507
3.3 MGM API Data Types 507
3.3.1 The ndb_mgm_node_type Type 508
3.3.2 The ndb_mgm_node_status Type 508
3.3.3 The ndb_mgm_error Type 508
3.3.4 The Ndb_logevent_type Type 508
3.3.5 The ndb_mgm_event_severity Type 513
3.3.6 The ndb_logevent_handle_error Type 513
3.3.7 The ndb_mgm_event_category Type 514
3.4 MGM API Structures 514
3.4.1 The ndb_logevent Structure 514
3.4.2 The ndb_mgm_node_state Structure 520
3.4.3 The ndb_mgm_cluster_state Structure 521
3.4.4 The ndb_mgm_reply Structure 521
3.5 MGM API Errors 521
3.5.1 Request Errors 522
3.5.2 Node ID Allocation Errors 522
3.5.3 Service Errors 522
3.5.4 Backup Errors 522
3.5.5 Single User Mode Errors 523
3.5.6 General Usage Errors 523
3.6 MGM API Examples 523
3.6.1 Basic MGM API Event Logging Example 523
3.6.2 MGM API Event Handling with Multiple Clusters 525

This chapter discusses the NDB Cluster Management API, a C language API that is used for administrative tasks such as starting and stopping Cluster nodes, backups, and logging. It also covers MGM API concepts, programming constructs, and event types.

3.1 MGM API Concepts

Each MGM API function needs a management server handle of type `NdbMgmHandle`. This handle is created by calling the function `ndb_mgm_create_handle()` and freed by calling `ndb_mgm_destroy_handle()`.

See Section 3.2.3.1, “ndb_mgm_create_handle()”, and Section 3.2.3.4, “ndb_mgm_destroy_handle()”, for more information about these two functions.

Important

You should not share an `NdbMgmHandle` between threads. While it is possible to do so (if you implement your own locks), this is not recommended; each thread should use its own management server handle.

A function can return any of the following:

- An integer value, with a value of `-1` indicating an error.
- A nonconstant pointer value. A `NULL` value indicates an error; otherwise, the return value must be freed by the programmer.
- A constant pointer value, with a `NULL` value indicating an error. The returned value should not be freed.

Error conditions can be identified by using the appropriate error-reporting functions `ndb_mgm_get_latest_error()` and `ndb_mgm_error()`.

Here is an example using the MGM API (without error handling for brevity's sake):

```
NdbMgmHandle handle= ndb_mgm_create_handle();
ndb_mgm_connect(handle,0,0,0);
struct ndb_mgm_cluster_state *state= ndb_mgm_get_status(handle);
for(int i=0; i < state->no_of_nodes; i++)
{
  struct ndb_mgm_node_state *node_state= &state->node_states[i];
  printf("node with ID=%d ", node_state->node_id);

  if(node_state->version != 0)
    printf("connected\n");
  else
    printf("not connected\n");
}
free((void*)state);
ndb_mgm_destroy_handle(&handle);
```

3.1.1 Working with Log Events

Data nodes and management servers regularly and on specific occasions report on various log events that occur in the cluster. These log events are written to the cluster log. Optionally an MGM API client may listen to these events using the method `ndb_mgm_listen_event()`. Each log event belongs to a category `ndb_mgm_event_category`) and has a severity `ndb_mgm_event_severity` associated with it. Each log event also has a level (0-15) associated with it.

Which log events that come out is controlled with `ndb_mgm_listen_event()`, `ndb_mgm_set_clusterlog_loglevel()`, and `ndb_mgm_set_clusterlog_severity_filter()`.

This is an example showing how to listen to events related to backup:

```
int filter[] = { 15, NDB_MGM_EVENT_CATEGORY_BACKUP, 0 };
int fd = ndb_mgm_listen_event(handle, filter);
```

3.1.2 Structured Log Events

The following steps are involved:

1. Create an `NdbLogEventHandle` using `ndb_mgm_create_logevent_handle()`.
2. Wait for and store log events using `ndb_logevent_get_next()`.

3. The log event data is available in the structure `ndb_logevent`. The data which is specific to a particular event is stored in a union between structures; use `ndb_logevent::type` to decide which structure is valid.

The following sample code demonstrates listening to events related to backups:

```
int filter[] = { 15, NDB_MGM_EVENT_CATEGORY_BACKUP, 0 };
NdbLogEventHandle le_handle= ndb_mgm_create_logevent_handle(handle, filter);
struct ndb_logevent le;
int r= ndb_logevent_get_next(le_handle, &le, 0);
if(r < 0)
  /*  error  */
else if(r == 0)
  /*  no event  */

switch(le.type)
{
  case NDB_LE_BackupStarted:
    ... le.BackupStarted.starting_node;
    ... le.BackupStarted.backup_id;
    break;
  case NDB_LE_BackupFailedToStart:
    ... le.BackupFailedToStart.error;
    break;
  case NDB_LE_BackupCompleted:
    ... le.BackupCompleted.stop_gci;
    break;
  case NDB_LE_BackupAborted:
    ... le.BackupStarted.backup_id;
    break;
  default:
    break;
}
```

For more information, see Section 3.2.1, “Log Event Functions”.

Available log event types are listed in Section 3.3.4, “The Ndb_logevent_type Type”, as well as in the file `/storage/ndb/include/mgmapi/ndb_logevent.h` in the NDB Cluster sources.

3.2 MGM API Function Listing

This section covers the structures and functions used in the MGM API. Listings are grouped by purpose or use.

3.2.1 Log Event Functions

This section discusses functions that are used for listening to log events.

3.2.1.1 ndb_mgm_listen_event()

Description. This function is used to listen to log events, which are read from the return file descriptor. Events use a text-based format, the same as in the cluster log.

Signature.

```
int ndb_mgm_listen_event
    (
      NdbMgmHandle handle,
      const int    filter[]
    )
```

Parameters. This function takes two arguments:

- An `NdbMgmHandle` *`handle`*.
- A *`filter`* which consists of a series of `{level, ndb_mgm_event_category}` pairs (in a single array) that are pushed to a file descriptor. Use `0` for the level to terminate the list.

Return value. The file descriptor from which events are to be read.

3.2.1.2 ndb_mgm_create_logevent_handle()

Description. This function is used to create a log event handle.

Signature.

```
NdbLogEventHandle ndb_mgm_create_logevent_handle
    (
      NdbMgmHandle handle,
      const int    filter[]
    )
```

Parameters. This function takes two arguments:

- An `NdbMgmHandle` *`handle`*.
- A *`filter`* which consists of a series of `{level, ndb_mgm_event_category}` pairs (in a single array) that are pushed to a file descriptor. Use `0` for the level to terminate the list.

Return value. A log event handle.

3.2.1.3 ndb_mgm_destroy_logevent_handle()

Description. Use this function to destroy a log event handle when there is no further need for it.

Signature.

```
void ndb_mgm_destroy_logevent_handle
    (
      NdbLogEventHandle* handle
    )
```

Parameters. A pointer to a log event *`handle`*.

Return value. *None*.

3.2.1.4 ndb_logevent_get_fd()

Description. This function retrieves a file descriptor from an `NdbMgmLogEventHandle`. It was implemented in MySQL 5.1.12.

Warning

Do not attempt to read from the file descriptor returned by this function.

Signature.

```
int ndb_logevent_get_fd
    (
      const NdbLogEventHandle handle
    )
```

Parameters. A `LogEventHandle`.

Return value. A file descriptor. In the event of failure, `-1` is returned.

3.2.1.5 ndb_logevent_get_next()

Description. This function is used to retrieve the next log event, using data from the event to fill in the supplied `ndb_logevent` structure.

Signature.

```
int ndb_logevent_get_next
    (
      const NdbLogEventHandle handle,
      struct ndb_logevent*    logevent,
      unsigned                timeout
    )
```

Important

Prior to NDB 7.2.14 and NDB 7.3.2, the log event's `ndb_mgm_event_category` was cast to an `enum` type. This behavior, although incorrect, interefered with existing applications and was reinstated in NDB 7.2.18 and NDB 7.3.7; a new function exhibiting the corrected behavior `ndb_logevent_get_next2()` was added in these releases.

Parameters. Three parameters are expected by this function:

- An `NdbLogEventHandle`
- A pointer to an `ndb_logevent` data structure
- The number of milliseconds to wait for the event before timing out; passing `0` for this parameter causes the function to block until the next log event is received

Return value. The value returned by this function is interpreted as follows: If the return value is less than or equal to zero, then the *`logevent`* is not altered or affected in any way.

- `> 0`: The event exists, and it data was retrieved into the *`logevent`*
- `0`: A timeout occurred while waiting for the event (more than *`timeout`* milliseconds elapsed)
- `< 0`: An error occurred.

3.2.1.6 ndb_logevent_get_next2()

Description. This function is used to retrieve the next log event, using data from the event to fill in the supplied `ndb_logevent` structure.

`ndb_logevent_get_next2()` was added in NDB 7.2.18 and NDB 7.3.7. It is intended to serve as a replacement for `ndb_logevent_get_next()` which corrects that function's handling of the structure's `ndb_mgm_event_category`, for applications which do not require backward compatibility. It is otherwise identical to `ndb_logevent_get_next()`.

Signature.

```
int ndb_logevent_get_next2
    (
      const NdbLogEventHandle handle,
      struct ndb_logevent*    logevent,
      unsigned                timeout
    )
```

Parameters. Three parameters are expected by this function:

- An `NdbLogEventHandle`
- A pointer to an `ndb_logevent` data structure
- The number of milliseconds to wait for the event before timing out; passing `0` for this parameter causes the function to block until the next log event is received

Return value. The value returned by this function is interpreted as follows: If the return value is less than or equal to zero, then the *`logevent`* is not altered or affected in any way.

- `> 0`: The event exists, and it data was retrieved into the *`logevent`*
- `0`: A timeout occurred while waiting for the event (more than *`timeout`* milliseconds elapsed)
- `< 0`: An error occurred.

3.2.1.7 ndb_logevent_get_latest_error()

Description. This function retrieves the error code from the most recent error.

Note

You may prefer to use `ndb_logevent_get_latest_error_msg()` instead. See Section 3.2.1.8, "ndb_logevent_get_latest_error_msg()"

Signature.

```
int ndb_logevent_get_latest_error
    (
      const NdbLogEventHandle handle
    )
```

Parameters. A log event handle.

Return value. An error code.

3.2.1.8 ndb_logevent_get_latest_error_msg()

Description. Retrieves the text of the most recent error obtained while trying to read log events.

Signature.

```
const char* ndb_logevent_get_latest_error_msg
    (
      const NdbLogEventHandle handle
    )
```

Parameters. A log event handle.

Return value. The text of the error message.

3.2.2 MGM API Error Handling Functions

The MGM API functions used for error handling are discussed in this section.

Each MGM API error is characterised by an error code and an error message. There may also be an error description that may provide additional information about the error. The API provides functions to obtain this information in the event of an error.

3.2.2.1 ndb_mgm_get_latest_error()

Description. This function is used to get the latest error code associated with a given management server handle.

Prior to NDB 7.4.8, this function was not safe for use with `NULL`. In later versions, `ndb_mgm_get_latest_error()` is null-safe but returns an arbitrary value. (Bug #78130, Bug #21651706)

Signature.

```
int ndb_mgm_get_latest_error
    (
      const NdbMgmHandle handle
    )
```

Parameters. An `NdbMgMHandle`.

Return value. An error code corresponding to an `ndb_mgm_error` value. You can obtain the related error message using `ndb_mgm_get_latest_error_msg()`.

3.2.2.2 ndb_mgm_get_latest_error_msg()

Description. This function is used to obtain the latest general error message associated with an `NdbMgmHandle`.

Prior to NDB 7.4.8, this function was not safe for use with `NULL`. In later versions, `ndb_mgm_get_latest_error_msg()` is null-safe but returns an arbitrary value. (Bug #78130, Bug #21651706)

Signature.

```
const char* ndb_mgm_get_latest_error_msg
    (
      const NdbMgmHandle handle
    )
```

Parameters. An `NdbMgmHandle`.

Return value. The error message text. More specific information can be obtained using `ndb_mgm_get_latest_error_desc()`-

3.2.2.3 ndb_mgm_get_latest_error_desc()

Description. Get the most recent error description associated with an `NdbMgmHandle`; this description provides additional information regarding the error message.

Prior to NDB 7.4.8, this function was not safe for use with `NULL`. In later versions, `ndb_mgm_get_latest_error_desc()` is null-safe but returns an arbitrary value. (Bug #78130, Bug #21651706)

Signature.

```
const char* ndb_mgm_get_latest_error_desc
    (
      const NdbMgmHandle handle
    )
```

Parameters. An `NdbMgmHandle`.

Return value. The error description text.

3.2.2.4 ndb_mgm_set_error_stream()

Description. The function can be used to set the error output stream.

Signature.

```
void ndb_mgm_set_error_stream
    (
      NdbMgmHandle handle,
      FILE*        file
    )
```

Parameters. This function requires two parameters:

- An `NdbMgmHandle`
- A pointer to the file to which errors are to be sent.

Return value. *None*.

3.2.3 Management Server Handle Functions

This section contains information about the MGM API functions used to create and destroy management server handles.

3.2.3.1 ndb_mgm_create_handle()

Description. This function is used to create a handle to a management server.

Signature.

```
NdbMgmHandle ndb_mgm_create_handle
    (
      void
    )
```

Parameters. *None*.

Return value. An `NdbMgmHandle`.

3.2.3.2 ndb_mgm_set_name()

Description. This function can be used to set a name for the management server handle, which is then reported in the Cluster log.

Signature.

```
void ndb_mgm_set_name
    (
      NdbMgmHandle handle,
      const char*  name
    )
```

Parameters. This function takes two arguments:

- A management server *`handle`*.
- The desired *`name`* for the *`handle`*.

Return value. *None*.

3.2.3.3 ndb_mgm_set_ignore_sigpipe()

Description. The MGM API by default installs a signal handler that ignores all `SIGPIPE` signals that might occur when writing to asocket that has been closed or reset. An application that provides its own handler for `SIGPIPE` should call this function after creating the management server handle and before using the handle to connect to the management server. (In other words, call this function after using `ndb_mgm_create_handle()` but before calling `ndb_mgm_connect()`, which causes the MGM API's `SIGPIPE` handler to be installed unless overridden.)

Signature.

```
int ndb_mgm_set_ignore_sigpipe
    (
      NdbMgmHandle handle,
      int ignore = 1
    )
```

Parameters. This function takes two parameters:

- A management server handle
- An integer value which determines whether to *`ignore`* `SIGPIPE` errors. Set this to 1 (the default) to cause the MGM API to ignore `SIGPIPE`; set to zero if you wish for `SIGPIPE` to propagate to your MGM API application.

Return value. *None*.

3.2.3.4 ndb_mgm_destroy_handle()

Description. This function destroys a management server handle

Signature.

```
void ndb_mgm_destroy_handle
    (
      NdbMgmHandle* handle
    )
```

Parameters. A pointer to the `NdbMgmHandle` to be destroyed.

Return value. *None*.

3.2.4 Management Server Connection Functions

This section discusses MGM API functions that are used to initiate, configure, and terminate connections to an `NDB` management server.

3.2.4.1 ndb_mgm_get_connectstring()

Description. This function retrieves the connection string used for a connection.

Note

This function returns the default connection string if no call to `ndb_mgm_set_connectstring()` has been performed. In addition, the returned connection string may be formatted slightly differently than the original in that it may contain specifiers not present in the original.

The connection string format is the same as that discussed for Section 3.2.4.10, “ndb_mgm_set_connectstring()”.

Signature.

```
const char* ndb_mgm_get_connectstring
    (
      NdbMgmHandle handle,
      char*        buffer,
      int          size
    )
```

Parameters. This function takes three arguments:

- An `NdbMgmHandle`.
- A pointer to a *`buffer`* in which to place the result.
- The *`size`* of the buffer.

Return value. The connection string—this is the same value that is pushed to the *`buffer`*.

3.2.4.2 ndb_mgm_get_configuration_nodeid()

Description. This function gets the ID of the node to which the connection is being (or was) made.

Signature.

```
int ndb_mgm_get_configuration_nodeid
    (
      NdbMgmHandle handle
    )
```

Parameters. A management server handle.

Return value. A node ID.

3.2.4.3 ndb_mgm_get_connected_port()

Description. This function retrieves the number of the port used by the connection.

Signature.

```
int ndb_mgm_get_connected_port
    (
      NdbMgmHandle handle
    )
```

Parameters. An `NdbMgmHandle`.

Return value. A port number.

3.2.4.4 ndb_mgm_get_connected_host()

Description. This function is used to obtain the name of the host to which the connection is made.

Signature.

```
const char* ndb_mgm_get_connected_host
    (
      NdbMgmHandle handle
    )
```

Parameters. A management server *`handle`*.

Return value. A host name.

3.2.4.5 ndb_mgm_get_version()

Description. Given a management server handle, this function gets `NDB` engine and MySQL Server version information for the indicated management server.

Signature.

```
int ndb_mgm_get_version
    (
      NdbMgmHandle handle,
      int* major,
      int* minor,
      int* build,
      int length,
      char* string
    )
```

Parameters. An `NdbMgmHandle`, and pointers to the `NDB` engine *`major`*, *`minor`*, and *`build`* version values, as well as a pointer to the version *`string`* (along with the strength's *`length`*).

The version string uses the format `mysql-x.x.x ndb-y.y.y-status`, where `x.x.x` is the three-part MySQL Server version, and `y.y.y` is the three-part NDB storage engine version. The `status` string indicates the release level or status; usually this is one of `beta`, `rc`, or `ga`, but other values are sometimes possible.

Return value. `ndb_mgm_get_version()` returns an integer: 0 on success; any nonzero value indicates an error.

3.2.4.6 ndb_mgm_is_connected()

Description. Used to determine whether a connection has been established.

Note

This function does not determine whether or not there is a “live” management server at the other end of the connection. Beginning with MySQL 5.1.17, you can use `ndb_mgm_check_connection()` to accomplish that task.

Signature.

```
int ndb_mgm_is_connected
    (
      NdbMgmHandle handle
    )
```

Parameters. A management server *`handle`*.

Return value. This function returns an integer, whose value is interpreted as follows:

- `0`: Not connected to the management node.
- Any nonzero value: A connection has been established with the management node.

3.2.4.7 ndb_mgm_check_connection()

Description. This function can be used to determine whether a management server is running on a given connection from a management client.

Prior to MySQL 5.1.17, this function was available but required extremely large timeouts to be configured for it to be effective.

Signature.

```
int ndb_mgm_check_connection
    (
      NdbMgmHandle handle
    )
```

Parameters. An `NdbMgmHandle` (see Section 3.1, “MGM API Concepts”).

Return value. In NDB 7.5 and later, this function returns 0 on success, -1 when the handle is null, and -2 when not connected.

In NDB 7.4 and earlier, this function returned -1 in the event of an error; otherwise it returned 0, even when the management server handle was NULL, or when the connection check failed (Bug #53242, Bug #11760802).

3.2.4.8 ndb_mgm_number_of_mgmd_in_connect_string()

Description. This is a convenience function which provides an easy way to determine the number of management servers referenced in a connection string as set using `ndb_mgm_set_connectstring()`.

This function was added in MySQL 5.1.18.

Signature.

```
int ndb_mgm_number_of_mgmd_in_connect_string
    (
      NdbMgmHandle handle
    )
```

Parameters. A management handle (`NdbMgmHandle`).

Return value. On success, a nonnegative integer; a negative integer indicates failure.

3.2.4.9 ndb_mgm_set_bindaddress()

Description. This function makes it possible to set a local bind address for the management server. If used, it must be called before connecting to the management server.

This function was added in MySQL 5.1.18.

Signature.

```
int ndb_mgm_set_bindaddress
    (
      NdbMgmHandle handle,
      const char*  address
    )
```

Parameters. This function takes two parameters:

- A management handle (`NdbMgmHandle`).
- A string *`address`* of the form *`host`*`[:`*`port`*`]`.

Return value. Returns an integer:

- `0` indicates success
- Any nonzero value indicates failure (the address was not valid)

Important

Errors caused by binding an otherwise valid local address are not reported until the connection to the management is actually attempted.

3.2.4.10 ndb_mgm_set_connectstring()

Description. This function is used to set the connection string for a management server connection to a node.

Signature.

```
int ndb_mgm_set_connectstring
    (
      NdbMgmHandle handle,
      const char*  connection_string
    )
```

Parameters. `ndb_mgm_set_connectstring()` takes two parameters:

- A management server *`handle`*.
- A *`connection_string`* whose format is shown here:

```
connection_string :=
    [nodeid-specification,]host-specification[,host-specification]
```

`ndb_mgm_get_connectstring()` also uses this format for connection strings.

It is possible to establish connections with multiple management servers using a single connection string.

```
nodeid-specification := nodeid=id
host-specification := host[:port]
```

`id`, *`port`*, and *`host`* are defined as follows:

- *`id`*: An integer greater than `0` identifying a node in `config.ini`.
- *`port`*: An integer referring to a standard Unix port.
- *`host`*: A string containing a valid network host address.

Return value. This function returns `-1` in the event of failure.

3.2.4.11 ndb_mgm_set_configuration_nodeid()

Description. This function sets the connection node ID.

Signature.

```
int ndb_mgm_set_configuration_nodeid
    (
      NdbMgmHandle handle,
      int          id
    )
```

Parameters. This function requires two parameters:

- An `NdbMgmHandle`.
- The `id` of the node to connect to.

Return value. This function returns `-1` in the event of failure.

3.2.4.12 ndb_mgm_set_timeout()

Description. Normally, network operations time out after 60 seconds. This function permits you to vary this time.

Important

The timeout set by this function applies not only to establishing network connections, but to *every* operation requiring communication using a network connection. This includes each network read or write performed by any MGM API function, NDB API method call, or `ndb_mgm` client command.

This function was introduced in MySQL 5.1.18.

Signature.

```
int ndb_mgm_set_timeout
    (
      NdbMgmHandle handle,
      unsigned int timeout
```

```
    )
```

Parameters. This function takes two parameters:

- A management server handle (`NdbMgmHandle`).
- An amount of time to wait before timing out, expressed in milliseconds.

Return value. Returns `0` on success, with any other value representing failure.

3.2.4.13 ndb_mgm_connect()

Description. This function establishes a connection to a management server specified by the connection string set by Section 3.2.4.10, “ndb_mgm_set_connectstring()”.

Signature.

```
int ndb_mgm_connect
    (
      NdbMgmHandle handle,
      int          retries,
      int          delay,
      int          verbose
    )
```

Parameters. This function takes 4 arguments:

- A management server *`handle`*.
- The number of `retries` to make when attempting to connect. `0` for this value means that one connection attempt is made.
- The number of seconds to *`delay`* between connection attempts.
- If *`verbose`* is `1`, then a message is printed for each connection attempt.

Return value. This function returns `-1` in the event of failure.

3.2.4.14 ndb_mgm_disconnect()

Description. This function terminates a management server connection.

Signature.

```
int ndb_mgm_disconnect
    (
      NdbMgmHandle handle
    )
```

Parameters. An `NdbMgmHandle`.

Return value. Returns `-1` if unable to disconnect.

3.2.5 Cluster Status Functions

This section discusses how to obtain status information from NDB Cluster nodes.

3.2.5.1 ndb_mgm_get_status()

Description. This function is used to obtain the status of the nodes in an NDB Cluster.

Note

The caller must free the pointer returned by this function.

Signature.

```
struct ndb_mgm_cluster_state* ndb_mgm_get_status
    (
      NdbMgmHandle handle
    )
```

Parameters. This function takes a single parameter, a management server *handle*.

Return value. A pointer to an ndb_mgm_cluster_state data structure.

3.2.5.2 ndb_mgm_get_status2()

Description. This function is similar to ndb_mgm_get_status(), in that it is used to obtain the status of the nodes in an NDB Cluster. However, ndb_mgm_get_status2() allows one to specify the type or types of nodes (ndb_mgm_node_type) to be checked.

Note

The caller must free the pointer returned by this function.

Signature.

```
struct ndb_mgm_cluster_state* ndb_mgm_get_status2
    (
      NdbMgmHandle handle,
      const enum ndb_mgm_node_type types[]
    )
```

Parameters. This function takes two parameters:

- A management server *handle*
- A pointer to array of the node types to be checked. These are ndb_mgm_node_type values. The array should be terminated by an element of type NDB_MGM_NODE_TYPE_UNKNOWN.

Return value. A pointer to an ndb_mgm_cluster_state data structure.

3.2.5.3 ndb_mgm_dump_state()

Description. This function can be used to dump debugging information to the cluster log. The NDB Cluster management client DUMP command is a wrapper for this function.

Important

ndb_mgm_dump_state(), like the DUMP command, can cause a running NDB Cluster to malfunction or even to fail completely if it is used improperly. Be sure to consult the relevant documentation before using this function. For more information on the DUMP command, and for a listing of current DUMP codes and their effects, see NDB Cluster Management Client DUMP Commands.

Signature.

```
int ndb_mgm_dump_state
    (
      NdbMgmHandle handle,
      int nodeId,
      const int* arguments,
      int numberOfArguments,
      struct ndb_mgm_reply* reply
    )
```

Parameters. This function takes the following pararemeters:

- A management server handle (`NdbMgmHandle`)
- The *`nodeId`* of a cluster data node.
- An array of *`arguments`*. The first of these is the `DUMP` code to be executed. Subsequent arguments can be passed in this array if needed by or desired for the corresponding `DUMP` command.
- The *`numberOfArguments`* to be passed.
- An `ndb_mgm_reply`, which contains a return code along with a response or error message.

Return value. `0` on success; otherwise, an error code.

Example. The following example has the same result as running `2 DUMP 1000` in the management client:

```
//  [...]
#include <mgmapi_debug.h>
//  [...]
struct ndb_mgm_reply reply;
int args[1];
int stat, arg_count, node_id;

args[0] = 1000;
arg_count = 1;
node_id = 2;

stat = ndb_mgm_dump_state(h, node_id, args, arg_count, &reply);
```

3.2.6 Functions for Starting & Stopping Nodes

The MGM API provides several functions which can be used to start, stop, and restart one or more Cluster data nodes. These functions are discussed in this section.

Starting, Stopping, and Restarting Nodes. You can start, stop, and restart Cluster nodes using the following functions, which are described in more detail in the next few sections.

- **Starting Nodes.** Use `ndb_mgm_start()`.
- **Stopping Nodes.** Use `ndb_mgm_stop()`, `ndb_mgm_stop2()`, `ndb_mgm_stop3()`, or `ndb_mgm_stop4()`.

  Normally, you cannot use any of these functions to stop a node while other nodes are starting. You can override this restriction using `ndb_mgm_stop4()` with the *`force`* parameter set to 1.
- **Restarting Nodes.** Use `ndb_mgm_restart()`, `ndb_mgm_restart2()`, `ndb_mgm_restart3()`, or `ndb_mgm_restart4()`.

  Normally, you cannot use any of these functions to restart a node while other nodes are starting. You can override this restriction using `ndb_mgm_restart4()` with the *`force`* parameter set to 1.

3.2.6.1 ndb_mgm_start()

Description. This function can be used to start one or more Cluster nodes. The nodes to be started must have been started with the no-start option (`-n`), meaning that the data node binary was started and is waiting for a `START` management command which actually enables the node.

Signature.

```
int ndb_mgm_start
    (
      NdbMgmHandle handle,
      int          number,
      const int*   list
```

```
    )
```

Parameters. `ndb_mgm_start()` takes 3 parameters:

- An `NdbMgmHandle`.
- A *`number`* of nodes to be started. Use `0` to start all of the data nodes in the cluster.
- A *`list`* of the node IDs of the nodes to be started.

Return value. The number of nodes actually started; in the event of failure, `-1` is returned.

3.2.6.2 ndb_mgm_stop()

Description. This function stops one or more data nodes.

Signature.

```
int ndb_mgm_stop
    (
      NdbMgmHandle handle,
      int          number,
      const int*   list
    )
```

Parameters. `ndb_mgm_stop()` takes 3 parameters: Calling this function is equivalent to calling `ndb_mgm_stop2(handle, number, list, 0)`.

- An `NdbMgmHandle`.
- The *`number`* of nodes to be stopped. Use `0` to stop all of the data nodes in the cluster.
- A *`list`* of the node IDs of the nodes to be stopped.

Return value. The number of nodes actually stopped; in the event of failure, `-1` is returned.

3.2.6.3 ndb_mgm_stop2()

Description. Like `ndb_mgm_stop()`, this function stops one or more data nodes. However, it offers the ability to specify whether or not the nodes shut down gracefully.

Signature.

```
int ndb_mgm_stop2
    (
      NdbMgmHandle handle,
      int          number,
      const int*   list,
      int          abort
    )
```

Parameters. `ndb_mgm_stop2()` takes 4 parameters:

- An `NdbMgmHandle`.
- The *`number`* of nodes to be stopped. Use `0` to stop all of the data nodes in the cluster.
- A *`list`* of the node IDs of the nodes to be stopped.
- The value of *`abort`* determines how the nodes will be shut down. `1` indicates the nodes will shut down immediately; `0` indicates that the nodes will stop gracefully.

Return value. The number of nodes actually stopped; in the event of failure, `-1` is returned.

3.2.6.4 ndb_mgm_stop3()

Description. Like `ndb_mgm_stop()` and `ndb_mgm_stop2()`, this function stops one or more data nodes. Like `ndb_mgm_stop2()`, it offers the ability to specify whether the nodes should shut down gracefully. In addition, it provides for a way to check to see whether disconnection is required prior to stopping a node.

Signature.

```
int ndb_mgm_stop3
    (
      NdbMgmHandle handle,
      int          number,
      const int*   list,
      int          abort,
      int*         disconnect
    )
```

Parameters. `ndb_mgm_stop3()` takes 5 parameters:

- An `NdbMgmHandle`.
- The *`number`* of nodes to be stopped. Use `0` to stop all of the data nodes in the cluster.
- A *`list`* of the node IDs of the nodes to be stopped.
- The value of *`abort`* determines how the nodes will be shut down. `1` indicates the nodes will shut down immediately; `0` indicates that the nodes will stop gracefully.
- If *`disconnect`* returns `1` (`true`), this means the you must disconnect before you can apply the command to stop. For example, disconnecting is required when stopping the management server to which the handle is connected.

Return value. The number of nodes actually stopped; in the event of failure, `-1` is returned.

3.2.6.5 ndb_mgm_stop4()

Description. Like the other `ndb_mgm_stop*()` functions, this function stops one or more data nodes. Like `ndb_mgm_stop2()`, it offers the ability to specify whether the nodes should shut down gracefully; like `ndb_mgm_stop3()` it provides for a way to check to see whether disconnection is required prior to stopping a node. In addition, it is possible to force the node to shut down even if this would cause the cluster to become nonviable.

Signature.

```
int ndb_mgm_stop4
    (
      NdbMgmHandle handle,
      int          number,
      const int*   list,
      int          abort,
      int          force,
      int*         disconnect
    )
```

Parameters. `ndb_mgm_stop4()` takes 6 parameters:

- An `NdbMgmHandle`.
- The *`number`* of nodes to be stopped. Use `0` to stop all of the data nodes in the cluster.
- A *`list`* of the node IDs of the nodes to be stopped.
- The value of *`abort`* determines how the nodes will be shut down. `1` indicates the nodes will shut down immediately; `0` indicates that the nodes will stop gracefully.

- The value of *force* determines the action to be taken in the event that the shutdown of a given node would cause an incomplete cluster. 1 causes the node—and the entire cluster—to be shut down in such cases, 0 means the node will not be shut down.

  Setting *force* equal to 1 also makes it possible to stop a node even while other nodes are starting. (Bug #58451)

- If *disconnect* returns 1 (true), this means the you must disconnect before you can apply the command to stop. For example, disconnecting is required when stopping the management server to which the handle is connected.

Return value. The number of nodes actually stopped; in the event of failure, -1 is returned.

3.2.6.6 ndb_mgm_restart()

Description. This function can be used to restart one or more Cluster data nodes.

Signature.

```
int ndb_mgm_restart
    (
      NdbMgmHandle handle,
      int          number,
      const int*   list
    )
```

Parameters. ndb_mgm_restart() takes 3 parameters:

- An NdbMgmHandle.
- The *number* of nodes to be stopped. Use 0 to stop all of the data nodes in the cluster.
- A *list* of the node IDs of the nodes to be stopped.

Calling this function is equivalent to calling

```
ndb_mgm_restart2(handle, number, list, 0, 0, 0);
```

See Section 3.2.6.7, “ndb_mgm_restart2()”, for more information.

Return value. The number of nodes actually restarted; -1 on failure.

3.2.6.7 ndb_mgm_restart2()

Description. Like ndb_mgm_restart(), this function can be used to restart one or more Cluster data nodes. However, ndb_mgm_restart2() provides additional restart options, including initial restart, waiting start, and immediate (forced) restart.

Signature.

```
int ndb_mgm_restart2
    (
      NdbMgmHandle handle,
      int          number,
      const int*   list,
      int          initial
      int          nostart,
      int          abort
    )
```

Parameters. ndb_mgm_restart2() takes 6 parameters:

- An NdbMgmHandle.

- The *number* of nodes to be stopped. Use 0 to stop all of the data nodes in the cluster.
- A *list* of the node IDs of the nodes to be stopped.
- If *initial* is true (1), then each node undergoes an initial restart—that is, its file system is removed.
- If nostart is true, then the nodes are not actually started, but instead are left ready for a start command.
- If *abort* is true, then the nodes are restarted immediately, bypassing any graceful restart.

Return value. The number of nodes actually restarted; -1 on failure.

3.2.6.8 ndb_mgm_restart3()

Description. Like ndb_mgm_restart2(), this function can be used to cause an initial restart, waiting restart, and immediate (forced) restart on one or more Cluster data nodes. However, ndb_mgm_restart3() provides the additional options of checking whether disconnection is required prior to the restart.

Signature.

```
int ndb_mgm_restart3
    (
      NdbMgmHandle handle,
      int          number,
      const int*   list,
      int          initial
      int          nostart,
      int          abort,
      int*         disconnect
    )
```

Parameters. ndb_mgm_restart3() takes 7 parameters:

- An NdbMgmHandle.
- The *number* of nodes to be stopped. Use 0 to stop all of the data nodes in the cluster.
- A *list* of the node IDs of the nodes to be stopped.
- If *initial* is true (1), then each node undergoes an initial restart—that is, its file system is removed.
- If nostart is true, then the nodes are not actually started, but instead are left ready for a start command.
- If *abort* is true, then the nodes are forced to restart immediately without performing a graceful restart.
- If *disconnect* returns 1 (true), this means the you must disconnect before you can apply the command to restart. For example, disconnecting is required when stopping the management server to which the handle is connected.

Return value. The number of nodes actually restarted; -1 on failure.

3.2.6.9 ndb_mgm_restart4()

Description. Like the other ndb_mgm_restart*() functions, this function restarts one or more data nodes. Like ndb_mgm_restart2(), it can be used to cause an initial restart, waiting restart, and immediate (forced) restart on one or more NDB Cluster data nodes; like ndb_mgm_stop3() it provides for a way to check to see whether disconnection is required prior to stopping a node. In addition, it is possible to force the node to restart even if this would cause a restart of the cluster.

Signature.

```
int ndb_mgm_restart4
    (
      NdbMgmHandle handle,
      int          number,
      const int*   list,
      int          initial
      int          nostart,
      int          abort,
      int          force,
      int*         disconnect
    )
```

Parameters. `ndb_mgm_restart4()` takes 7 parameters:

- An `NdbMgmHandle`.
- The *`number`* of nodes to be stopped. Use `0` to stop all of the data nodes in the cluster.
- A *`list`* of the node IDs of the nodes to be stopped.
- If *`initial`* is true (`1`), then each node undergoes an initial restart—that is, its file system is removed.
- If `nostart` is true, then the nodes are not actually started, but instead are left ready for a start command.
- If *`abort`* is true, then the nodes are forced to restart immediately without performing a graceful restart.
- The value of *`force`* determines the action to be taken in the event that the loss of a given node due to restarting would cause an incomplete cluster.

  `1` causes the node—and the entire cluster—to be restarted in such cases, `0` means that the node will not be restarted.

  Setting *`force`* equal to 1 also makes it possible to restart a node even while other nodes are starting. (Bug #58451)
- If *`disconnect`* returns `1` (`true`), this means the you must disconnect before you can apply the command to restart. For example, disconnecting is required when stopping the management server to which the handle is connected.

Return value. The number of nodes actually restarted; `-1` on failure.

3.2.7 Cluster Log Functions

This section covers the functions available in the MGM API for controlling the output of the cluster log.

3.2.7.1 ndb_mgm_get_clusterlog_severity_filter()

Description. This function is used to retrieve the cluster log severity filter currently in force.

Important

The parameters and return type of this function changed significantly between MySQL 5.1.13 and 5.1.14. The changes are detailed in the *Signature*, *Parameters*, and *Return Type* sections that follow.

These changes were done in order to make this function thread-safe. The pre-5.1.14 version is still supported for backward compatibility, but you should protect it with a mutex if you intend to call it from more than one thread.

Signature. As of MySQL 5.1.14:

```
int ndb_mgm_get_clusterlog_severity_filter
    (
      NdbMgmHandle handle,
      struct ndb_mgm_severity* severity,
      unsigned int size
    )
```

In MySQL 5.1.13 and earlier, this function took only a single parameter, as shown here:

```
const unsigned int* ndb_mgm_get_clusterlog_severity_filter
    (
      NdbMgmHandle handle
    )
```

Parameters. This function added two new parameters in MySQL 5.1.14.

- *All MySQL 5.1 releases*:

  An `NdbMgmHandle`.

- *Additionally, in MySQL 5.1.14 and later*:

  - A vector *`severity`* of seven (`NDB_MGM_EVENT_SEVERITY_ALL`) elements, each of which is an `ndb_mgm_severity` structure, where each element contains `1` if a severity indicator is enabled and `0` if not. A severity level is stored at position *`ndb_mgm_clusterlog_level`*; for example the error level is stored at position `NDB_MGM_EVENT_SEVERITY_ERROR`. The first element (position `NDB_MGM_EVENT_SEVERITY_ON`) in the vector signals whether the cluster log is disabled or enabled.

  - The *`size`* of the vector (`NDB_MGM_EVENT_SEVERITY_ALL`).

Return value. This function's return type changed beginning with MySQL 5.1.14.

- *MySQL 5.1.13 and earlier*: A *severity filter*, which is a vector containing 7 elements. Each element equals `1` if the corresponding severity indicator is enabled, and `0` if it is not. A severity level is stored at position *`ndb_mgm_clusterlog_level`*—for example, the “error” level is stored at position `NDB_MGM_EVENT_SEVERITY_ERROR`. The first element in the vector (`NDB_MGM_EVENT_SEVERITY_ON`) signals whether the cluster log is enabled or disabled.

- *MySQL 5.1.14 and later*: The number of returned severities, or `-1` in the event of an error.

3.2.7.2 ndb_mgm_set_clusterlog_severity_filter()

Description. This function is used to set a cluster log severity filter.

Signature.

```
int ndb_mgm_set_clusterlog_severity_filter
    (
      NdbMgmHandle                handle,
      enum ndb_mgm_event_severity severity,
      int                         enable,
      struct ndb_mgm_reply*       reply
    )
```

Parameters. This function takes 4 parameters:

- A management server *`handle`*.

- A cluster log *`severity`* to filter.

- A flag to *`enable`* or disable the filter; `1` enables and `0` disables the filter.
- A pointer to an `ndb_mgm_reply` structure for a reply message.

Return value. The function returns `-1` in the event of failure.

3.2.7.3 ndb_mgm_get_clusterlog_loglevel()

Description. This function, added in MySQL 5.1.16, is used to obtain log category and level information. It replaces the older `ndb_mgm_get_loglevel_clusterlog()` function, which performed the same purpose, but was not thread-safe. (See later in this section for a brief description of the deprecated version of the function.)

Signature.

```
int ndb_mgm_get_clusterlog_loglevel
    (
      NdbMgmHandle handle,
      struct ndb_mgm_loglevel* loglevel,
      unsigned int size
    )
```

Parameters. `ndb_mgm_get_clusterlog_loglevel()` takes the following parameters:

- A management *`handle`* (`NdbMgmHandle`).
- A *`loglevel`* (log level) vector consisting of twelve elements, each of which is an `ndb_mgm_loglevel` structure and which represents a log level of the corresponding category.
- The *`size`* of the vector (`MGM_LOGLEVELS`).

Return value. This function returns the number of returned loglevels or `-1` in the event of an error.

Note

Prior to MySQL 5.1.14, this function was known as `ndb_mgm_get_loglevel_clusterlog()`, and had the following signature:

```
const unsigned int* ndb_mgm_get_loglevel_clusterlog
    (
      NdbMgmHandle handle
    )
```

This version of the function is now deprecated, but is still available for backward compatibility; however, in new applications, it is recommended that you use `ndb_mgm_get_clusterlog_loglevel()`, since it is thread-safe, and the older function is not.

3.2.7.4 ndb_mgm_set_clusterlog_loglevel()

Description. This function is used to set the log category and levels for the cluster log.

Signature.

```
int ndb_mgm_set_clusterlog_loglevel
    (
      NdbMgmHandle               handle,
      int                        id,
      enum ndb_mgm_event_category category,
      int                        level,
      struct ndb_mgm_reply*      reply)
```

Parameters. This function takes 5 parameters:

- An `NdbMgmHandle`.
- The *`id`* of the node affected.
- An event *`category`*mdash;this is one of the values listed in Section 3.3.7, “The ndb_mgm_event_category Type” .
- A logging *`level`*.
- A pointer to an `ndb_mgm_reply` structure for the *`reply`* message.

Return value. In the event of an error, this function returns `-1`.

3.2.8 Backup Functions

This section covers the functions provided in the MGM API for starting and stopping backups.

3.2.8.1 ndb_mgm_start_backup()

Description. This function is used to initiate a backup of an NDB Cluster.

Signature.

```
int ndb_mgm_start_backup
    (
      NdbMgmHandle          handle,
      int                   wait,
      unsigned int*         id,
      struct ndb_mgm_reply* reply
    )
```

Parameters. This function requires 4 parameters:

- A management server *`handle`* (an `NdbMgmHandle`).
- A *`wait`* flag, with the following possible values:
  - `0`: Do not wait for confirmation of the backup.
  - `1`: Wait for the backup to be started.
  - `2`: Wait for the backup to be completed.
- A backup *`id`* to be returned by the function.

Note

No backup *`id`* is returned if *`wait`* is set equal to 0.

- A pointer to an `ndb_mgm_reply` structure to accommodate a *`reply`*.

Return value. In the event of failure, the function returns `-1`.

3.2.8.2 ndb_mgm_abort_backup()

Description. This function is used to stop a Cluster backup.

Signature.

```
int ndb_mgm_abort_backup
    (
      NdbMgmHandle          handle,
      unsigned int          id,
```

```
      struct ndb_mgm_reply* reply)
```

Parameters. This function takes 3 parameters:

- An `NdbMgmHandle`.
- The *`id`* of the backup to be aborted.
- A pointer to an `ndb_mgm_reply` structure.

Return value. In case of an error, this function returns `-1`.

3.2.9 Single-User Mode Functions

The MGM API makes it possible for the programmer to put the cluster into single-user mode—and to return it to normal mode again—from within an application. This section covers the functions that are used for these operations.

3.2.9.1 ndb_mgm_enter_single_user()

Description. This function is used to enter single-user mode on a given node.

Signature.

```
int ndb_mgm_enter_single_user
    (
      NdbMgmHandle          handle,
      unsigned int          id,
      struct ndb_mgm_reply* reply
    )
```

Parameters. This function takes 3 parameters:

- An `NdbMgmHandle`.
- The *`id`* of the node to be used in single-user mode.
- A pointer to an `ndb_mgm_reply` structure, used for a *`reply`* message.

Return value. Returns `-1` in the event of failure.

3.2.9.2 ndb_mgm_exit_single_user()

Description. This function is used to exit single-user mode and to return to normal operation.

Signature.

```
int ndb_mgm_exit_single_user
    (
      NdbMgmHandle          handle,
      struct ndb_mgm_reply* reply
    )
```

Parameters. This function requires 2 arguments:

- An `NdbMgmHandle`.
- A pointer to an `ndb_mgm_reply`.

Return value. Returns `-1` in case of an error.

3.3 MGM API Data Types

This section discusses the data types defined by the MGM API.

Note

The types described in this section are all defined in the file `/storage/ndb/include/mgmapi/mgmapi.h`, with the exception of `Ndb_logevent_type`, `ndb_mgm_event_severity`, `ndb_mgm_logevent_handle_error`, and `ndb_mgm_event_category`, which are defined in `/storage/ndb/include/mgmapi/ndb_logevent.h`.

3.3.1 The ndb_mgm_node_type Type

Description. This is used to classify the different types of nodes in an NDB Cluster.

Enumeration values. Possible values are shown, along with descriptions, in the following table:

Value	Description
`NDB_MGM_NODE_TYPE_UNKNOWN`	Unknown
`NDB_MGM_NODE_TYPE_API`	API Node (SQL node)
`NDB_MGM_NODE_TYPE_NDB`	Data node
`NDB_MGM_NODE_TYPE_MGM`	Management node

3.3.2 The ndb_mgm_node_status Type

Description. This type describes a Cluster node's status.

Enumeration values. Possible values are shown, along with descriptions, in the following table:

Value	Description
`NDB_MGM_NODE_STATUS_UNKNOWN`	The node's status is not known
`NDB_MGM_NODE_STATUS_NO_CONTACT`	The node cannot be contacted
`NDB_MGM_NODE_STATUS_NOT_STARTED`	The node has not yet executed the startup protocol
`NDB_MGM_NODE_STATUS_STARTING`	The node is executing the startup protocol
`NDB_MGM_NODE_STATUS_STARTED`	The node is running
`NDB_MGM_NODE_STATUS_SHUTTING_DOWN`	The node is shutting down
`NDB_MGM_NODE_STATUS_RESTARTING`	The node is restarting
`NDB_MGM_NODE_STATUS_SINGLEUSER`	The node is running in single-user (maintenance) mode
`NDB_MGM_NODE_STATUS_RESUME`	The node is in resume mode
`NDB_MGM_NODE_STATUS_CONNECTED`	The node is connected

3.3.3 The ndb_mgm_error Type

Description. The values for this type are the error codes that may be generated by MGM API functions. These may be found in Section 3.5, “MGM API Errors”.

See also Section 3.2.2.1, “ndb_mgm_get_latest_error()”, for more information.

3.3.4 The Ndb_logevent_type Type

Description. These are the types of log events available in the MGM API, grouped by event category. (See Section 3.3.7, “The ndb_mgm_event_category Type”.)

Enumeration values. Possible values are shown, along with descriptions, in the following table:

Category	Type	Description
NDB_MGM_EVENT_CATEGORY_CONNECTION		(Connection events)
	NDB_LE_Connected	The node has connected
	NDB_LE_Disconnected	The node was disconnected
	NDB_LE_CommunicationClosed	Communication with the node has been closed
	NDB_LE_CommunicationOpened	Communication with the node has been started
	NDB_LE_ConnectedApiVersion	The API version used by an API node; in the case of a MySQL server (SQL node), this is the same as displayed by SELECT VERSION()
NDB_MGM_EVENT_CATEGORY_CHECKPOINT		(Checkpoint events)
	NDB_LE_GlobalCheckpointStarted	A global checkpoint has been started
	NDB_LE_GlobalCheckpointCompleted	A global checkpoint has been completed
	NDB_LE_LocalCheckpointStarted	The node has begun a local checkpoint
	NDB_LE_LocalCheckpointCompleted	The node has completed a local checkpoint
	NDB_LE_LCPStoppedInCalcKeepGci	The lcoal checkpoint was aborted, but the last global checkpoint was preserved
	NDB_LE_LCPFragmentCompleted	Copying of a table fragment was completed
NDB_MGM_EVENT_CATEGORY_STARTUP		(Startup events)
	NDB_LE_NDBStartStarted	The node has begun to start
	NDB_LE_NDBStartCompleted	The node has completed the startup process
	NDB_LE_STTORRYRecieved	The node received an STTORRY signal, indicating that the reading of configuration data is underway; see Configuration Read Phase (STTOR Phase -1), and STTOR Phase 0, for more information
	NDB_LE_StartPhaseCompleted	A node start phase has been completed

Category	Type	Description
	NDB_LE_CM_REGCONF	The node has received a CM_REGCONF signal; see STTOR Phase 1, for more information
	NDB_LE_CM_REGREF	The node has received a CM_REGREF signal; see STTOR Phase 1, for more information
	NDB_LE_FIND_NEIGHBOURS	The node has discovered its neighboring nodes in the cluster
	NDB_LE_NDBStopStarted	The node is beginning to shut down
	NDB_LE_NDBStopCompleted	
	NDB_LE_NDBStopForced	The node is being forced to shut down (usually indicates a severe problem in the cluster)
	NDB_LE_NDBStopAborted	The started to shut down, but was forced to continue running; this happens, for example, when a STOP command was issued in the management client for a node such that the cluster would no longer be able to keep all data available if the node were shut down
	NDB_LE_StartREDOLog	Redo logging has been started
	NDB_LE_StartLog	Logging has started
	NDB_LE_UNDORecordsExecuted	The node has read and executed all records from the redo log
	NDB_LE_StartReport	The node is issuing a start report
NDB_MGM_EVENT_CATEGORY_NODE_RESTART		(Restart events)
	NDB_LE_NR_CopyDict	The node is copying the data dictionary
	NDB_LE_NR_CopyDistr	The node is copying data distribution information
	NDB_LE_NR_CopyFragsStarted	The node is copying table fragments

Category	Type	Description
	NDB_LE_NR_CopyFragDone	The node has completed copying a table fragment
	NDB_LE_NR_CopyFragsCompleted	The node has completed copying all necessary table fragments
NDB_MGM_EVENT_CATEGORY_NODE_RESTART		(Node failure and arbitration)
	NDB_LE_NodeFailCompleted	All (remaining) nodes has been notified of the failure of a data node
	NDB_LE_NODE_FAILREP	A data node has failed
	NDB_LE_ArbitState	This event is used to report on the current state of arbitration in the cluster
	NDB_LE_ArbitResult	This event is used to report on the outcome of node arbitration
	NDB_LE_GCP_TakeoverStarted	The node is attempting to become the master node (to assume responsibility for GCPs)
	NDB_LE_GCP_TakeoverCompleted	The node has become the master (and assumed responsibility for GCPs)
	NDB_LE_LCP_TakeoverStarted	The node is attempting to become the master node (to assume responsibility for LCPs)
	NDB_LE_LCP_TakeoverCompleted	The node has become the master (and assumed responsibility for LCPs)
NDB_MGM_EVENT_CATEGORY_STATISTIC		(Statistics events)
	NDB_LE_TransReportCounters	This indicates a report of transaction activity, which is given approximately once every 10 seconds
	NDB_LE_OperationReportCounters	Indicates a report on the number of operations performed by this node (also provided approximately once every 10 seconds)
	NDB_LE_TableCreated	A new table has been created

Category	Type	Description
	NDB_LE_UndoLogBlocked	Undo logging is blocked because the log buffer is close to overflowing
	NDB_LE_JobStatistic	
	NDB_LE_SendBytesStatistic	Indicates a report of the average number of bytes transmitted per send operation by this node
	NDB_LE_ReceiveBytesStatistic	Indicates a report of the average number of bytes received per send operation to this node
	NDB_LE_MemoryUsage	A DUMP 1000 command has been issued to this node, and it is reporting its memory usage in turn
NDB_MGM_EVENT_CATEGORY_ERROR		(Errors and warnings)
	NDB_LE_TransporterError	A transporter error has occurred; see NDB Transporter Errors, for transporter error codes and messages
	NDB_LE_TransporterWarning	A potential problem is occurring in the transporter; see NDB Transporter Errors, for transporter error codes and messages
	NDB_LE_MissedHeartbeat	Indicates a data node has missed a hreatbeat expected from another data node
	NDB_LE_DeadDueToHeartbeat	A data node has missed at least 3 heartbeats in succssion from another data node, and is reporting that it can no longer communicate with that data node
	NDB_LE_WarningEvent	Indicates a warning message
NDB_MGM_EVENT_CATEGORY_INFO		(Information events)
	NDB_LE_SentHeartbeat	A node heartbeat has been sent
	NDB_LE_CreateLogBytes	
	NDB_LE_InfoEvent	Indicates an informational message

Category	Type	Description
[Single-User Mode]	NDB_LE_SingleUser	The cluster has entered or exited single user mode
	NDB_LE_EventBufferStatus	This type of event indicates potentially excessive usage of the event buffer
	NDB_LE_EventBufferStatus2	Provides improved reporting of event buffer status; added in NDB 7.5.1
NDB_MGM_EVENT_CATEGORY_BACKUP		(Backups)
	NDB_LE_BackupStarted	A backup has been started
	NDB_LE_BackupFailedToStart	A backup has failed to start
	NDB_LE_BackupCompleted	A backup has been completed successfully
	NDB_LE_BackupAborted	A backup in progress was terminated by the user

3.3.5 The ndb_mgm_event_severity Type

Description. These are the log event severities used to filter the cluster log by `ndb_mgm_set_clusterlog_severity_filter()`, and to filter listening to events by `ndb_mgm_listen_event()`.

Enumeration values. Possible values are shown, along with descriptions, in the following table:

Value	Description
NDB_MGM_ILLEGAL_EVENT_SEVERITY	Invalid event severity specified
NDB_MGM_EVENT_SEVERITY_ON	Cluster logging is enabled
NDB_MGM_EVENT_SEVERITY_DEBUG	*Used for NDB Cluster development only*
NDB_MGM_EVENT_SEVERITY_INFO	Informational messages
NDB_MGM_EVENT_SEVERITY_WARNING	Conditions that are not errors as such, but that might require special handling
NDB_MGM_EVENT_SEVERITY_ERROR	Nonfatal error conditions that should be corrected
NDB_MGM_EVENT_SEVERITY_CRITICAL	Critical conditions such as device errors or out of memory errors
NDB_MGM_EVENT_SEVERITY_ALERT	Conditions that require immediate attention, such as corruption of the cluster
NDB_MGM_EVENT_SEVERITY_ALL	All severity levels

See Section 3.2.7.2, “ndb_mgm_set_clusterlog_severity_filter()”, and Section 3.2.1.1, “ndb_mgm_listen_event()”, for information on how this type is used by those functions.

3.3.6 The ndb_logevent_handle_error Type

Description. This type is used to describe log event errors.

Enumeration values. Possible values are shown, along with descriptions, in the following table:

Value	Description
NDB_LEH_NO_ERROR	No error
NDB_LEH_READ_ERROR	Read error
NDB_LEH_MISSING_EVENT_SPECIFIER	Invalid, incomplete, or missing log event specification
NDB_LEH_UNKNOWN_EVENT_TYPE	Unknown log event type
NDB_LEH_UNKNOWN_EVENT_VARIABLE	Unknown log event variable
NDB_LEH_INTERNAL_ERROR	Internal error
NDB_LEH_CONNECTION_ERROR	Connection error, or lost connection with management server

NDB_LEH_CONNECTION_ERROR was added in NDB 7.4.13 and NDB 7.5.4. (BUG #19474782)

3.3.7 The ndb_mgm_event_category Type

Description. These are the log event categories referenced in Section 3.3.4, "The Ndb_logevent_type Type". They are also used by the MGM API functions ndb_mgm_set_clusterlog_loglevel() and ndb_mgm_listen_event().

Enumeration values. Possible values are shown, along with descriptions, in the following table:

Value	Description
NDB_MGM_ILLEGAL_EVENT_CATEGORY	Invalid log event category
NDB_MGM_EVENT_CATEGORY_STARTUP	Log events occurring during startup
NDB_MGM_EVENT_CATEGORY_SHUTDOWN	Log events occurring during shutdown
NDB_MGM_EVENT_CATEGORY_STATISTIC	Statistics log events
NDB_MGM_EVENT_CATEGORY_CHECKPOINT	Log events related to checkpoints
NDB_MGM_EVENT_CATEGORY_NODE_RESTART	Log events occurring during node restart
NDB_MGM_EVENT_CATEGORY_CONNECTION	Log events relating to connections between cluster nodes
NDB_MGM_EVENT_CATEGORY_BACKUP	Log events relating to backups
NDB_MGM_EVENT_CATEGORY_CONGESTION	Log events relating to congestion
NDB_MGM_EVENT_CATEGORY_INFO	Uncategorised log events (severity level INFO)
NDB_MGM_EVENT_CATEGORY_ERROR	Uncategorised log events (severity level WARNING, ERROR, CRITICAL, or ALERT)

See Section 3.2.7.4, "ndb_mgm_set_clusterlog_loglevel()", and Section 3.2.1.1, "ndb_mgm_listen_event()", for more information.

3.4 MGM API Structures

This section covers the programming structures available in the MGM API.

3.4.1 The ndb_logevent Structure

Description. This structure models a Cluster log event, and is used for storing and retrieving log event information.

Definition. ndb_logevent has 8 members, the first 7 of which are shown in the following list:

- void* *handle*: An NdbLogEventHandle, set by ndb_logevent_get_next(). This handle is used only for purposes of comparison.

- *type*: Tells which type of event (Ndb_logevent_type) this is.
- unsigned *time*: The time at which the log event was registered with the management server.
- *category*: The log event category (ndb_mgm_event_category).
- *severity*: The log event severity (ndb_mgm_event_severity).
- unsigned *level*: The log event level. This is a value in the range of 0 to 15, inclusive.
- unsigned *source_nodeid*: The node ID of the node that reported this event.

The 8th member of this structure contains data specific to the log event, and is dependent on its type. It is defined as the union of a number of data structures, each corresponding to a log event type. Which structure to use is determined by the value of *type*, and is shown in the following table:

Ndb_logevent_type Value	**Structure**
NDB_LE_Connected	Connected: unsigned *node*
NDB_LE_Disconnected	Disconnected: unsigned *node*
NDB_LE_CommunicationClosed	CommunicationClosed: unsigned *node*
NDB_LE_CommunicationOpened	CommunicationOpened: unsigned *node*
NDB_LE_ConnectedApiVersion	ConnectedApiVersion: unsigned *node* unsigned *version*
NDB_LE_GlobalCheckpointStarted	GlobalCheckpointStarted: unsigned *gci*
NDB_LE_GlobalCheckpointCompleted	GlobalCheckpointCompleted: unsigned *gci*
NDB_LE_LocalCheckpointStarted	LocalCheckpointStarted: unsigned *lci* unsigned *keep_gci* unsigned *restore_gci*
NDB_LE_LocalCheckpointCompleted	LocalCheckpointCompleted: unsigned *lci*
NDB_LE_LCPStoppedInCalcKeepGci	LCPStoppedInCalcKeepGci: unsigned *data*
NDB_LE_LCPFragmentCompleted	LCPFragmentCompleted:

Ndb_logevent_type Value	**Structure**
	unsigned *node* unsigned *table_id* unsigned *fragment_id*
NDB_LE_UndoLogBlocked	UndoLogBlocked: unsigned *acc_count* unsigned *tup_count*
NDB_LE_NDBStartStarted	NDBStartStarted: unsigned *version*
NDB_LE_NDBStartCompleted	NDBStartCompleted: unsigned *version*
NDB_LE_STTORRYRecieved	STTORRYRecieved: *[NONE]*
NDB_LE_StartPhaseCompleted	StartPhaseCompleted: unsigned *phase* unsigned *starttype*
NDB_LE_CM_REGCONF	CM_REGCONF: unsigned *own_id* unsigned *president_id* unsigned *dynamic_id*
NDB_LE_CM_REGREF	CM_REGREF: unsigned *own_id* unsigned *other_id* unsigned *cause*
NDB_LE_FIND_NEIGHBOURS	FIND_NEIGHBOURS: unsigned *own_id* unsigned *left_id* unsigned *right_id* unsigned *dynamic_id*
NDB_LE_NDBStopStarted	NDBStopStarted: unsigned *stoptype*
NDB_LE_NDBStopCompleted	NDBStopCompleted: unsigned *action* unsigned *signum*
NDB_LE_NDBStopForced	NDBStopForced: unsigned *action* unsigned *signum* unsigned *error* unsigned *sphase* unsigned *extra*

Ndb_logevent_type Value	**Structure**
NDB_LE_NDBStopAborted	NDBStopAborted: *[NONE]*
NDB_LE_StartREDOLog	StartREDOLog: unsigned *node* unsigned *keep_gci* unsigned *completed_gci* unsigned *restorable_gci*
NDB_LE_StartLog	StartLog: unsigned *log_part* unsigned *start_mb* unsigned *stop_mb* unsigned *gci*
NDB_LE_UNDORecordsExecuted	UNDORecordsExecuted: unsigned *block* unsigned *data1* unsigned *data2* unsigned *data3* unsigned *data4* unsigned *data5* unsigned *data6* unsigned *data7* unsigned *data8* unsigned *data9* unsigned *data10*
NDB_LE_NR_CopyDict	NR_CopyDict: *[NONE]*
NDB_LE_NR_CopyDistr	NR_CopyDistr: *[NONE]*
NDB_LE_NR_CopyFragsStarted	NR_CopyFragsStarted: unsigned *dest_node*
NDB_LE_NR_CopyFragDone	NR_CopyFragDone: unsigned *dest_node* unsigned *table_id* unsigned *fragment_id*
NDB_LE_NR_CopyFragsCompleted	NR_CopyFragsCompleted: unsigned *dest_node*
NDB_LE_NodeFailCompleted	NodeFailCompleted: unsigned *block* unsigned *failed_node* unsigned *completing_node* (For *block* and *completing_node*, 0 is interpreted as “all”.)

Ndb_logevent_type Value	**Structure**
NDB_LE_NODE_FAILREP	NODE_FAILREP: unsigned *failed_node* unsigned *failure_state*
NDB_LE_ArbitState	ArbitState: unsigned *code* unsigned *arbit_node* unsigned *ticket_0* unsigned *ticket_1*
NDB_LE_ArbitResult	ArbitResult: unsigned *code* unsigned *arbit_node* unsigned *ticket_0* unsigned *ticket_1*
NDB_LE_GCP_TakeoverStarted	GCP_TakeoverStarted: *[NONE]*
NDB_LE_GCP_TakeoverCompleted	GCP_TakeoverCompleted: *[NONE]*
NDB_LE_LCP_TakeoverStarted	LCP_TakeoverStarted: *[NONE]*
NDB_LE_TransReportCounters	TransReportCounters: unsigned *trans_count* unsigned *commit_count* unsigned *read_count* unsigned *simple_read_count* unsigned *write_count* unsigned *attrinfo_count* unsigned *conc_op_count* unsigned *abort_count* unsigned *scan_count* unsigned *range_scan_count*
NDB_LE_OperationReportCounters	OperationReportCounters: unsigned *ops*
NDB_LE_TableCreated	TableCreated: unsigned *table_id*
NDB_LE_JobStatistic	JobStatistic: unsigned *mean_loop_count*
NDB_LE_SendBytesStatistic	SendBytesStatistic: unsigned *to_node* unsigned *mean_sent_bytes*
NDB_LE_ReceiveBytesStatistic	ReceiveBytesStatistic:

Ndb_logevent_type Value	**Structure**
	`unsigned from_node` `unsigned mean_received_bytes`

NDB_LE_MemoryUsage	MemoryUsage:
	`int      gth`
	`unsigned page_size_kb`
	`unsigned pages_used`
	`unsigned pages_total`
	`unsigned block`

NDB_LE_TransporterError	TransporterError:
	`unsigned to_node`
	`unsigned code`

NDB_LE_TransporterWarning	TransporterWarning:
	`unsigned to_node`
	`unsigned code`

NDB_LE_MissedHeartbeat	MissedHeartbeat:
	`unsigned node`
	`unsigned count`

NDB_LE_DeadDueToHeartbeat	DeadDueToHeartbeat: `unsigned node`
NDB_LE_WarningEvent	WarningEvent: *[NOT YET IMPLEMENTED]*
NDB_LE_SentHeartbeat	SentHeartbeat: `unsigned node`
NDB_LE_CreateLogBytes	CreateLogBytes: `unsigned node`
NDB_LE_InfoEvent	InfoEvent: *[NOT YET IMPLEMENTED]*

NDB_LE_EventBufferStatus (NDB 7.5.0 and earlier)	EventBufferStatus::
	`unsigned usage`
	`unsigned alloc`
	`unsigned max`
	`unsigned apply_gci_l`
	`unsigned apply_gci_h`
	`unsigned latest_gci_l`
	`unsigned latest_gci_h`

NDB_LE_EventBufferStatus2 (NDB 7.5.1 and later)	EventBufferStatus2:
	`unsigned usage`
	`unsigned alloc`
	`unsigned max`

Ndb_logevent_type Value	Structure
	unsigned *latest_consumed_epoch_l* unsigned *latest_consumed_epoch_h* unsigned *latest_buffered_epoch_l* unsigned *latest_buffered_epoch_h* unsigned *ndb_reference* unsigned *report_reason* *report_reason* is one of NO_REPORT, COMPLETELY_BUFFERING, PARTIALLY_DISCARDING, COMPLETELY_DISCARDING, PARTIALLY_BUFFERING, BUFFERED_EPOCHS_OVER_THRESHOLD, ENOUGH_FREE_EVENTBUFFER, or LOW_FREE_EVENTBUFFER; see Event Buffer Reporting in the Cluster Log, for descriptions of these values
NDB_LE_BackupStarted	BackupStarted: unsigned *starting_node* unsigned *backup_id*
NDB_LE_BackupFailedToStart	BackupFailedToStart: unsigned *starting_node* unsigned *error*
NDB_LE_BackupCompleted	BackupCompleted: unsigned *starting_node* unsigned *backup_id* unsigned *start_gci* unsigned *stop_gci* unsigned *n_records* unsigned *n_log_records* unsigned *n_bytes* unsigned *n_log_bytes*
NDB_LE_BackupAborted	BackupAborted: unsigned *starting_node* unsigned *backup_id* unsigned *error*
NDB_LE_SingleUser	SingleUser: unsigned *type* unsigned *node_id*
NDB_LE_StartReport	StartReport: unsigned *report_type* unsigned *remaining_time* unsigned *bitmask_size* unsigned *bitmask_data[1]*

3.4.2 The ndb_mgm_node_state Structure

Description. Provides information on the status of a Cluster node.

Definition. This structure contains the following members:

- `int node_id`: The cluster node's node ID.
- `enum ndb_mgm_node_type node_type`: The node type.

  See Section 3.3.1, “The ndb_mgm_node_type Type”, for permitted values.
- `enum ndb_mgm_node_status node_status`: The node's status.

  See Section 3.3.2, “The ndb_mgm_node_status Type”, for permitted values.
- `int start_phase`: The start phase.

  This is valid only if the *`node_type`* is `NDB_MGM_NODE_TYPE_NDB` and the *`node_status`* is `NDB_MGM_NODE_STATUS_STARTING`.
- `int dynamic_id`: The ID for heartbeats and master takeover.

  Valid only for data (`ndbd`) nodes.
- `int node_group`: The node group to which the node belongs.

  Valid only for data (`ndbd`) nodes.
- `int version`: Internal version number.
- `int connect_count`: The number of times this node has connected to or disconnected from the management server.
- `char connect_address[]`: The IP address of this node as seen by the other nodes in the cluster.

3.4.3 The ndb_mgm_cluster_state Structure

Description. Provides information on the status of all Cluster nodes. This structure is returned by `ndb_mgm_get_status()`.

Definition. This structure has the following two members;

- `int no_of_nodes`: The number of elements in the *`node_states`* array.
- `struct ndb_mgm_node_state node_states[]`: An array containing the states of the nodes.

  Each element of this array is an `ndb_mgm_node_state` structure.

See Section 3.2.5.1, “ndb_mgm_get_status()”.

3.4.4 The ndb_mgm_reply Structure

Description. Contains response information, consisting of a response code and a corresponding message, from the management server.

Definition. This structure contains two members, as shown here:

- `int return_code`: For a successful operation, this value is `0`; otherwise, it contains an error code.

  For error codes, see Section 3.3.3, “The ndb_mgm_error Type”.
- `char message[256]`: contains the text of the response or error message.

See Section 3.2.2.1, “ndb_mgm_get_latest_error()”, and Section 3.2.2.2, “ndb_mgm_get_latest_error_msg()”.

3.5 MGM API Errors

The following sections list the values of `MGM` errors by type. There are six types of `MGM` errors:

1. request errors
2. node ID allocation errors
3. service errors
4. backup errors
5. single user mode errors
6. general usage errors

There is only one general usage error.

3.5.1 Request Errors

These are errors generated by failures to connect to a management server.

Value	Description
NDB_MGM_ILLEGAL_CONNECT_STRING	Invalid connection string
NDB_MGM_ILLEGAL_SERVER_HANDLE	Invalid management server handle
NDB_MGM_ILLEGAL_SERVER_REPLY	Invalid response from management server
NDB_MGM_ILLEGAL_NUMBER_OF_NODES	Invalid number of nodes
NDB_MGM_ILLEGAL_NODE_STATUS	Invalid node status
NDB_MGM_OUT_OF_MEMORY	Memory allocation error
NDB_MGM_SERVER_NOT_CONNECTED	Management server not connected
NDB_MGM_COULD_NOT_CONNECT_TO_SOCKET	Not able to connect to socket

3.5.2 Node ID Allocation Errors

These errors result from a failure to assign a node ID to a cluster node.

Value	Description
NDB_MGM_ALLOCID_ERROR	Generic error; may be possible to retry and recover
NDB_MGM_ALLOCID_CONFIG_MISMATCH	Non-recoverable generic error

3.5.3 Service Errors

These errors result from the failure of a node or cluster to start, shut down, or restart.

Value	Description
NDB_MGM_START_FAILED	Startup failure
NDB_MGM_STOP_FAILED	Shutdown failure
NDB_MGM_RESTART_FAILED	Restart failure

3.5.4 Backup Errors

These are errors which result from problems with initiating or aborting backups.

Value	Description
NDB_MGM_COULD_NOT_START_BACKUP	Unable to initiate backup
NDB_MGM_COULD_NOT_ABORT_BACKUP	Unable to abort backup

3.5.5 Single User Mode Errors

These errors result from failures to enter or exit single user mode.

Value	Description
NDB_MGM_COULD_NOT_ENTER_SINGLE_USER_MODE	Unable to enter single-user mode
NDB_MGM_COULD_NOT_EXIT_SINGLE_USER_MODE	Unable to exit single-user mode

3.5.6 General Usage Errors

This is a general error type for errors which are otherwise not classifiable.

Value	Description
NDB_MGM_USAGE_ERROR	General usage error

3.6 MGM API Examples

This section contains MGM API coding examples.

3.6.1 Basic MGM API Event Logging Example

This example shows the basics of handling event logging using the MGM API.

The source code for this program may be found in the NDB Cluster source tree, in the file `storage/ndb/ndbapi-examples/mgmapi_logevent/main.cpp`.

```
#include <mysql.h>
#include <ndbapi/NdbApi.hpp>
#include <mgmapi.h>
#include <stdio.h>
#include <stdlib.h>

/*
 * export LD_LIBRARY_PATH=../../../../libmysql_r/.libs:../../src/.libs
 */

#define MGMERROR(h) \
{ \
  fprintf(stderr, "code: %d msg: %s\n", \
          ndb_mgm_get_latest_error(h), \
          ndb_mgm_get_latest_error_msg(h)); \
  exit(-1); \
}

#define LOGEVENTERROR(h) \
{ \
  fprintf(stderr, "code: %d msg: %s\n", \
          ndb_logevent_get_latest_error(h), \
          ndb_logevent_get_latest_error_msg(h)); \
  exit(-1); \
}

#define make_uint64(a,b) (((Uint64)(a)) + (((Uint64)(b)) << 32))

int main(int argc, char** argv)
{
  NdbMgmHandle h;
  NdbLogEventHandle le;
  int filter[] = { 15, NDB_MGM_EVENT_CATEGORY_BACKUP,
     15, NDB_MGM_EVENT_CATEGORY_CONNECTION,
     15, NDB_MGM_EVENT_CATEGORY_NODE_RESTART,
     15, NDB_MGM_EVENT_CATEGORY_STARTUP,
     15, NDB_MGM_EVENT_CATEGORY_ERROR,
     0 };
  struct ndb_logevent event;
```

```
 if (argc < 2)
 {
   printf("Arguments are <connect_string cluster> [<iterations>].\n");
   exit(-1);
 }
 const char *connectstring = argv[1];
 int iterations = -1;
 if (argc > 2)
   iterations = atoi(argv[2]);
 ndb_init();

 h= ndb_mgm_create_handle();
 if ( h == 0)
 {
   printf("Unable to create handle\n");
   exit(-1);
 }
 if (ndb_mgm_set_connectstring(h, connectstring) == -1)
 {
   printf("Unable to set connection string\n");
   exit(-1);
 }
 if (ndb_mgm_connect(h,0,0,0)) MGMERROR(h);

 le= ndb_mgm_create_logevent_handle(h, filter);
 if ( le == 0 )  MGMERROR(h);

 while (iterations-- != 0)
 {
   int timeout= 1000;
   int r= ndb_logevent_get_next(le,&event,timeout);
   if (r == 0)
     printf("No event within %d milliseconds\n", timeout);
   else if (r < 0)
     LOGEVENTERROR(le)
   else
   {
     switch (event.type) {
     case NDB_LE_BackupStarted:
printf("Node %d: BackupStarted\n", event.source_nodeid);
printf("  Starting node ID: %d\n", event.BackupStarted.starting_node);
printf("  Backup ID: %d\n", event.BackupStarted.backup_id);
break;
     case NDB_LE_BackupStatus:
printf("Node %d: BackupStatus\n", event.source_nodeid);
printf("  Starting node ID: %d\n", event.BackupStarted.starting_node);
printf("  Backup ID: %d\n", event.BackupStarted.backup_id);
printf("  Data written: %llu bytes (%llu records)\n",
              make_uint64(event.BackupStatus.n_bytes_lo,
                          event.BackupStatus.n_bytes_hi),
              make_uint64(event.BackupStatus.n_records_lo,
                          event.BackupStatus.n_records_hi));
printf("  Log written: %llu bytes (%llu records)\n",
              make_uint64(event.BackupStatus.n_log_bytes_lo,
                          event.BackupStatus.n_log_bytes_hi),
              make_uint64(event.BackupStatus.n_log_records_lo,
                          event.BackupStatus.n_log_records_hi));
break;
     case NDB_LE_BackupCompleted:
printf("Node %d: BackupCompleted\n", event.source_nodeid);
printf("  Backup ID: %d\n", event.BackupStarted.backup_id);
printf("  Data written: %llu bytes (%llu records)\n",
              make_uint64(event.BackupCompleted.n_bytes,
                          event.BackupCompleted.n_bytes_hi),
              make_uint64(event.BackupCompleted.n_records,
                          event.BackupCompleted.n_records_hi));
printf("  Log written: %llu bytes (%llu records)\n",
              make_uint64(event.BackupCompleted.n_log_bytes,
                          event.BackupCompleted.n_log_bytes_hi),
              make_uint64(event.BackupCompleted.n_log_records,
                          event.BackupCompleted.n_log_records_hi));
```

```
 break;
      case NDB_LE_BackupAborted:
 printf("Node %d: BackupAborted\n", event.source_nodeid);
 break;
      case NDB_LE_BackupFailedToStart:
 printf("Node %d: BackupFailedToStart\n", event.source_nodeid);
 break;

      case NDB_LE_NodeFailCompleted:
 printf("Node %d: NodeFailCompleted\n", event.source_nodeid);
 break;
      case NDB_LE_ArbitResult:
 printf("Node %d: ArbitResult\n", event.source_nodeid);
 printf("  code %d, arbit_node %d\n",
        event.ArbitResult.code & 0xffff,
        event.ArbitResult.arbit_node);
 break;
      case NDB_LE_DeadDueToHeartbeat:
 printf("Node %d: DeadDueToHeartbeat\n", event.source_nodeid);
 printf("  node %d\n", event.DeadDueToHeartbeat.node);
 break;

      case NDB_LE_Connected:
 printf("Node %d: Connected\n", event.source_nodeid);
 printf("  node %d\n", event.Connected.node);
 break;
      case NDB_LE_Disconnected:
 printf("Node %d: Disconnected\n", event.source_nodeid);
 printf("  node %d\n", event.Disconnected.node);
 break;
      case NDB_LE_NDBStartCompleted:
 printf("Node %d: StartCompleted\n", event.source_nodeid);
 printf("  version %d.%d.%d\n",
        event.NDBStartCompleted.version >> 16 & 0xff,
        event.NDBStartCompleted.version >> 8 & 0xff,
        event.NDBStartCompleted.version >> 0 & 0xff);
 break;
      case NDB_LE_ArbitState:
 printf("Node %d: ArbitState\n", event.source_nodeid);
 printf("  code %d, arbit_node %d\n",
        event.ArbitState.code & 0xffff,
        event.ArbitResult.arbit_node);
 break;

      default:
 break;
      }
    }
  }

  ndb_mgm_destroy_logevent_handle(&le);
  ndb_mgm_destroy_handle(&h);
  ndb_end(0);
  return 0;
}
```

3.6.2 MGM API Event Handling with Multiple Clusters

This example shown in this section illustrates the handling of log events using the MGM API on multiple clusters in a single application.

The source code for this program may be found in the NDB Cluster source tree, in the file `storage/ndb/ndbapi-examples/mgmapi_logevent2/main.cpp`.

Note

This file was previously named `mgmapi_logevent2.cpp`.

```
#include <mysql.h>
```

```
#include <ndbapi/NdbApi.hpp>
#include <mgmapi.h>
#include <stdio.h>
#include <stdlib.h>

/*
 * export LD_LIBRARY_PATH=../../../libmysql_r/.libs:../../../ndb/src/.libs
 */

#define MGMERROR(h) \
{ \
  fprintf(stderr, "code: %d msg: %s\n", \
          ndb_mgm_get_latest_error(h), \
          ndb_mgm_get_latest_error_msg(h)); \
  exit(-1); \
}

#define LOGEVENTERROR(h) \
{ \
  fprintf(stderr, "code: %d msg: %s\n", \
          ndb_logevent_get_latest_error(h), \
          ndb_logevent_get_latest_error_msg(h)); \
  exit(-1); \
}

int main(int argc, char** argv)
{
  NdbMgmHandle h1,h2;
  NdbLogEventHandle le1,le2;
  int filter[] = { 15, NDB_MGM_EVENT_CATEGORY_BACKUP,
     15, NDB_MGM_EVENT_CATEGORY_CONNECTION,
     15, NDB_MGM_EVENT_CATEGORY_NODE_RESTART,
     15, NDB_MGM_EVENT_CATEGORY_STARTUP,
     15, NDB_MGM_EVENT_CATEGORY_ERROR,
     0 };
  struct ndb_logevent event1, event2;

  if (argc < 3)
  {
    printf("Arguments are <connect_string cluster 1>",
           "<connect_string cluster 2> [<iterations>].\n");
    exit(-1);
  }
  const char *connectstring1 = argv[1];
  const char *connectstring2 = argv[2];
  int iterations = -1;
  if (argc > 3)
    iterations = atoi(argv[3]);
  ndb_init();

  h1= ndb_mgm_create_handle();
  h2= ndb_mgm_create_handle();
  if ( h1 == 0 || h2 == 0 )
  {
    printf("Unable to create handle\n");
    exit(-1);
  }
  if (ndb_mgm_set_connectstring(h1, connectstring1) == -1 ||
      ndb_mgm_set_connectstring(h2, connectstring1))
  {
    printf("Unable to set connection string\n");
    exit(-1);
  }
  if (ndb_mgm_connect(h1,0,0,0)) MGMERROR(h1);
  if (ndb_mgm_connect(h2,0,0,0)) MGMERROR(h2);

  if ((le1= ndb_mgm_create_logevent_handle(h1, filter)) == 0) MGMERROR(h1);
  if ((le2= ndb_mgm_create_logevent_handle(h1, filter)) == 0) MGMERROR(h2);

  while (iterations-- != 0)
  {
    int timeout= 1000;
```

```
   int r1= ndb_logevent_get_next(le1,&event1,timeout);
   if (r1 == 0)
     printf("No event within %d milliseconds\n", timeout);
   else if (r1 < 0)
     LOGEVENTERROR(le1)
   else
   {
     switch (event1.type) {
     case NDB_LE_BackupStarted:
printf("Node %d: BackupStarted\n", event1.source_nodeid);
printf("  Starting node ID: %d\n", event1.BackupStarted.starting_node);
printf("  Backup ID: %d\n", event1.BackupStarted.backup_id);
break;
     case NDB_LE_BackupCompleted:
printf("Node %d: BackupCompleted\n", event1.source_nodeid);
printf("  Backup ID: %d\n", event1.BackupStarted.backup_id);
break;
     case NDB_LE_BackupAborted:
printf("Node %d: BackupAborted\n", event1.source_nodeid);
break;
     case NDB_LE_BackupFailedToStart:
printf("Node %d: BackupFailedToStart\n", event1.source_nodeid);
break;

     case NDB_LE_NodeFailCompleted:
printf("Node %d: NodeFailCompleted\n", event1.source_nodeid);
break;
     case NDB_LE_ArbitResult:
printf("Node %d: ArbitResult\n", event1.source_nodeid);
printf("  code %d, arbit_node %d\n",
       event1.ArbitResult.code & 0xffff,
       event1.ArbitResult.arbit_node);
break;
     case NDB_LE_DeadDueToHeartbeat:
printf("Node %d: DeadDueToHeartbeat\n", event1.source_nodeid);
printf("  node %d\n", event1.DeadDueToHeartbeat.node);
break;

     case NDB_LE_Connected:
printf("Node %d: Connected\n", event1.source_nodeid);
printf("  node %d\n", event1.Connected.node);
break;
     case NDB_LE_Disconnected:
printf("Node %d: Disconnected\n", event1.source_nodeid);
printf("  node %d\n", event1.Disconnected.node);
break;
     case NDB_LE_NDBStartCompleted:
printf("Node %d: StartCompleted\n", event1.source_nodeid);
printf("  version %d.%d.%d\n",
       event1.NDBStartCompleted.version >> 16 & 0xff,
       event1.NDBStartCompleted.version >> 8 & 0xff,
       event1.NDBStartCompleted.version >> 0 & 0xff);
break;
     case NDB_LE_ArbitState:
printf("Node %d: ArbitState\n", event1.source_nodeid);
printf("  code %d, arbit_node %d\n",
       event1.ArbitState.code & 0xffff,
       event1.ArbitResult.arbit_node);
break;

     default:
break;
     }
   }

   int r2= ndb_logevent_get_next(le1,&event2,timeout);
   if (r2 == 0)
     printf("No event within %d milliseconds\n", timeout);
   else if (r2 < 0)
     LOGEVENTERROR(le2)
   else
   {
```

```
      switch (event2.type) {
      case NDB_LE_BackupStarted:
 printf("Node %d: BackupStarted\n", event2.source_nodeid);
 printf("  Starting node ID: %d\n", event2.BackupStarted.starting_node);
 printf("  Backup ID: %d\n", event2.BackupStarted.backup_id);
 break;
      case NDB_LE_BackupCompleted:
 printf("Node %d: BackupCompleted\n", event2.source_nodeid);
 printf("  Backup ID: %d\n", event2.BackupStarted.backup_id);
 break;
      case NDB_LE_BackupAborted:
 printf("Node %d: BackupAborted\n", event2.source_nodeid);
 break;
      case NDB_LE_BackupFailedToStart:
 printf("Node %d: BackupFailedToStart\n", event2.source_nodeid);
 break;

      case NDB_LE_NodeFailCompleted:
 printf("Node %d: NodeFailCompleted\n", event2.source_nodeid);
 break;
      case NDB_LE_ArbitResult:
 printf("Node %d: ArbitResult\n", event2.source_nodeid);
 printf("  code %d, arbit_node %d\n",
        event2.ArbitResult.code & 0xffff,
        event2.ArbitResult.arbit_node);
 break;
      case NDB_LE_DeadDueToHeartbeat:
 printf("Node %d: DeadDueToHeartbeat\n", event2.source_nodeid);
 printf("  node %d\n", event2.DeadDueToHeartbeat.node);
 break;

      case NDB_LE_Connected:
 printf("Node %d: Connected\n", event2.source_nodeid);
 printf("  node %d\n", event2.Connected.node);
 break;
      case NDB_LE_Disconnected:
 printf("Node %d: Disconnected\n", event2.source_nodeid);
 printf("  node %d\n", event2.Disconnected.node);
 break;
      case NDB_LE_NDBStartCompleted:
 printf("Node %d: StartCompleted\n", event2.source_nodeid);
 printf("  version %d.%d.%d\n",
        event2.NDBStartCompleted.version >> 16 & 0xff,
        event2.NDBStartCompleted.version >> 8 & 0xff,
        event2.NDBStartCompleted.version >> 0 & 0xff);
 break;
      case NDB_LE_ArbitState:
 printf("Node %d: ArbitState\n", event2.source_nodeid);
 printf("  code %d, arbit_node %d\n",
        event2.ArbitState.code & 0xffff,
        event2.ArbitResult.arbit_node);
 break;

      default:
 break;
      }
    }
  }

  ndb_mgm_destroy_logevent_handle(&le1);
  ndb_mgm_destroy_logevent_handle(&le2);
  ndb_mgm_destroy_handle(&h1);
  ndb_mgm_destroy_handle(&h2);
  ndb_end(0);
  return 0;
}
```

Chapter 4 MySQL Cluster Connector for Java

Table of Contents

4.1 MySQL Cluster Connector for Java: Overview 529
4.1.1 MySQL Cluster Connector for Java Architecture 529
4.1.2 Java and NDB Cluster 529
4.1.3 The ClusterJ API and Data Object Model 531
4.2 Using MySQL Cluster Connector for Java 533
4.2.1 Getting, Installing, and Setting Up MySQL Cluster Connector for Java 533
4.2.2 Using ClusterJ 535
4.2.3 Using JPA with NDB Cluster 543
4.2.4 Using Connector/J with NDB Cluster 544
4.3 ClusterJ API Reference 544
4.3.1 com.mysql.clusterj 544
4.3.2 com.mysql.clusterj.annotation 598
4.3.3 com.mysql.clusterj.query 609
4.3.4 Constant field values 616
4.4 MySQL Cluster Connector for Java: Limitations and Known Issues 618

This chapter discusses using NDB Cluster with MySQL Cluster Connector for Java, which includes ClusterJ and ClusterJPA, a plugin for use with OpenJPA.

ClusterJ is a high level database API that is similar in style and concept to object-relational mapping persistence frameworks such as Hibernate and JPA. Because ClusterJ does not use the MySQL Server to access data in NDB Cluster, it can perform some operations much more quickly than can be done using JDBC. ClusterJ supports primary key and unique key operations and single-table queries; it does not support multi-table operations, including joins.

ClusterJPA is an OpenJPA implementation for NDB Cluster that attempts to offer the best possible performance by leveraging the strengths of both ClusterJ and JDBC. ID lookups and other such operations are performed using ClusterJ, while more complex queries that can benefit from the MySQL query optimizer are sent through the MySQL Server, using JDBC.

4.1 MySQL Cluster Connector for Java: Overview

This section provides a conceptual and architectural overview of the APIs available using the MySQL Cluster Connector for Java.

4.1.1 MySQL Cluster Connector for Java Architecture

MySQL Cluster Connector for Java is a collection of Java APIs for writing applications against NDB Cluster, including JDBC, JPA, and ClusterJ. These provide a number of different access paths and styles of access to NDB Cluster data. Section 4.1.2, “Java and NDB Cluster”, describes each of these APIs in more detail.

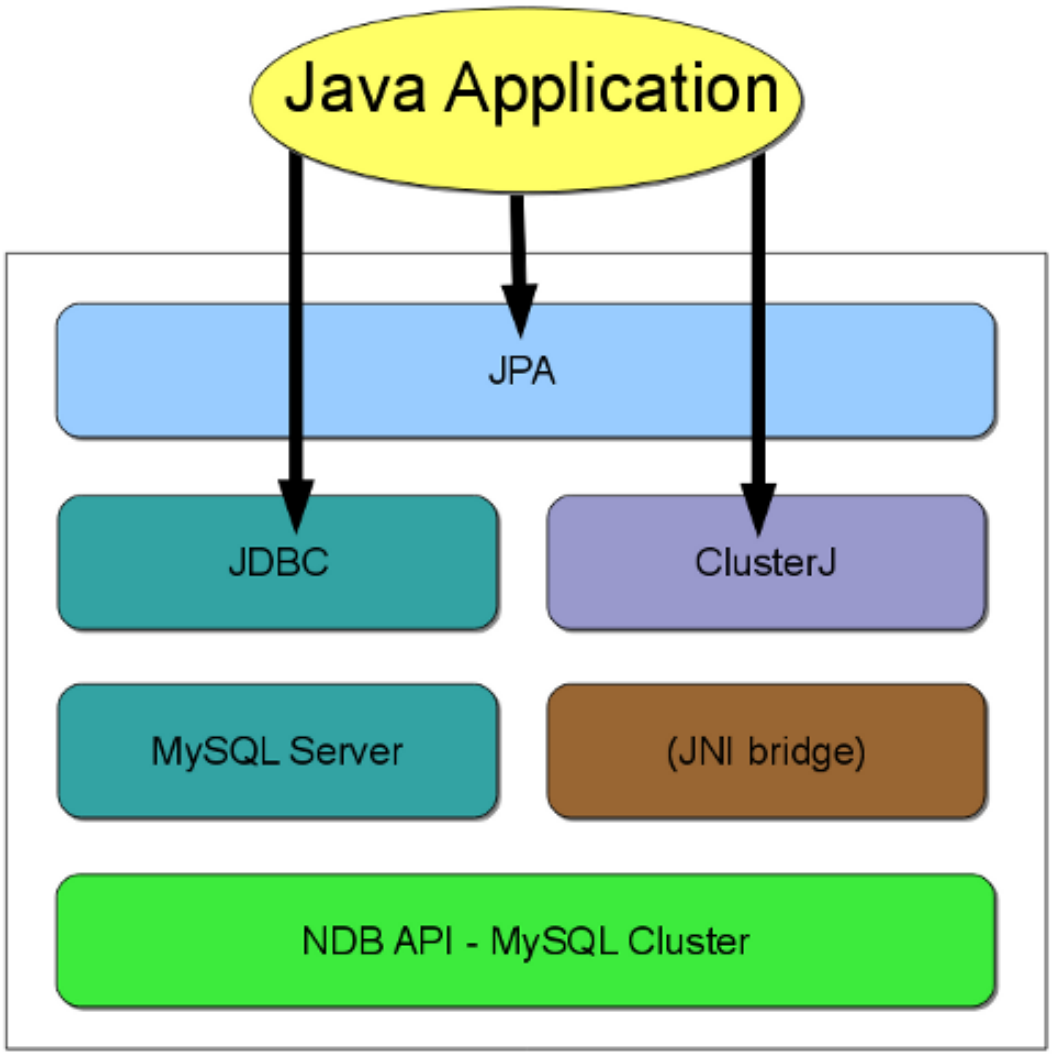
MySQL Cluster Connector for Java is included with all NDB Cluster source and binary releases. Building MySQL Cluster Connector for Java from source can be done as part of building NDB Cluster; however, it can also be built with Maven.

4.1.2 Java and NDB Cluster

A NDB Cluster is defined as one or more MySQL Servers providing access to an `NDBCLUSTER` storage engine—that is, to a set of NDB Cluster data nodes (`ndbd` processes). There are four main access paths from Java to `NDBCLUSTER`, listed here:

- **JDBC and mysqld.** JDBC works by sending SQL statements to the MySQL Server and returning result sets. When using JDBC, you must write the SQL, manage the connection, and copy any data from the result set that you want to use in your program as objects. The JDBC implementation most often used with the MySQL Server is MySQL Connector/J.
- **Java Persistence API (JPA) and JDBC.** JPA uses JDBC to connect to the MySQL Server. Unlike JDBC, JPA provides an object view of the data in the database.
- **ClusterJ.** ClusterJ uses a JNI bridge to the NDB API for direct access to `NDBCLUSTER`. It employs a style of data access that is based on a domain object model, similar in many ways to that employed by JPA. ClusterJ does not depend on the MySQL Server for data access.
- **ClusterJPA.** ClusterJPA is an adapter for the OpenJPA implementation. It can use either of two different access paths—JDBC or ClusterJ—to NDB Cluster data, depending on the type of operation to be performed. This can significantly improve performance for some uses, bypassing SQL, JDBC, and the MySQL Server entirely when it is more efficient to do so.

These paths are shown in the following API stack diagram:



JDBC and mysqld. Connector/J provides standard access through the MySQL JDBC driver. Using Connector/J, JDBC applications can be written to work with a MySQL server acting as an NDB Cluster SQL node in much the same way that other Connector/J applications work with any other MySQL Server instance.

For more information, see Section 4.2.4, “Using Connector/J with NDB Cluster”.

ClusterJ. ClusterJ is a native Java Connector for `NDBCLUSTER` (or `NDB`), the storage engine for NDB Cluster, in the style of Hibernate, JPA, and JDO. Like other persistence frameworks, ClusterJ uses the Data Mapper pattern, in which data is represented as domain objects, separate from business logic, mapping Java classes to database tables stored in the `NDBCLUSTER` storage engine.

Note

The `NDBCLUSTER` storage engine is often referred to (in MySQL documentation and elsewhere) simply as `NDB`. The terms `NDB` and `NDBCLUSTER` are synonymous, and you can use either `ENGINE=NDB` or `ENGINE=NDBCLUSTER` in a `CREATE TABLE` statement to create a clustered table.

ClusterJ does not need to connect to a `mysqld` process, having direct access to `NDBCLUSTER` using a JNI bridge that is included in the dynamic library `libnbdclient`. However, unlike JDBC, ClusterJ does not support table creation and other data definition operations; these must be performed by some other means, such as JDBC or the `mysql` client.

OpenJPA (ClusterJPA). ClusterJPA is an adapter for OpenJPA that can also bypass JDBC and MySQL Server, using ClusterJ for fast-track access to the cluster. However, for complex queries (not primary key lookups) ClusterJPA uses JDBC.

OpenJPA is an implementation of the JPA (Java Persistence API) specification, which provides an object-relational persistence framework with relationships, inheritance, and persistent classes. See openjpa.apache.org, for more information about OpenJPA.

ClusterJ is independent of ClusterJPA as well as JDBC. However, ClusterJ can be used together with these APIs. Because ClusterJ is limited to queries on single tables, and does not support relations or inheritance, you should use JPA if you need support for these features in your applications.

For more information, see Section 4.2.3, “Using JPA with NDB Cluster”.

Differences Between ClusterJPA and ClusterJ. While ClusterJPA and ClusterJ are similar in many ways, there are importance differences between the two, highlighted in the following list.

- ClusterJPA supports persistent classes, whereas ClusterJ only supports persistent interfaces.
- ClusterJPA supports relationships between persistent classes (typically modeled as logical foreign keys), whereas ClusterJ only supports single-valued fields directly mapped to database columns.
- ClusterJPA makes it possible for you to formulate queries that contain joins based on relationships in the domain object model, while ClusterJ does not support either relationships or joins.

  However, once you retrieve instances using a JPA query, you can update or delete these using the fast path of ClusterJ.
- ClusterJPA makes it possible to use the JPA API to declare some fields as lazily loaded, meaning that the data is only brought into memory when your program actually references it. ClusterJ, however, reads all mapped columns from the table whenever you access a row.

4.1.3 The ClusterJ API and Data Object Model

This section discusses the ClusterJ API and the object model used to represent the data handled by the application.

Application Programming Interface. The ClusterJ API depends on 4 main interfaces: `Session`, `SessionFactory`, `Transaction`, and `QueryBuilder`.

Session interface. All access to NDB Cluster data is done in the context of a session. The `Session` interface represents a user's or application's individual connection to an NDB Cluster. It contains methods for the following operations:

- Finding persistent instances by primary key
- Creating, updating, and deleting persistent instances
- Getting a query builder (see com.mysql.clusterj.query.QueryBuilder)
- Getting the current transaction (see com.mysql.clusterj.Transaction).

SessionFactory interface. Sessions are obtained from a `SessionFactory`, of which there is typically a single instance for each NDB Cluster that you want to access from the Java VM. `SessionFactory` stores configuration information about the cluster, such as the hostname and port number of the NDB Cluster management server. It also stores parameters regarding how to connect to the cluster, including connection delays and timeouts. For more information about SessionFactory and its use in a ClusterJ application, see Getting the SessionFactory and getting a Session.

Transaction interface. Transactions are not managed by the `Session` interface; like other modern application frameworks, ClusterJ separates transaction management from other persistence methods. Transaction demarcation might be done automatically by a container or in a web server servlet filter. Removing transaction completion methods from `Session` facilitates this separation of concerns.

The `Transaction` interface supports the standard begin, commit, and rollback behaviors required by a transactional database. In addition, it enables the user to mark a transaction as being rollback-only, which makes it possible for a component that is not responsible for completing a transaction to indicate that—due to an application or database error—the transaction must not be permitted to complete normally.

QueryBuilder interface. The `QueryBuilder` interface makes it possible to construct criteria queries dynamically, using domain object model properties as query modeling elements. Comparisons between parameters and database column values can be specified, including equal, greater and less than, between, and in operations. These comparisons can be combined using methods corresponding to the Boolean operators AND, OR, and NOT. Comparison of values to `NULL` is also supported.

Data model. ClusterJ provides access to data in NDB Cluster using domain objects, similar in many ways to the way that JPA models data.

In ClusterJ, the domain object mapping has the following characteristics:

- All tables map to persistent interfaces. For every `NDB` table in the cluster, ClusterJ uses one or more interfaces. In many cases, a single interface is used; but for cases where different columns are needed by different parts of the application, multiple interfaces can be mapped to the same table.

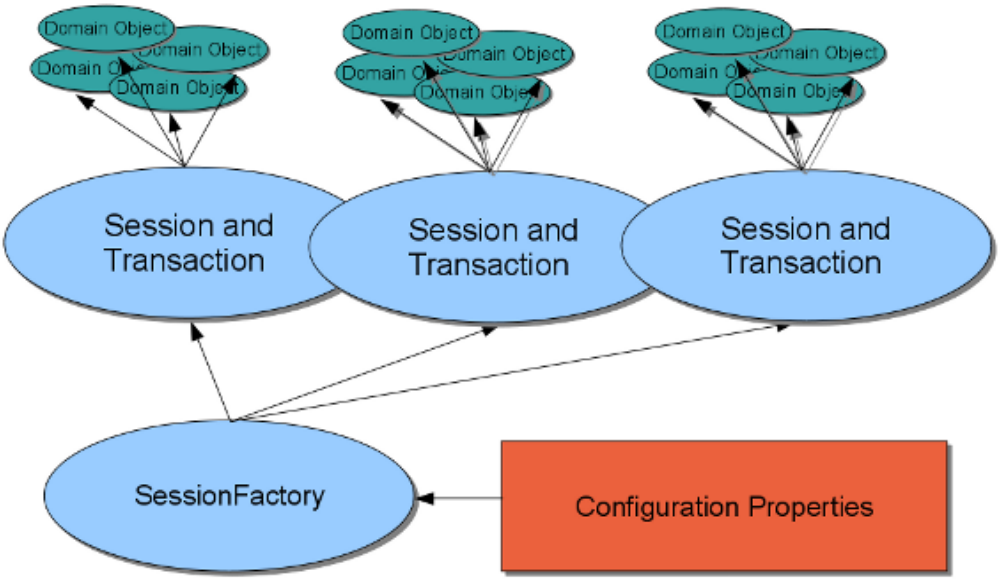
  However, the classes themselves are not persistent.

- Users map a subset of columns to persistent properties in interfaces. Thus, all properties map to columns; however, not all columns necessarily map to properties.

  All ClusterJ property names default to column names. The interface provides getter and setter methods for each property, with predictable corresponding method names.

- Annotations on interfaces define mappings.

The user view of the application environment and domain objects is illustrated in the following diagram, which shows the logical relationships among the modeling elements of the ClusterJ interfaces:



The `SessionFactory` is configured by a properties object that might have been loaded from a file or constructed dynamically by the application using some other means (see Section 4.2.2.1, “Executing ClusterJ Applications and Sessions”).

The application obtains `Session` instances from the `SessionFactory`, with at most one thread working with a `Session` at a time. A thread can manage multiple `Session` instances if there is some application requirement for multiple connections to the database.

Each session has its own collection of domain objects, each of which represents the data from one row in the database. The domain objects can represent data in any of the following states:

- New; not yet stored in the database
- Retrieved from the database; available to the application
- Updated; to be stored back in the database
- To be deleted from the database

4.2 Using MySQL Cluster Connector for Java

This section provides basic information about building and running Java applications using MySQL Cluster Connector for Java.

4.2.1 Getting, Installing, and Setting Up MySQL Cluster Connector for Java

This section discusses how to obtain MySQL Cluster Connector for Java sources, binaries, compiling, installing, getting started.

Obtaining MySQL Cluster Connector for Java. You can obtain the most recent NDB Cluster release incorporating MySQL Cluster Connector for Java from downloads.mysql.com.

Building and installing MySQL Cluster Connector for Java from source. You can build and install ClusterJ, ClusterJPA, or both as part of building and installing NDB Cluster, which always requires you to configure the build using a `--with-plugins` option that causes `NDB` support to be included, such as `--with-plugins=ndbcluster` or `--with-plugins=max`. Other relevant plugin names that can be used with this option include the following:

- `clusterj`: Required for building NDB Cluster with ClusterJ support.

- `openjpa`: Required for building NDB Cluster with ClusterJPA support.

In addition, you should use the following two `configure` options when configuring a build to include ClusterJ, ClusterJPA, or both:

- `--with-classpath=path`: Required for building NDB Cluster with ClusterJPA support. *`path`* must include the path or paths to the OpenJPA libraries and OpenJPA dependencies on your system.
- `--with-extra-charsets`: ClusterJ uses the `ucs2` character set for internal storage, and ClusterJ cannot be built without it. The simplest way to ensure that this character set is available is to configure using `--with-extra-charsets=all`. This is what we recommend that you use, unless you have some reason for not wanting to include all character sets, in which case you should make sure that `ucs2` is specified in the character set list passed to this option.

A typical configuration step in building NDB Cluster with support for both ClusterJ and ClusterJPA might look like this:

```
shell> ./configure --with-plugins=ndbcluster,clusterj,openjpa \
  --with-extra-charsets=all  \
  --with-classpath=path/to/openjpa/libs \
  --prefix=path/to/install/directory
```

`path/to/openjpa/libs` must include the following:

- `openjpa-2.2.2.jar`
- `driver-5.1.10.jar` (This is the MySQL JDBC driver)
- `geronimo-jta_1.1_spec-1.1.jar`
- `geronimo-jpa_3.0_spec-1.0.jar`
- `serp-1.13.1.jar`
- `commons-lang-2.1.jar`
- `commons-collections-3.2.jar`

Note

Not all available options for configuring an NDB Cluster build are shown in this section. For information about other options that can be used, see Installing MySQL Using a Development Source Tree, or consult the output of `configure --help`.

After configuring the build, run `make` and `make install` as you normally would to compile and install the NDB Cluster software. Following installation, the MySQL Cluster Connector for Java jar files can be found in `share/mysql/java` under the MySQL installation directory (by default, this is `/usr/local/mysql/share/mysql/java`).

You can also use the included file `storage/ndb/clusterj/pom.xml` for building ClusterJ with Maven. Setting up your build environment to use Maven requires that you run `mvn install` in the `clusterj` directory. Assuming that you are already in the root directory of the NDB Cluster source tree, you can accomplish this by performing in your shell the steps shown here:

```
cd storage/ndb/clusterj
mvn install
```

This causes ClusterJ—including the `clusterj-jpa` plugin—to be built, with the resulting `.jar` files installed in the local repository.

MySQL Cluster Connector for Java jar files. After building and installing NDB Cluster with MySQL Cluster Connector for Java, you should find the following JAR files that are needed for using ClusterJ and ClusterJPA in `share/mysql/java/` under the NDB Cluster installation directory:

- `clusterj-api.jar`: This is the compile-time jar file, required for compiling ClusterJ application code.
- `clusterj.jar`: This is the runtime library required for executing ClusterJ applications.
- `clusterjpa.jar`: This is the runtime library required for executing ClusterJPA applications. This jar file must be in your classpath when running OpenJPA with ClusterJPA.

4.2.2 Using ClusterJ

This section provides basic information for writing, compiling, and executing applications that use ClusterJ. For the API documentation for ClusterJ, see Section 4.3, "ClusterJ API Reference".

Requirements. ClusterJ requires Java 1.5 or 1.6. *NDB Cluster must be compiled with ClusterJ support*; NDB Cluster binaries supplied by Oracle include ClusterJ support. If you are building NDB Cluster from source, see Building and installing MySQL Cluster Connector for Java from source, for information on configuring the build to enable ClusterJ support.

To compile applications that use ClusterJ, you need the `clusterj-api` jar file in your classpath. To run applications that use ClusterJ, you need the `clusterj` runtime jar file; in addition, `libndbclient` must be in the directory specified by `java.library.path`. Section 4.2.2.1, "Executing ClusterJ Applications and Sessions", provides more information about these requirements.

4.2.2.1 Executing ClusterJ Applications and Sessions

In this section, we discuss how to start ClusterJ applications and the ClusterJ application environment.

Executing a ClusterJ application. All of the ClusterJ jar files are normally found in `share/mysql/java/` in the MySQL installation directory. When executing a ClusterJ application, you must set the classpath to point to these files. In addition, you must set `java.library.path` variable to point to the directory containing the Cluster `ndbclient` library, normally found in `lib/mysql` (also in the MySQL installation directory). Thus you might execute a ClusterJ program `MyClusterJApp` in a manner similar to what is shown here:

```
shell> java -classpath /usr/local/mysql/share/mysql/java/clusterj.jar \
            -Djava.library.path=/usr/local/mysql/lib MyClusterJApp
```

Note

The precise locations of the ClusterJ jar files and of `libndbclient` depend on how the NDB Cluster software was installed. See Installation Layouts, for more information.

ClusterJ encourages you to use different jar files at compile time and runtime. This is to remove the ability of applications to access implementation artifacts accidentally. ClusterJ is intended to be independent of the NDB Cluster software version, whereas the `ndbclient` layer is version-specific. This makes it possible to maintain a stable API, so that applications written against it using a given NDB Cluster version continue to run following an upgrade of the cluster to a new version.

Getting the SessionFactory and getting a Session. `SessionFactory` is the source of all ClusterJ sessions that use a given NDB Cluster. Usually, there is only a single `SessionFactory` per NDB Cluster, per Java Virtual Machine.

`SessionFactory` can be configured by setting one or more properties. The preferred way to do this is by putting these in a properties file, like this:

```
com.mysql.clusterj.connectstring=localhost:1186
com.mysql.clusterj.database=mydb
```

The name of the properties file is arbitrary; howver, by convention, such files are named with a `.properties` extension. For ClusterJ applications, it is customary to name the file `clusterj.properties`.

After editing and saving the file, you can load the its contents into an instance of `Properties`, as shown here:

```
File propsFile = new File("clusterj.properties");
InputStream inStream = new FileInputStream(propsFile);
Properties props = new Properties();
props.load(inStream);
```

It is also possible to set these properties directly, without the use of a properties file:

```
Properties props = new Properties();

props.put("com.mysql.clusterj.connectstring", "localhost:1186");
props.put("com.mysql.clusterj.database", "mydb");
```

Once the properties have been set and loaded (using either of the techniques just shown), you can obtain a `SessionFactory`, and then from that a `Session` instance. For this, you use the `SessionFactory`'s `getSession()` method, as shown here:

```
SessionFactory factory = ClusterJHelper.getSessionFactory(props);

Session session = factory.getSession();
```

It is usually sufficient to set and load the `com.mysql.clusterj.connectstring` and `com.mysql.clusterj.database` properties (and these properties, along with `com.mysql.clusterj.max.transactions`, cannot be changed after starting the `SessionFactory`). For a complete list of available `SessionFactory` properties and usual values, see com.mysql.clusterj.Constants.

Note

`Session` instances must not be shared among threads. Each thread in your application should use its own instance of `Session`.

For `com.mysql.clusterj.connectstring`, we use the default NDB Cluster connection string `localhost:1186` (see NDB Cluster Connection Strings, for more information). For the value of `com.mysql.clusterj.database`, we use `mydb` in this example, but this value can be the name of any database containing `NDB` tables. For a listing of all `SessionFactory` properties that can be set in this manner, see com.mysql.clusterj.Constants.

Reconnecting to an NDB Cluster. ClusterJ does not attempt to reconnect to an NDB Cluster once the connection is lost (`error 4009`) due to, for example, a network outage or the cluster going down. This, together with other errors, should be handled by the application with a common error handler. The handler needs to be able to detect and distinguish among three types of errors, and handle them accordingly:

- *Normal errors*: These are errors at the application level (for example, those to deal with duplicate key, foreign key constraint, or timeout). They should be handled in application-specific ways, and, if resolved, the application can continue with the transaction.
- *Unexpected errors*: These are failures to work with the cluster that cannot be accounted for by the conditions of the application, but are nonfatal. The application should close the ClusterJ session and reopen a new one.

- *Fatal errors*: These are fatal errors like, for example, `error 4009`, `Cluster Failure`, which indicates a network outage or the cluster going down. The application should close the session and notify the rest of the application to shut down and then restart the `SessionFactory`.

  The restarting of the `SessionFactory` can be an automatic application function or a manual intervention. In either case, the code should wait until all sessions have been closed (that is, the public method getConnectionPoolSessionCounts() in the `SessionFactory` interface returns zeros for all pooled connections). Then the `SessionFactory` can be closed and reopened, and applications can start to get sessions again.

Logging. ClusterJ uses Java logging. Here are some default settings for the ClusterJ logging, which are specified in the `logging.properties` file and can be modified there:

- Logging level is set at `INFO` for all classes.
- Using `java.util.logging.FileHandler` as the handler.
- Default level for `java.util.logging.FileHandler` is set at `FINEST`
- Using `java.util.logging.SimpleFormatter` as the formatter for the handler.
- Log files are put inside the `target` directory under the current working directory, and file names are, generally, in the pattern of `log`*`Num`*, where *`Num`* is a unique number for resolving file name conflicts (see the Java documentation for `java.util.logging.FileHandler` for details).

The `logging.properties` file is located by default in the current working directory, but the location can be changed by specifying the system property `java.util.logging.config.file` when you start Java.

4.2.2.2 Creating tables

ClusterJ's main purpose is to read, write, and update row data in an existing database, rather than to perform DDL. You can create the `employee` table that matches this interface, using the following `CREATE TABLE` statement, in a MySQL client application such as `mysql`.

```
CREATE TABLE employee (
    id INT NOT NULL PRIMARY KEY,
    first VARCHAR(64) DEFAULT NULL,
    last VARCHAR(64) DEFAULT NULL,
    municipality VARCHAR(64) DEFAULT NULL,
    started DATE DEFAULT NULL,
    ended DATE DEFAULT NULL,
    department INT NOT NULL DEFAULT 1,
    UNIQUE KEY idx_u_hash (last,first USING HASH,
    KEY idx_municipality (municipality)
) ENGINE=NDBCLUSTER;
```

Now that the table has been created in NDB Cluster, you can map a ClusterJ interface to it using annotations. We show you how to do this in the next section.

4.2.2.3 Annotations

In ClusterJ (as in JPA), annotations are used to describe how the interface is mapped to tables in a database. An annotated interface looks like this:

```
@PersistenceCapable(table="employee")
@Index(name="idx_uhash")
public interface Employee {

    @PrimaryKey
    int getId();
    void setId(int id);
```

```
    String getFirst();
    void setFirst(String first);
    String getLast();
    void setLast(String last);

    @Column(name="municipality")
    @Index(name="idx_municipality")
    String getCity();
    void setCity(String city);

    Date getStarted();
    void setStarted(Date date);

    Date getEnded();
    void setEnded(Date date);

    Integer getDepartment();
    void setDepartment(Integer department);
}
```

This interface maps seven columns: `id`, `first`, `last`, `municipality` `started`, `ended`, and `department`. The annotation `@PersistenceCapable(table="employee")` is used to let ClusterJ know which database table to map the `Employee` to (in this case, the `employee` table). The `@Column` annotation is used because the `city` property name implied by the `getCity()` and `setCity()` methods is different from the mapped column name `municipality`. The annotations `@PrimaryKey` and `@Index` inform ClusterJ about indexes in the database table.

The implementation of this interface is created dynamically by ClusterJ at runtime. When the `newInstance()` method is called, ClusterJ creates an implementation class for the `Employee` interface; this class stores the values in an internal object array.

ClusterJ does not require an annotation for every attribute. ClusterJ automatically detects the primary keys of tables; while there is an annotation in ClusterJ to permit the user to describe the primary keys of a table (see previous example), when specified, it is currently ignored. (The intended use of this annotation is for the generation of schemas from the domain object model interfaces, but this is not yet supported.)

The annotations themselves must be imported from the ClusterJ API. They can be found in package `com.mysql.clusterj.annotation`, and can be imported like this:

```
import com.mysql.clusterj.annotation.Column;
import com.mysql.clusterj.annotation.Index;
import com.mysql.clusterj.annotation.PersistenceCapable;
import com.mysql.clusterj.annotation.PrimaryKey;
```

4.2.2.4 ClusterJ Basic Operations

In this section, we describe how to perform operations basic to ClusterJ applications, including the following:

- Creating new instances, setting their properties, and saving them to the database
- Performing primary key lookups (reads)
- Updating existing rows and saving the changes to the database
- Deleting rows from the database
- Constructing and executing queries to fetch a set of rows meeting certain criteria from the database

Creating new rows. To insert a new row into the table, first create a new instance of `Employee`. This can be accomplished by calling the `Session` method `newInstance()`, as shown here:

```
Employee newEmployee = session.newInstance(Employee.class);
```

Set the `Employee` instance properties corresponding with the desired `employee` table columns. For example, the following sets the `id`, `firstName`, `lastName`, and `started` properties.

```
emp.setId(988);

newEmployee.setFirstName("John");
newEmployee.setLastName("Jones");

newEmployee.setStarted(new Date());
```

Once you are satisfied with the changes, you can persist the `Employee` instance, causing a new row containing the desired values to be inserted into the `employee` table, like this:

```
session.persist(newEmployee);
```

If autocommit is on, and a row with the same `id` as this instance of `Employee` already exists in the database, the `persist()` method fails. If autocommit is off and a row with the same `id` as this `Employee` instance already exists in the database, the `persist()` method succeeds but a subsequent `commit()` fails.

If you want the data to be saved even though the row already exists, use the `savePersistent()` method instead of the `persist()` method. The `savePersistent()` method updates an existing instance or creates a new instance as needed, without throwing an exception.

Values that you have not specified are stored with their Java default values (`0` for integral types, `0.0` for numeric types, and `null` for reference types).

Primary key lookups. You can find an existing row in an `NDB` table using the `Session`'s `find()` method, like this:

```
Employee theEmployee = session.find(Employee.class, 988);
```

This is equivalent to the primary key lookup query `SELECT * FROM employee WHERE id = 988`.

ClusterJ also supports compound primary keys. The `find()` method can take an object array as a key, where the components of the object array are used to represent the primary key columns in the order they were declared. In addition, queries are optimized to detect whether columns of the primary key are specified as part of the query criteria, and if so, a primary key lookup or scan is executed as a strategy to implement the query.

Note

ClusterJ also supports multiple column ordered btree and unique hash indexes. As with primary keys, if a query specifies values for ordered or unique index fields, ClusterJ optimizes the query to use the index for scanning the table.

NDB Cluster automatically spreads table data across multiple data nodes. For some operations—find, insert, delete, and update—it is more efficient to tell the cluster on which data node the data is physically located, and to have the transaction execute on that data node. ClusterJ automatically detects the partition key; if the operation can be optimized for a specific data node, ClusterJ automatically starts the transaction on that node.

Update and save a row. To update the value of a given column in the row that we just obtained as `theEmployee`, use the `set*()` method whose name corresponds to the name of that column. For example, to update the `started` date for this `Employee`, use the `Employee`'s `setStarted()` method, as shown here:

```
theEmployee.setStarted(new Date(getMillisFor(2010, 01, 04)));
```

Note

For convenience, we use in this example a method `getMillisFor()`, which is defined as shown here, in the file `AbstractClusterJModelTest.java` (found in the `storage/ndb/clusterj/clusterj-test/src/main/java/testsuite/clusterj` directory of the NDB Cluster source tree):

```
/** Convert year, month, day into milliseconds after the Epoch, UTC.
* Set hours, minutes, seconds, and milliseconds to zero.
* @param year the year
* @param month the month (0 for January)
* @param day the day of the month
* @return
*/

protected static long getMillisFor(int year, int month, int day) {
  Calendar calendar = Calendar.getInstance();
  calendar.clear();
  calendar.set(Calendar.YEAR, year);
  calendar.set(Calendar.MONTH, month);
  calendar.set(Calendar.DATE, day);
  calendar.set(Calendar.HOUR, 0);
  calendar.set(Calendar.MINUTE, 0);
  calendar.set(Calendar.SECOND, 0);
  calendar.set(Calendar.MILLISECOND, 0);
  long result = calendar.getTimeInMillis();
  return result;
}
```

See the indicated file for further information.

You can update additional columns by invoking other `Employee` setter methods, like this:

```
theEmployee.setDepartment(3);
```

To save the changed row back to the NDB Cluster database, use the `Session`'s `updatePersistent()` method, like this:

```
session.updatePersistent(theEmployee);
```

Deleting rows. You can delete a single row easily using the `deletePersistent()` method of `Session`. In this example, we find the employee whose ID is 13, then delete this row from the `employee` table:

```
Employee exEmployee = session.find(Employee.class, 13);

session.deletePersistent(exEmployee);'

System.out.println("Deleted employee named " + exEmployee.getFirst()
                + " " + exEmployee.getLast() + ".");
```

There also exists a method for deleting multiple rows, which provides two options:

1. Delete all rows from a table.

2. Delete an arbitrary collection of rows.

Both kinds of multi-row delete can be performed using the `deletePersistentAll()` method. The first variant of this method acts on a `Class`. For example, the following statement deletes all rows from the `employee` table and returns the number of rows deleted, as shown here:

```
int numberDeleted = session.deletePersistentAll(Employee);

System.out.println("There used to be "  + numberDeleted + " employees, but now there are none.");
```

The call to `deletePersistentAll()` just shown is equivalent to issuing the SQL statement `DELETE FROM employee` in the `mysql` client.

`deletePersistentAll()` can also be used to delete a collection of rows, as shown in this example:

```
//  Assemble the collection of rows to be deleted...

List<Employee> redundancies = new ArrayList<Employee>();

for (int i = 1000; i < 2000; i += 100) {
  Employee redundant = session.newInstance(Employee.class);
  redundant.setId(i);
  redundancies.add(redundant);
}

numberDeleted = session.deletePersistentAll(redundancies);

System.out.println("Deleted " + numberDeleted + " rows.");
```

It is not necessary to find the instances in the database before deleting them.

Writing queries. The ClusterJ `QueryBuilder` interface is used to instantiate queries. The process begins with obtaining an instance of `QueryBuilder`, which is supplied by the current `Session`; we can then obtain a `QueryDefinition`, as shown here:

```
QueryBuilder builder = session.getQueryBuilder();

QueryDomainType<Employee> domain = builder.createQueryDefinition(Employee.class);
```

This is then used to set a column for comparison by the query. Here, we show how to prepare a query that compares the value of the `employee` table's `department` column with the constant value `8`.

```
domain.where( domain.get("department").equal(domain.param("department") );

Query<Employee> query = session.createQuery(domain);

query.setParameter("department", 8);
```

To obtain the results from the query, invoke the `Query`'s `getResultList()` method, as shown here;

```
List<Employee> results = query.getResultList();
```

The return value is a `List` that you can iterate over to retrieve and process the rows in the usual manner.

Transactions. The `Transaction` interface can optionally be used to bound transactions, via the following methods:

- `begin()`: Begin a transaction.
- `commit()`: Commit a transaction.
- `rollback()`: Roll back a transaction.

It is also possible using `Transaction` to check whether the transaction is active (via the `isActive()` method, and to get and set a rollback-only flag (using `getRollbackOnly()` and `setRollbackOnly()`, respectively).

If you do not use the `Transaction` interface, methods in `Session` that affect the database—such as `persist()`, `deletePersistent()`, `updatePersistent()`, and so on—are automatically enclosed in a database transaction.

4.2.2.5 ClusterJ Mappings Between MySQL and Java Data Types

ClusterJ provides mappings for all of the common MySQL database types to Java types. Java object wrappers of primitive types should be mapped to nullable database columns.

Note

Since Java does not have native unsigned data types, `UNSIGNED` columns should be avoided in table schemas if possible.

Compatibility with JDBC mappings. ClusterJ is implemented so as to be bug-compatible with the JDBC driver in terms of mapping from Java types to the database. That is, if you use ClusterJ to store or retrieve data, you obtain the same value as if you used the JDBC driver directly or through JPA.

The following tables show the mappings used by ClusterJ between common Java data types and MySQL column types. Separate tables are provided for numeric, floating-point, and variable-width types.

Numeric types. This table shows the ClusterJ mappings between Java numeric data types and MySQL column types:

Java Data Type	MySQL Column Type
`boolean`, `Boolean`	`BIT(1)`
`byte`, `Byte`	`BIT(1)` to `BIT(8)`, `TINYINT`
`short`, `Short`	`BIT(1)` to `BIT(16)`, `SMALLINT`, `YEAR`
`int`, `Integer`	`BIT(1)` to `BIT(32)`, `INT`
`long`, `Long`	`BIT(1)` to `BIT(64)`, `BIGINT`, `BIGINT UNSIGNED`
`float`, `Float`	`FLOAT`
`double`, `Double`	`DOUBLE`
`java.math.BigDecimal`	`NUMERIC`, `DECIMAL`
`java.math.BigInteger`	`NUMERIC` (precision = 0), `DECIMAL` (precision = 0)

Date and time types. The following table shows the ClusterJ mappings between Java date and time data types and MySQL column types:

Java Data Type	MySQL Column Type
`Java.util.Date`	`DATETIME`, `TIMESTAMP`, `TIME`, `DATE`
`Java.sql.Date`	`DATE`
`Java.sql.Time`	`TIME`
`Java.sql.Timestamp`	`DATETIME`, `TIMESTAMP`

Note

ClusterJ maps the MySQL `YEAR` type to a Java `short` (or `java.lang.Short`), as shown in the first table in this section.

`java.util.Date` represents date and time similar to the way in which Unix does so, but with more precision and a larger range. Where Unix represents a point in time as a 32-bit signed number of seconds since the Unix Epoch (01

January 1970), Java uses a 64-bit signed number of milliseconds since the Epoch.

Variable-width types. The following table shows the ClusterJ mappings between Java data types and MySQL variable-width column types:

Java Data Type	MySQL Column Type
`String`	`CHAR`, `VARCHAR`, `TEXT`
`byte[]`	`BINARY`, `VARBINARY`, `BLOB`

Note

No translation binary data is performed when mapping from MySQL `BINARY`, `VARBINARY`, or `BLOB` column values to Java byte arrays. Data is presented to the application exactly as it is stored.

4.2.3 Using JPA with NDB Cluster

ClusterJPA is implemented as a plugin for OpenJPA. The best way to use ClusterJPA is to start with the standard configuration of OpenJPA with JDBC and MySQL Server. Once you know that this configuration works for your application, you can switch to the ClusterJ code path.

Compiling applications for ClusterJPA is the same as compiling them for OpenJPA. To do this, you must have the following jar files in your classpath:

- `openjpa-2.2.2.jar`
- `driver-5.1.10.jar` (This is the MySQL JDBC driver)
- `geronimo-jta_1.1_spec-1.1.jar`
- `geronimo-jpa_3.0_spec-1.0.jar`
- `serp-1.13.1.jar`
- `commons-lang-2.1.jar`
- `commons-collections-3.2.jar`

You must also have the OpenJPA jar files to run OpenJPA applications. To run them using ClusterJPA, you also need the `clusterj.jar` jar file in your classpath, as well as the MySQL Server JDBC connector jar file `mysql-connector-j.jar` (see Connector/J Installation), and your `java.library.path` must include the directory where `libndbclient` can be found.

You must also update the `persistence.xml` file, which selects the JPA implementation to be used by the application. The contents of a sample `persistence.xml` file are shown here (with the relevant portions shown in emphasized text):

```
<persistence-unit name="personnel" transaction-type="RESOURCE_LOCAL">
  <provider>
    org.apache.openjpa.persistence.PersistenceProviderImpl
  </provider>
  <class>com.mysql.clusterj.jpatest.model.Employee</class>
  <properties>
    <property name="openjpa.BrokerFactory" value="ndb"/>
    <property name="openjpa.ndb.connectString" value="localhost:1186"/>
    <property name="openjpa.ConnectionURL" value="jdbc:mysql://localhost:3306/test"/>
    <property name="openjpa.ConnectionDriverName" value="com.mysql.jdbc.Driver"/>
    <property name="openjpa.ConnectionRetainMode" value="transaction"/>
  </properties>
</persistence-unit>
```

To activate ClusterJPA, the property named `openjpa.BrokerFactory` must have the value `ndb`.

The name of the persistence unit is arbitrary; for this example, we have chosen `personnel`.

4.2.4 Using Connector/J with NDB Cluster

JDBC clients of an NDB Cluster data source, and using Connector/J 5.0.6 (or later), accept `jdbc:mysql:loadbalance://` URLs (see Driver/Datasource Class Names, URL Syntax and Configuration Properties for Connector/J), with which you can take advantage of the ability to connect with multiple MySQL servers to achieve load balancing and failover.

However, while Connector/J does not depend on the MySQL client libraries, it does require a connection to a MySQL Server, which ClusterJ does not. JDBC also does not provide any object mappings for database objects, properties, or operations, or any way to persist objects.

See MySQL Connector/J 5.1 Developer Guide, for general information about using Connector/J.

4.3 ClusterJ API Reference

The following sections contain specifications for ClusterJ packages, interfaces, classes, and methods.

4.3.1 com.mysql.clusterj

Provides classes and interfaces for using NDB Cluster directly from Java.

- A class for bootstrapping
- Interfaces for use in application programs
- Classes to define exceptions

This package contains three main groups of classes and interfaces:

4.3.1.1 Major Interfaces

ClusterJ provides these major interfaces for use by application programs: `com.mysql.clusterj.SessionFactory`, `com.mysql.clusterj.Session`, `com.mysql.clusterj.Transaction`, `com.mysql.clusterj.query.QueryBuilder`, and `com.mysql.clusterj.Query`. *Bootstrapping* The helper class `com.mysql.clusterj.ClusterJHelper` contains methods for creating the `com.mysql.clusterj.SessionFactory`. *Bootstrapping* is the process of identifying an NDB Cluster and obtaining the SessionFactory for use with the cluster. There is one SessionFactory per cluster per Java VM.

SessionFactory

The `com.mysql.clusterj.SessionFactory` is configured via properties, which identify the NDB Cluster that the application connects to:

- com.mysql.clusterj.connectstring identifies the ndb_mgmd host name and port
- com.mysql.clusterj.connect.retries is the number of retries when connecting
- com.mysql.clusterj.connect.delay is the delay in seconds between connection retries
- com.mysql.clusterj.connect.verbose tells whether to display a message to System.out while connecting
- com.mysql.clusterj.connect.timeout.before is the number of seconds to wait until the first node responds to a connect request
- com.mysql.clusterj.connect.timeout.after is the number of seconds to wait until the last node responds to a connect request

- com.mysql.clusterj.connect.database is the name of the database to use

```
File propsFile = new File("clusterj.properties");
InputStream inStream = new FileInputStream(propsFile);
Properties props = new Properties();
props.load(inStream);
SessionFactory sessionFactory = ClusterJHelper.getSessionFactory(props);
```

Session The `com.mysql.clusterj.Session` represents the user's individual connection to the cluster. It contains methods for:

- finding persistent instances by primary key
- persistent instance factory (newInstance)
- persistent instance life cycle management (persist, remove)
- getting the QueryBuilder
- getting the Transaction (currentTransaction)

```
Session session = sessionFactory.getSession();
Employee existing = session.find(Employee.class, 1);
if (existing != null) {
    session.remove(existing);
}
Employee newemp = session.newInstance(Employee.class);
newemp.initialize(2, "Craig", 15, 146000.00);
session.persist(newemp);
```

Transaction The `com.mysql.clusterj.Transaction` allows users to combine multiple operations into a single database transaction. It contains methods to:

- begin a unit of work
- commit changes from a unit of work
- roll back all changes made since the unit of work was begun
- mark a unit of work for rollback only
- get the rollback status of the current unit of work

```
Transaction tx = session.currentTransaction();
tx.begin();
Employee existing = session.find(Employee.class, 1);
Employee newemp = session.newInstance(Employee.class);
newemp.initialize(2, "Craig", 146000.00);
session.persist(newemp);
tx.commit();
```

QueryBuilder The `com.mysql.clusterj.query.QueryBuilder` allows users to build queries. It contains methods to:

- define the Domain Object Model to query
- compare properties with parameters using:
  - equal
  - lessThan
  - greaterThan
  - lessEqual

  - greaterEqual
  - between
  - in
- combine comparisons using "and", "or", and "not" operators

```
    QueryBuilder builder = session.getQueryBuilder();
    QueryDomainType<Employee> qemp = builder.createQueryDefinition(Employee.class);
    Predicate service = qemp.get("yearsOfService").greaterThan(qemp.param("service"));
    Predicate salary = qemp.get("salary").lessEqual(qemp.param("salaryCap"));
    qemp.where(service.and(salary));
    Query<Employee> query = session.createQuery(qemp);
    query.setParameter("service", 10);
    query.setParameter("salaryCap", 180000.00);
    List<Employee> results = query.getResultList();
```

4.3.1.2 ClusterJDatastoreException

ClusterJUserException represents a database error. The underlying cause of the exception is contained in the "cause".

Synopsis

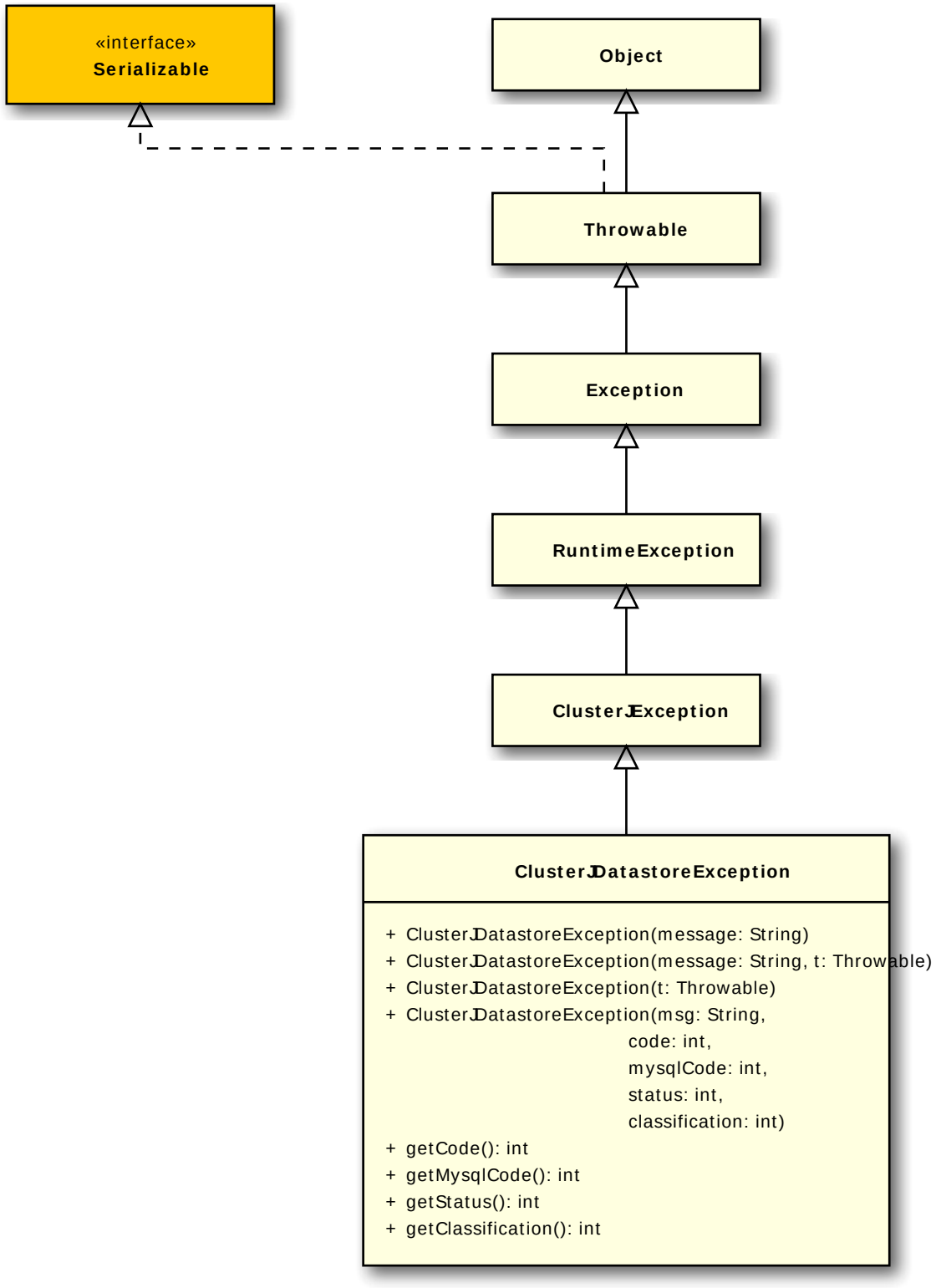
```
public class ClusterJDatastoreException,
  extends  ClusterJException {
// Public Constructors

  public ClusterJDatastoreException(
    String message);

  public ClusterJDatastoreException(
    String msg,
    int code,
    int mysqlCode,
    int status,
    int classification);

  public ClusterJDatastoreException(
    String message,
    Throwable t);

  public ClusterJDatastoreException(
    Throwable t);

// Public Methods

  public int getClassification();

  public int getCode();

  public int getMysqlCode();

  public int getStatus();

}
```

Methods inherited from com.mysql.clusterj.ClusterJException: `printStackTrace`

Methods inherited from java.lang.Throwable: `addSuppressed` , `fillInStackTrace` , `getCause` , `getLocalizedMessage` , `getMessage` , `getStackTrace` , `getSuppressed` , `initCause` , `setStackTrace` , `toString`

Methods inherited from java.lang.Object: `equals` , `getClass` , `hashCode` , `notify` , `notifyAll` , `wait`

Figure 4.1 ClusterJDatastoreException Class



getClassification()

```
public int getClassification();
```

Get the classification

getCode()

```
  public int getCode();
```

Get the code

Since 7.3.15, 7.4.13, 7.5.4

getMysqlCode()

```
  public int getMysqlCode();
```

Get the mysql code

Since 7.3.15, 7.4.13, 7.5.4

getStatus()

```
  public int getStatus();
```

Get the status

4.3.1.3 ClusterJDatastoreException.Classification

Helper class for getClassification(). import com.mysql.clusterj.ClusterJDatastoreException.Classification; Classification c = Classification.lookup(datastoreException.getClassification()); System.out.println("exceptionClassification " + c + " with value " + c.value);

Synopsis

```
public static final class ClusterJDatastoreException.Classification,
  extends  Enum<Classification> {
// Public Static Fields

  public static final Classification
    ApplicationError ;

  public static final Classification
    ConstraintViolation ;

  public static final Classification
    FunctionNotImplemented ;

  public static final Classification
    InsufficientSpace ;

  public static final Classification
    InternalError ;

  public static final Classification
    InternalTemporary ;

  public static final Classification
    NoDataFound ;

  public static final Classification
    NoError ;

  public static final Classification
    NodeRecoveryError ;

  public static final Classification
    NodeShutdown ;

  public static final Classification
    OverloadError ;

  public static final Classification
    SchemaError ;
```

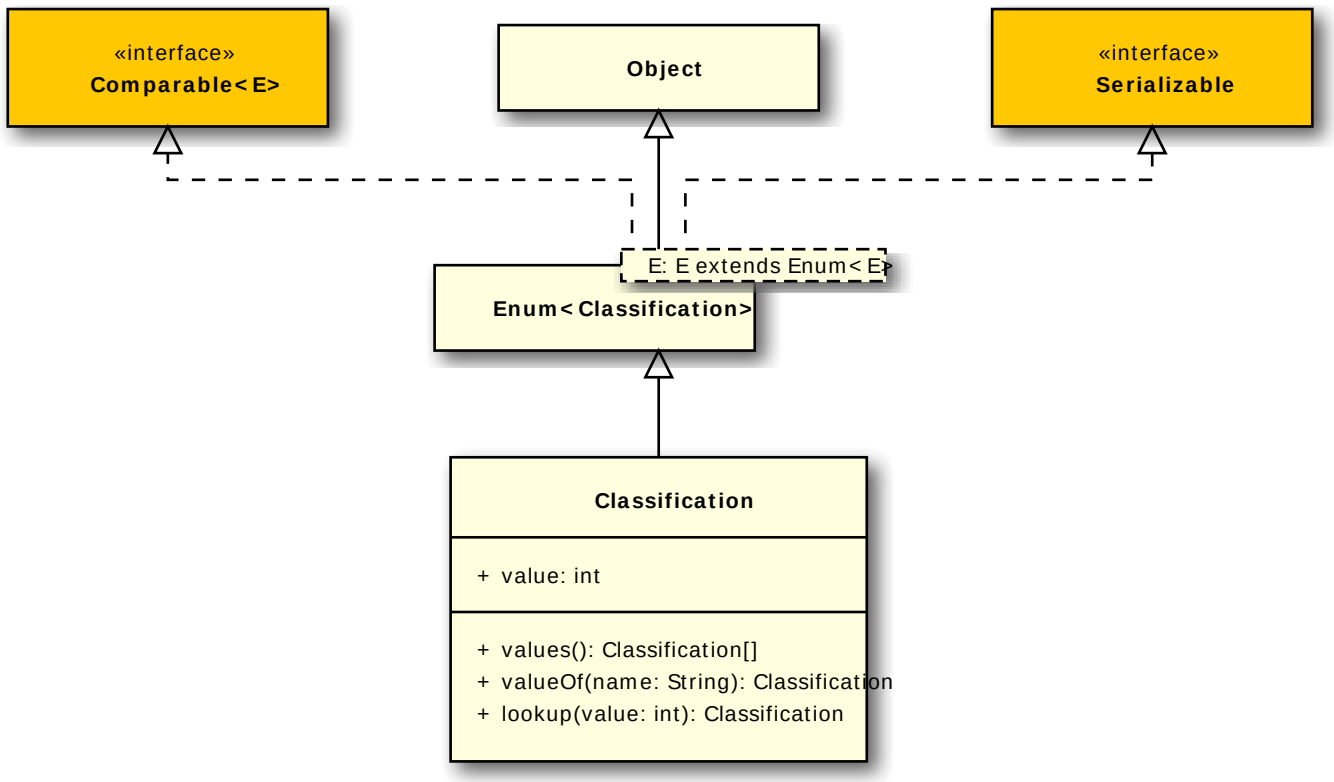
```
  public static final Classification
    SchemaObjectExists ;

  public static final Classification
    TemporaryResourceError ;

  public static final Classification
    TimeoutExpired ;

  public static final Classification
    UnknownErrorCode ;

  public static final Classification
    UnknownResultError ;

  public static final Classification
    UserDefinedError ;

// Public Static Methods

  public static Classification lookup(
    int value);

  public static Classification valueOf(
    String name);

  public static Classification[] values();

}
```

Methods inherited from java.lang.Enum: `compareTo` , `equals` , `getDeclaringClass` , `hashCode` , `name` , `ordinal` , `toString` , `valueOf`

Methods inherited from java.lang.Object: `getClass` , `notify` , `notifyAll` , `wait`

Since 7.3.15, 7.4.13, 7.5.4

Figure 4.2 ClusterJDatastoreException.Classification Class



lookup(int)

```
  public static Classification lookup(
    int value);
```

Get the Classification enum for a value returned by ClusterJDatastoreException.getClassification().

Parameters	
value	the classification returned by getClassification()
return	the Classification for the error

4.3.1.4 ClusterJException

ClusterJException is the base for all ClusterJ exceptions. Applications can catch ClusterJException to be notified of all ClusterJ reported issues.

- User exceptions are caused by user error, for example providing a connect string that refers to an unavailable host or port.
  - If a user exception is detected during bootstrapping (acquiring a SessionFactory), it is thrown as a fatal exception. `com.mysql.clusterj.ClusterJFatalUserException`
  - If an exception is detected during initialization of a persistent interface, for example annotating a column that doesn't exist in the mapped table, it is reported as a user exception. `com.mysql.clusterj.ClusterJUserException`
- Datastore exceptions report conditions that result from datastore operations after bootstrapping. For example, duplicate keys on insert, or record does not exist on delete. `com.mysql.clusterj.ClusterJDatastoreException`
- Internal exceptions report conditions that are caused by errors in implementation. These exceptions should be reported as bugs. `com.mysql.clusterj.ClusterJFatalInternalException`

Exceptions are in three general categories: User exceptions, Datastore exceptions, and Internal exceptions.

Synopsis

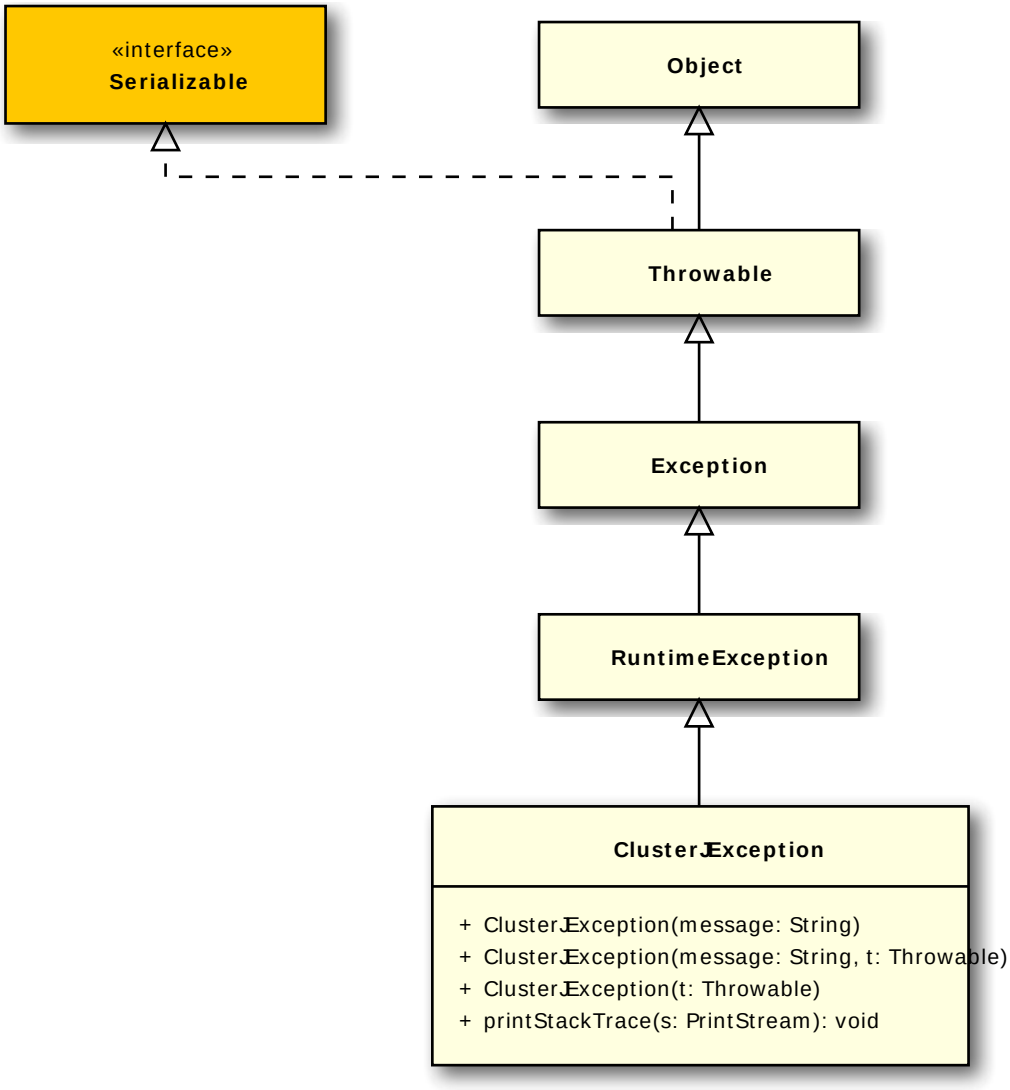
```
public class ClusterJException,
  extends  RuntimeException {
// Public Constructors

  public ClusterJException(
    String message);

  public ClusterJException(
    String message,
    Throwable t);

  public ClusterJException(
    Throwable t);

// Public Methods

  public synchronized void printStackTrace(
    PrintStream s);

}
```

Direct known subclasses: `com.mysql.clusterj.ClusterJDatastoreException` , `com.mysql.clusterj.ClusterJFatalException` , `com.mysql.clusterj.ClusterJUserException`

Methods inherited from java.lang.Throwable: `addSuppressed` , `fillInStackTrace` , `getCause` , `getLocalizedMessage` , `getMessage` , `getStackTrace` , `getSuppressed` , `initCause` , `printStackTrace` , `setStackTrace` , `toString`

Methods inherited from java.lang.Object: `equals` , `getClass` , `hashCode` , `notify` , `notifyAll` , `wait`

Figure 4.3 ClusterJException Class



4.3.1.5 ClusterJFatalException

ClusterJFatalException represents an exception that is not recoverable.

Synopsis

```
public class ClusterJFatalException,
  extends  ClusterJException {
// Public Constructors

  public ClusterJFatalException(
    String string);

  public ClusterJFatalException(
    String string,
    Throwable t);

  public ClusterJFatalException(
```

```
    Throwable t);

}
```

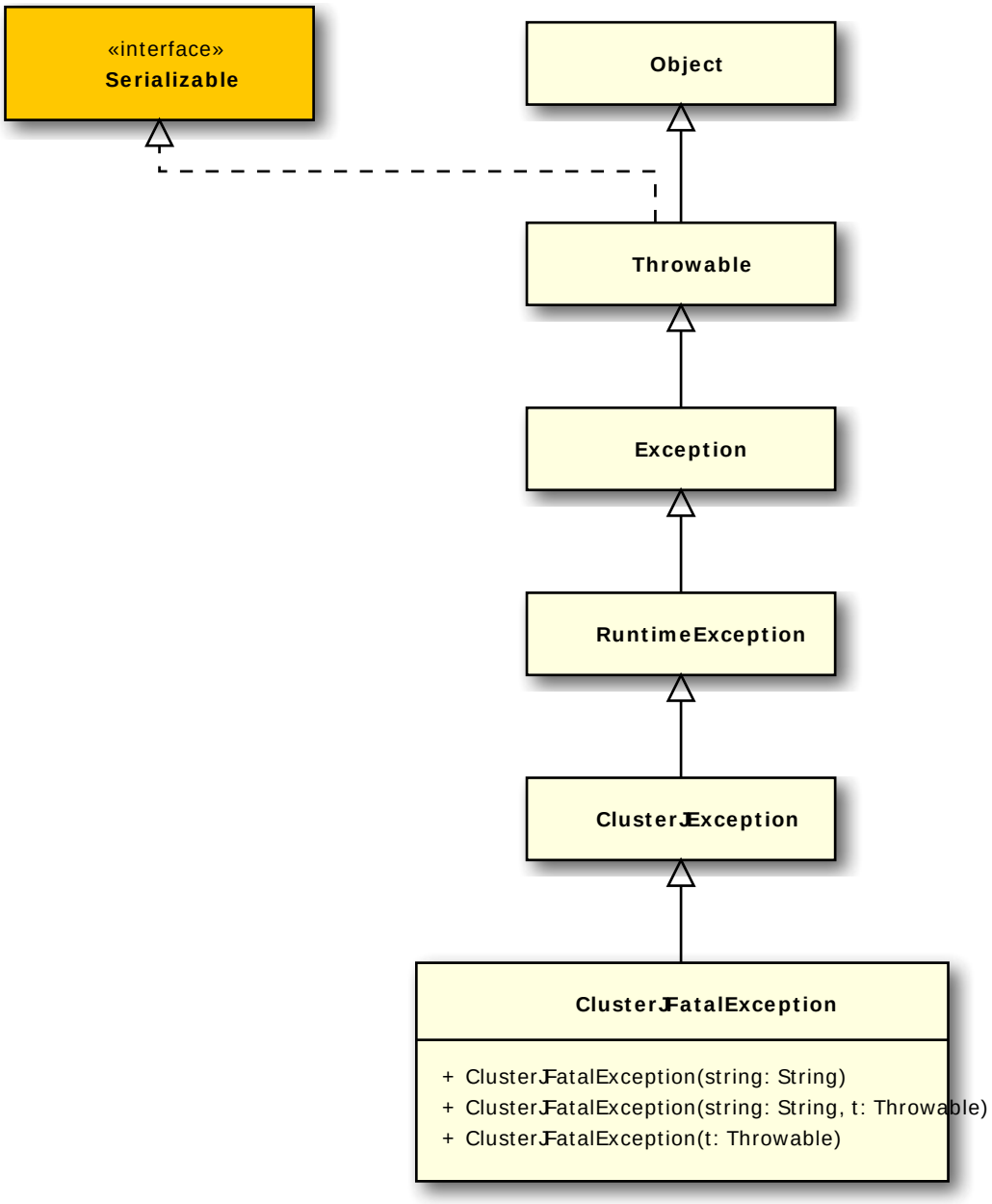
Direct known subclasses: `com.mysql.clusterj.ClusterJFatalInternalException` , `com.mysql.clusterj.ClusterJFatalUserException`

Methods inherited from com.mysql.clusterj.ClusterJException: `printStackTrace`

Methods inherited from java.lang.Throwable: `addSuppressed` , `fillInStackTrace` , `getCause` , `getLocalizedMessage` , `getMessage` , `getStackTrace` , `getSuppressed` , `initCause` , `setStackTrace` , `toString`

Methods inherited from java.lang.Object: `equals` , `getClass` , `hashCode` , `notify` , `notifyAll` , `wait`

Figure 4.4 ClusterJFatalException Class



4.3.1.6 ClusterJFatalInternalException

ClusterJFatalInternalException represents an implementation error that the user cannot recover from.

Synopsis

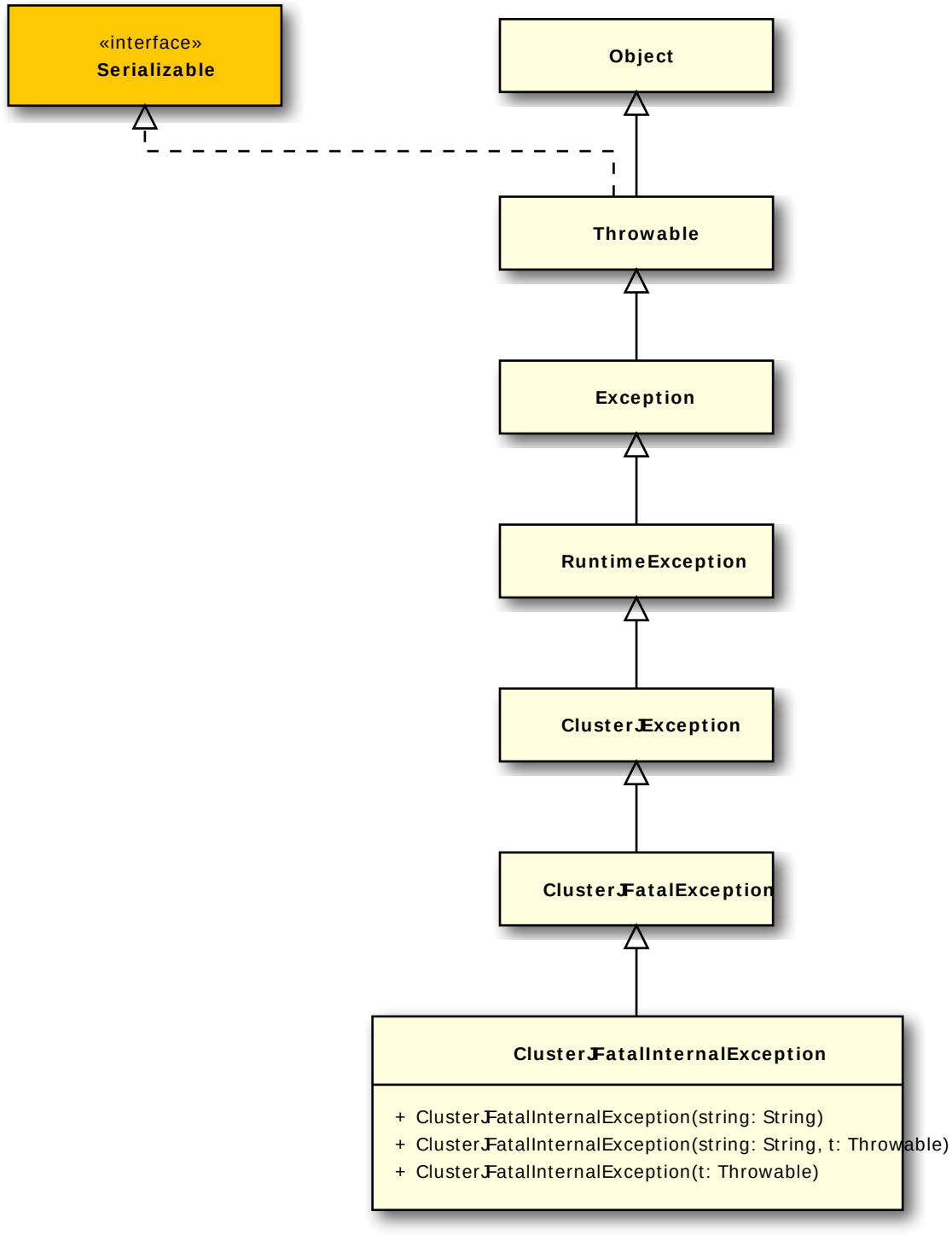
```
public class ClusterJFatalInternalException,
  extends  ClusterJFatalException {
// Public Constructors

  public ClusterJFatalInternalException(
    String string);

  public ClusterJFatalInternalException(
    String string,
    Throwable t);

  public ClusterJFatalInternalException(
    Throwable t);

}
```

Methods inherited from com.mysql.clusterj.ClusterJException: `printStackTrace`

Methods inherited from java.lang.Throwable: `addSuppressed` , `fillInStackTrace` , `getCause` , `getLocalizedMessage` , `getMessage` , `getStackTrace` , `getSuppressed` , `initCause` , `setStackTrace` , `toString`

Methods inherited from java.lang.Object: `equals` , `getClass` , `hashCode` , `notify` , `notifyAll` , `wait`

Figure 4.5 ClusterJFatalInternalException Class



4.3.1.7 ClusterJFatalUserException

ClusterJFatalUserException represents a user error that is unrecoverable, such as programming errors in persistent classes or missing resources in the execution environment.

Synopsis

```
public class ClusterJFatalUserException,
```

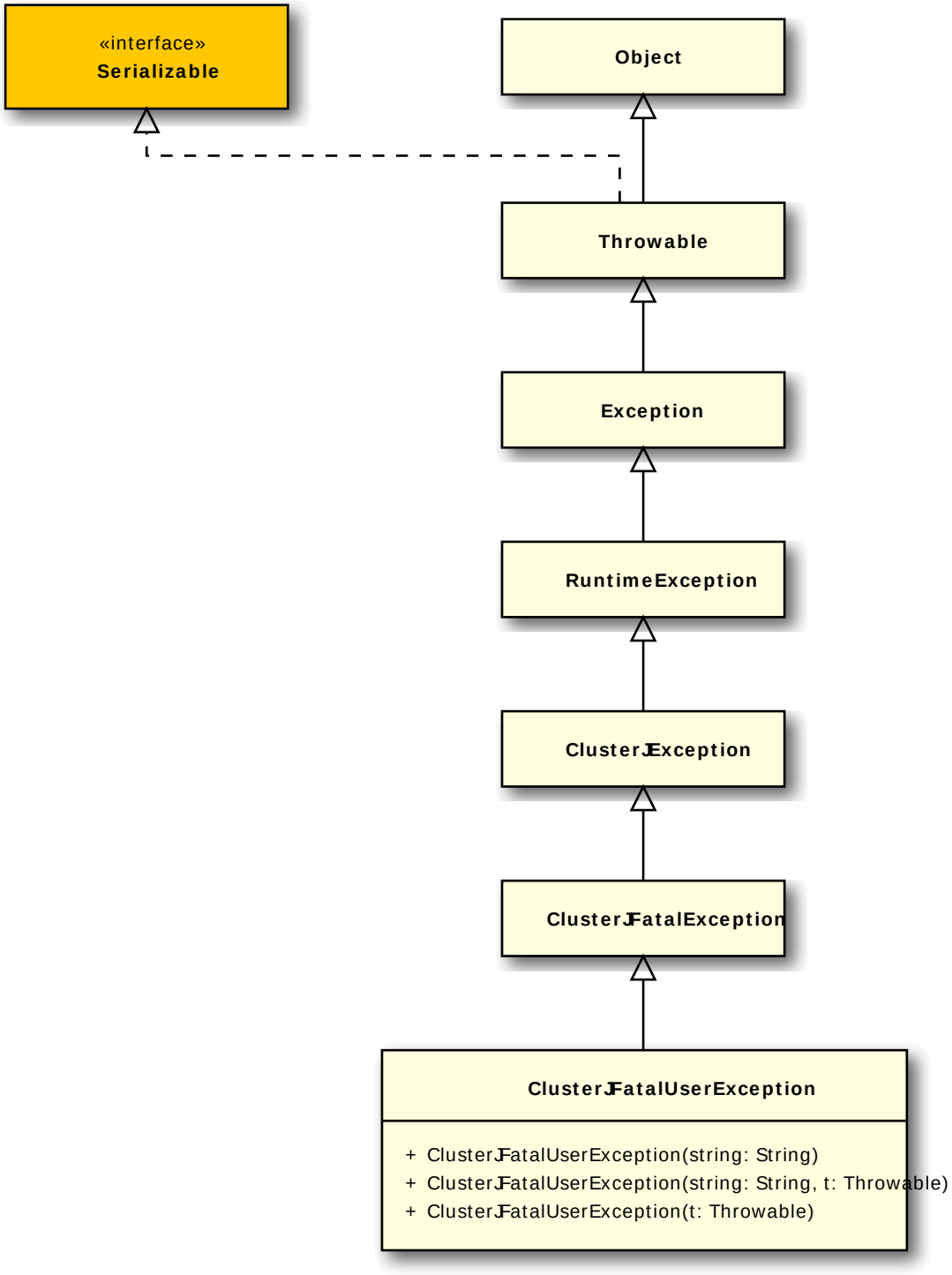
```
  extends  ClusterJFatalException {
// Public Constructors

  public ClusterJFatalUserException(
    String string);

  public ClusterJFatalUserException(
    String string,
    Throwable t);

  public ClusterJFatalUserException(
    Throwable t);

}
```

Methods inherited from com.mysql.clusterj.ClusterJException: `printStackTrace`

Methods inherited from java.lang.Throwable: `addSuppressed` , `fillInStackTrace` , `getCause` , `getLocalizedMessage` , `getMessage` , `getStackTrace` , `getSuppressed` , `initCause` , `setStackTrace` , `toString`

Methods inherited from java.lang.Object: `equals` , `getClass` , `hashCode` , `notify` , `notifyAll` , `wait`

Figure 4.6 ClusterJFatalUserException Class



4.3.1.8 ClusterJHelper

ClusterJHelper provides helper methods to bridge between the API and the implementation.

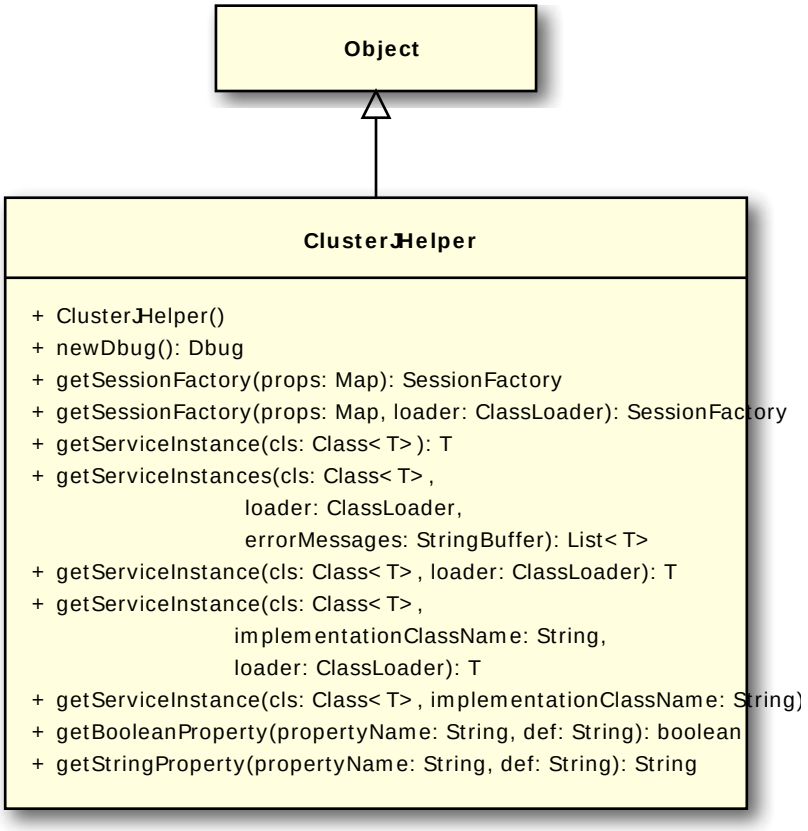
Synopsis

```
public class ClusterJHelper {
// Public Constructors
```

```
  public ClusterJHelper();

// Public Static Methods

  public static boolean getBooleanProperty(
    String propertyName,
    String def);

  public static T getServiceInstance(
    Class<T> cls);

  public static T getServiceInstance(
    Class<T> cls,
    ClassLoader loader);

  public static T getServiceInstance(
    Class<T> cls,
    String implementationClassName);

  public static T getServiceInstance(
    Class<T> cls,
    String implementationClassName,
    ClassLoader loader);

  public static List<T> getServiceInstances(
    Class<T> cls,
    ClassLoader loader,
    StringBuffer errorMessages);

  public static SessionFactory getSessionFactory(
    Map props);

  public static SessionFactory getSessionFactory(
    Map props,
    ClassLoader loader);

  public static String getStringProperty(
    String propertyName,
    String def);

  public static Dbug newDbug();

}
```

Methods inherited from java.lang.Object: `equals` , `getClass` , `hashCode` , `notify` , `notifyAll` , `toString` , `wait`

Figure 4.7 ClusterJHelper Class



getBooleanProperty(String, String)

```
public static boolean getBooleanProperty(
  String propertyName,
  String def);
```

Get the named boolean property from either the environment or system properties. If the property is not 'true' then return false.

Parameters	
propertyName	the name of the property
def	the default if the property is not set
return	the system property if it is set via -D or the system environment

getServiceInstance(Class<T>)

```
public static T getServiceInstance(
  Class<T> cls);
```

Locate a service implementation by services lookup of the context class loader.

Parameters	
cls	the class of the factory
return	the service instance

getServiceInstance(Class<T>, ClassLoader)

```
public static T getServiceInstance(
  Class<T> cls,
  ClassLoader loader);
```

Locate a service implementation for a service by services lookup of a specific class loader. The first service instance found is returned.

Parameters	
cls	the class of the factory
loader	the class loader for the factory implementation
return	the service instance

getServiceInstance(Class<T>, String)

```
public static T getServiceInstance(
  Class<T> cls,
  String implementationClassName);
```

Locate a service implementation for a service. If the implementation name is not null, use it instead of looking up. If the implementation class is not loadable or does not implement the interface, throw an exception. Use the ClusterJHelper class loader to find the service.

Parameters	
cls	
implementationClassName	
return	the implementation instance for a service

getServiceInstance(Class<T>, String, ClassLoader)

```
public static T getServiceInstance(
  Class<T> cls,
  String implementationClassName,
  ClassLoader loader);
```

Locate a service implementation for a service. If the implementation name is not null, use it instead of looking up. If the implementation class is not loadable or does not implement the interface, throw an exception.

Parameters	
cls	
implementationClassName	name of implementation class to load
loader	the ClassLoader to use to find the service
return	the implementation instance for a service

getServiceInstances(Class<T>, ClassLoader, StringBuffer)

```
public static List<T> getServiceInstances(
  Class<T> cls,
  ClassLoader loader,
  StringBuffer errorMessages);
```

Locate all service implementations by services lookup of a specific class loader. Implementations in the services file are instantiated and returned. Failed instantiations are remembered in the errorMessages buffer.

Parameters	
cls	the class of the factory
loader	the class loader for the factory implementation
errorMessages	a buffer used to hold the error messages
return	the service instance

getSessionFactory(Map)

```
public static SessionFactory getSessionFactory(
  Map props);
```

Locate a SessionFactory implementation by services lookup. The class loader used is the thread's context class loader.

Parameters	
props	properties of the session factory
return	the session factory

Exceptions

`ClusterFatalUserException` if the connection to the cluster cannot be made

getSessionFactory(Map, ClassLoader)

```
public static SessionFactory getSessionFactory(
  Map props,
  ClassLoader loader);
```

Locate a SessionFactory implementation by services lookup of a specific class loader. The properties are a Map that might contain implementation-specific properties plus standard properties.

Parameters	
props	the properties for the factory
loader	the class loader for the factory implementation
return	the session factory

Exceptions

`ClusterFatalUserException` if the connection to the cluster cannot be made

getStringProperty(String, String)

```
public static String getStringProperty(
  String propertyName,
  String def);
```

Get the named String property from either the environment or system properties.

Parameters	
propertyName	the name of the property
def	the default if the property is not set

return	the system property if it is set via -D or the system environment

newDbug()

```
  public static Dbug newDbug();
```

Return a new Dbug instance.

Parameters	
return	a new Dbug instance

4.3.1.9 ClusterJUserException

ClusterJUserException represents a user programming error.

Synopsis

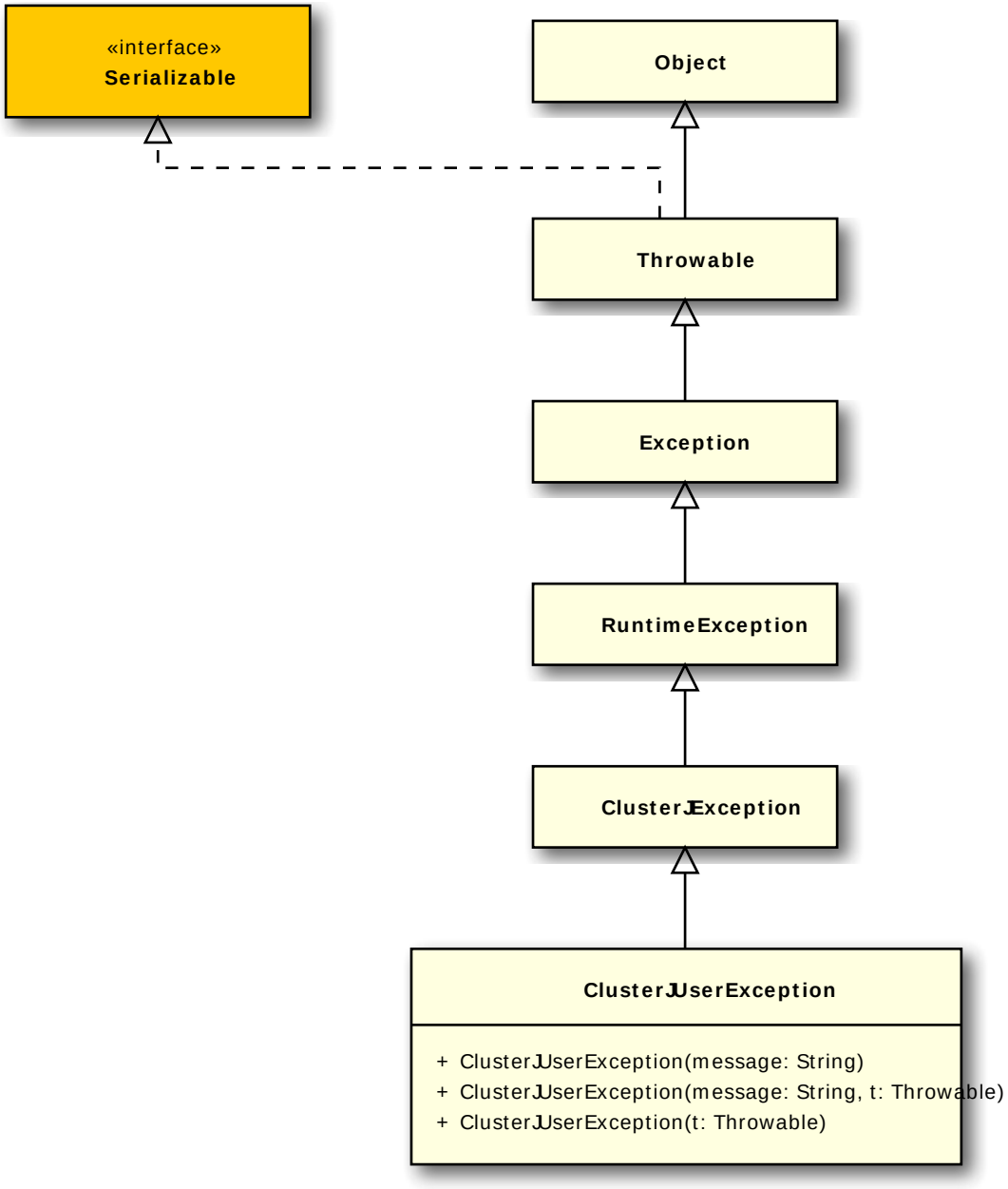
```
public class ClusterJUserException,
  extends  ClusterJException {
// Public Constructors

  public ClusterJUserException(
    String message);

  public ClusterJUserException(
    String message,
    Throwable t);

  public ClusterJUserException(
    Throwable t);

}
```

Methods inherited from com.mysql.clusterj.ClusterJException: `printStackTrace`

Methods inherited from java.lang.Throwable: `addSuppressed` , `fillInStackTrace` , `getCause` , `getLocalizedMessage` , `getMessage` , `getStackTrace` , `getSuppressed` , `initCause` , `setStackTrace` , `toString`

Methods inherited from java.lang.Object: `equals` , `getClass` , `hashCode` , `notify` , `notifyAll` , `wait`

Figure 4.8 ClusterJUserException Class

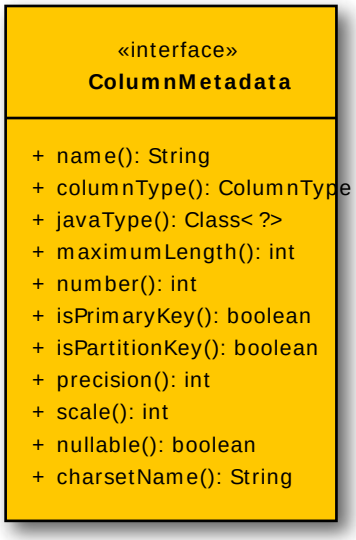


4.3.1.10 ColumnMetadata

```
public interface ColumnMetadata {
// Public Methods

  public abstract String charsetName();

  public abstract ColumnType columnType();

  public abstract boolean isPartitionKey();

  public abstract boolean isPrimaryKey();

  public abstract Class<?> javaType();

  public abstract int maximumLength();
```

```
  public abstract String name();

  public abstract boolean nullable();

  public abstract int number();

  public abstract int precision();

  public abstract int scale();

}
```

Figure 4.9 ColumnMetadata Class



charsetName()

```
  public abstract String charsetName();
```

Return the charset name.

Parameters	
return	the charset name

columnType()

```
  public abstract ColumnType columnType();
```

Return the type of the column.

Parameters	
return	the type of the column

isPartitionKey()

```
  public abstract boolean isPartitionKey();
```

Return whether this column is a partition key column.

Parameters	
return	true if this column is a partition key column

isPrimaryKey()

```
public abstract boolean isPrimaryKey();
```

Return whether this column is a primary key column.

Parameters	
return	true if this column is a primary key column

javaType()

```
public abstract Class<?> javaType();
```

Return the java type of the column.

Parameters	
return	the java type of the column

maximumLength()

```
public abstract int maximumLength();
```

Return the maximum number of bytes that can be stored in the column after translating the characters using the character set.

Parameters	
return	the maximum number of bytes that can be stored in the column

name()

```
public abstract String name();
```

Return the name of the column.

Parameters	
return	the name of the column

nullable()

```
public abstract boolean nullable();
```

Return whether this column is nullable.

Parameters	
return	whether this column is nullable

number()

```
public abstract int number();
```

Return the column number. This number is used as the first parameter in the get and set methods of DynamicColumn.

Parameters	
return	the column number.

precision()

```
  public abstract int precision();
```

Return the precision of the column.

Parameters	
return	the precision of the column

scale()

```
  public abstract int scale();
```

Return the scale of the column.

Parameters	
return	the scale of the column

4.3.1.11 ColumnType

This class enumerates the column types for columns in ndb.

Synopsis

```
public final class ColumnType,
  extends  Enum<ColumnType> {
// Public Static Fields

  public static final ColumnType
    Bigint ;

  public static final ColumnType
    Bigunsigned ;

  public static final ColumnType
    Binary ;

  public static final ColumnType
    Bit ;

  public static final ColumnType
    Blob ;

  public static final ColumnType
    Char ;

  public static final ColumnType
    Date ;

  public static final ColumnType
    Datetime ;

  public static final ColumnType
    Datetime2 ;

  public static final ColumnType
```

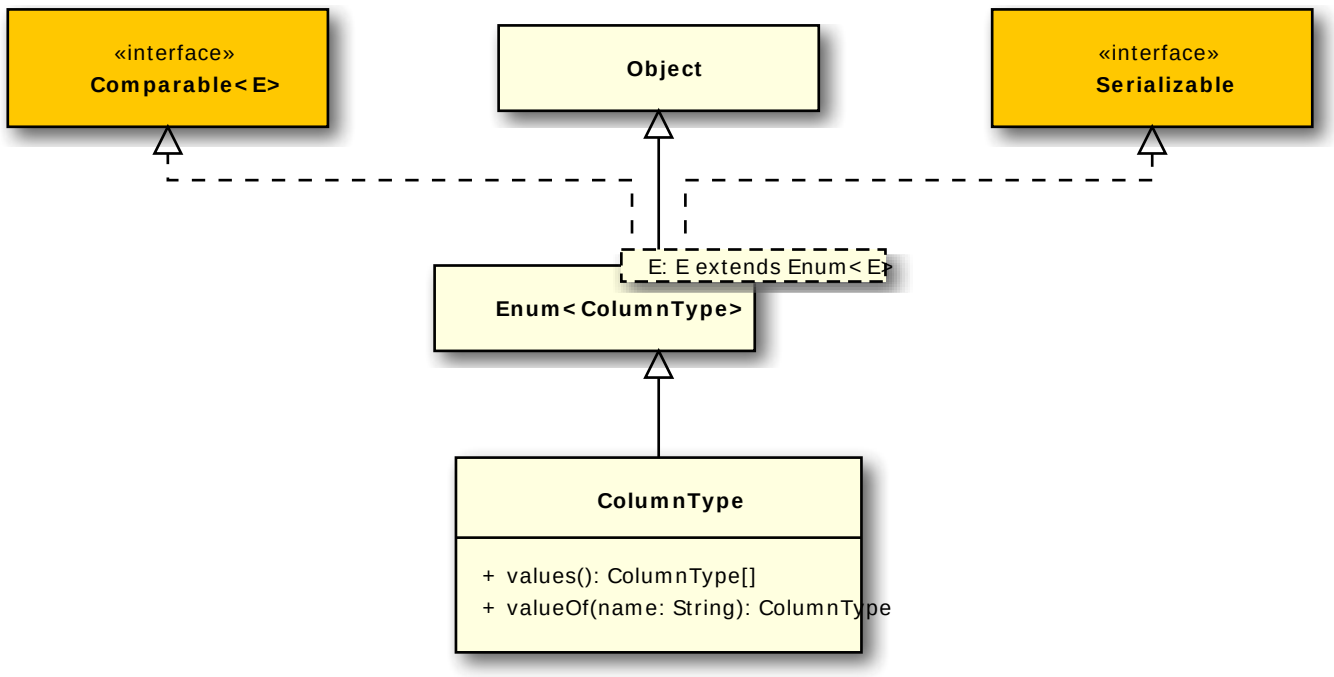
```
  Decimal ;

public static final ColumnType
  Decimalunsigned ;

public static final ColumnType
  Double ;

public static final ColumnType
  Float ;

public static final ColumnType
  Int ;

public static final ColumnType
  Longvarbinary ;

public static final ColumnType
  Longvarchar ;

public static final ColumnType
  Mediumint ;

public static final ColumnType
  Mediumunsigned ;

public static final ColumnType
  Olddecimal ;

public static final ColumnType
  Olddecimalunsigned ;

public static final ColumnType
  Smallint ;

public static final ColumnType
  Smallunsigned ;

public static final ColumnType
  Text ;

public static final ColumnType
  Time ;

public static final ColumnType
  Time2 ;

public static final ColumnType
  Timestamp ;

public static final ColumnType
  Timestamp2 ;

public static final ColumnType
  Tinyint ;

public static final ColumnType
  Tinyunsigned ;

public static final ColumnType
  Undefined ;

public static final ColumnType
  Unsigned ;

public static final ColumnType
  Varbinary ;

public static final ColumnType
```

```
    Varchar ;

  public static final ColumnType
    Year ;

// Public Static Methods

  public static ColumnType valueOf(
    String name);

  public static ColumnType[] values();

}
```

Methods inherited from java.lang.Enum: `compareTo` , `equals` , `getDeclaringClass` , `hashCode` , `name` , `ordinal` , `toString` , `valueOf`

Methods inherited from java.lang.Object: `getClass` , `notify` , `notifyAll` , `wait`

Figure 4.10 ColumnType Class



4.3.1.12 Constants

Constants used in the ClusterJ project.

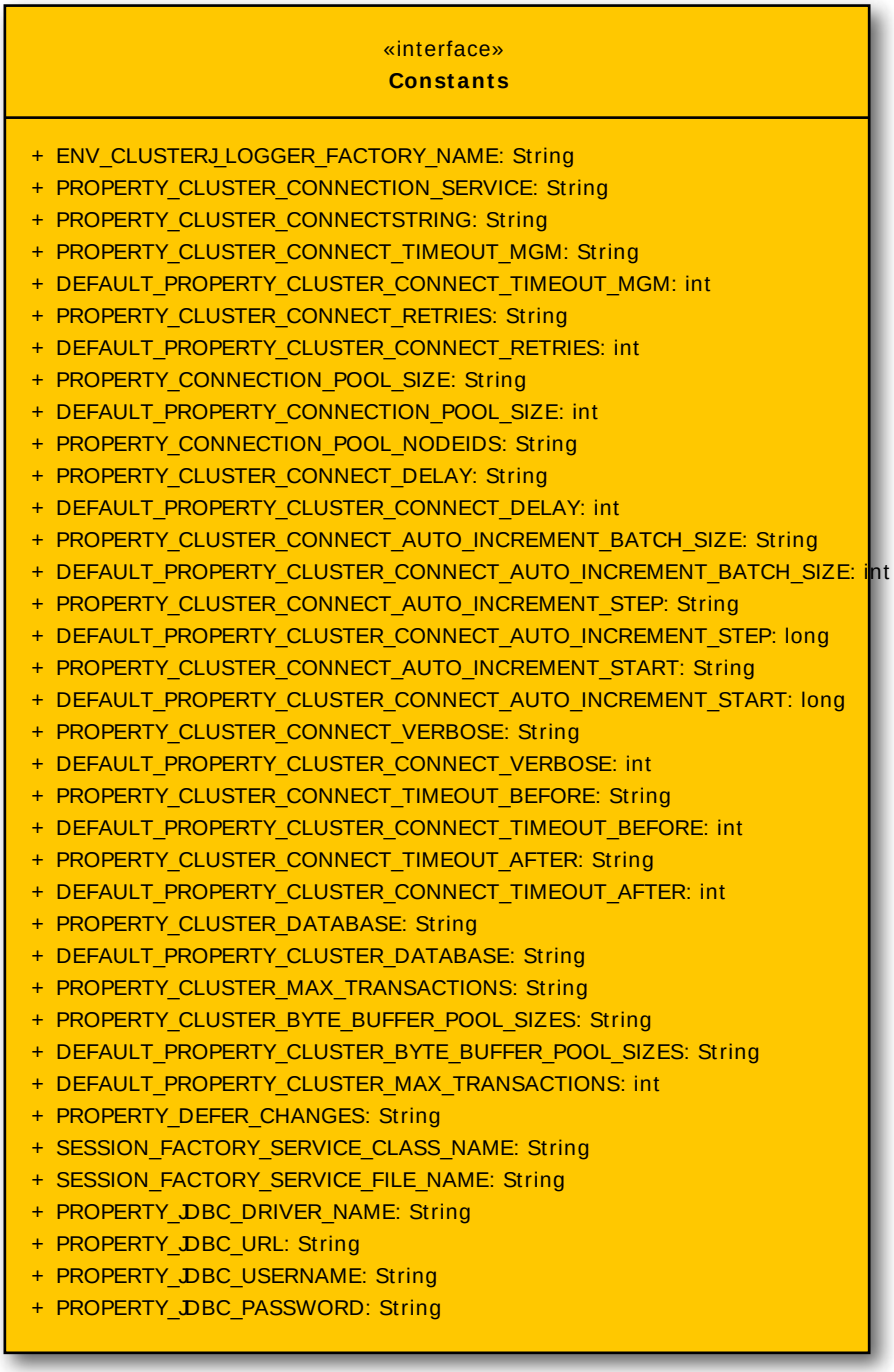
Synopsis

```
public interface Constants {
// Public Static Fields

  public static final String
    DEFAULT_PROPERTY_CLUSTER_BYTE_BUFFER_POOL_SIZES
        =  "256, 10240, 102400, 1048576";

  public static final int
    DEFAULT_PROPERTY_CLUSTER_CONNECT_AUTO_INCREMENT_BATCH_SIZE
        =  10;
```

```
public static final long
  DEFAULT_PROPERTY_CLUSTER_CONNECT_AUTO_INCREMENT_START
      =  1L;

public static final long
  DEFAULT_PROPERTY_CLUSTER_CONNECT_AUTO_INCREMENT_STEP
      =  1L;

public static final int
  DEFAULT_PROPERTY_CLUSTER_CONNECT_DELAY
      =  5;

public static final int
  DEFAULT_PROPERTY_CLUSTER_CONNECT_RETRIES
      =  4;

public static final int
  DEFAULT_PROPERTY_CLUSTER_CONNECT_TIMEOUT_AFTER
      =  20;

public static final int
  DEFAULT_PROPERTY_CLUSTER_CONNECT_TIMEOUT_BEFORE
      =  30;

public static final int
  DEFAULT_PROPERTY_CLUSTER_CONNECT_TIMEOUT_MGM
      =  30000;

public static final int
  DEFAULT_PROPERTY_CLUSTER_CONNECT_VERBOSE
      =  0;

public static final String
  DEFAULT_PROPERTY_CLUSTER_DATABASE
      =  "test";

public static final int
  DEFAULT_PROPERTY_CLUSTER_MAX_TRANSACTIONS
      =  4;

public static final int
  DEFAULT_PROPERTY_CONNECTION_POOL_SIZE
      =  1;

public static final String
  ENV_CLUSTERJ_LOGGER_FACTORY_NAME
      =  "CLUSTERJ_LOGGER_FACTORY";

public static final String
  PROPERTY_CLUSTER_BYTE_BUFFER_POOL_SIZES
      =  "com.mysql.clusterj.byte.buffer.pool.sizes";

public static final String
  PROPERTY_CLUSTER_CONNECTION_SERVICE
      =  "com.mysql.clusterj.connection.service";

public static final String
  PROPERTY_CLUSTER_CONNECTSTRING
      =  "com.mysql.clusterj.connectstring";

public static final String
  PROPERTY_CLUSTER_CONNECT_AUTO_INCREMENT_BATCH_SIZE
      =  "com.mysql.clusterj.connect.autoincrement.batchsize";

public static final String
  PROPERTY_CLUSTER_CONNECT_AUTO_INCREMENT_START
      =  "com.mysql.clusterj.connect.autoincrement.offset";

public static final String
  PROPERTY_CLUSTER_CONNECT_AUTO_INCREMENT_STEP
```

```
        =  "com.mysql.clusterj.connect.autoincrement.increment";

  public static final String
    PROPERTY_CLUSTER_CONNECT_DELAY
        =  "com.mysql.clusterj.connect.delay";

  public static final String
    PROPERTY_CLUSTER_CONNECT_RETRIES
        =  "com.mysql.clusterj.connect.retries";

  public static final String
    PROPERTY_CLUSTER_CONNECT_TIMEOUT_AFTER
        =  "com.mysql.clusterj.connect.timeout.after";

  public static final String
    PROPERTY_CLUSTER_CONNECT_TIMEOUT_BEFORE
        =  "com.mysql.clusterj.connect.timeout.before";

  public static final String
    PROPERTY_CLUSTER_CONNECT_TIMEOUT_MGM
        =  "com.mysql.clusterj.connect.timeout.mgm";

  public static final String
    PROPERTY_CLUSTER_CONNECT_VERBOSE
        =  "com.mysql.clusterj.connect.verbose";

  public static final String
    PROPERTY_CLUSTER_DATABASE
        =  "com.mysql.clusterj.database";

  public static final String
    PROPERTY_CLUSTER_MAX_TRANSACTIONS
        =  "com.mysql.clusterj.max.transactions";

  public static final String
    PROPERTY_CONNECTION_POOL_NODEIDS
        =  "com.mysql.clusterj.connection.pool.nodeids";

  public static final String
    PROPERTY_CONNECTION_POOL_SIZE
        =  "com.mysql.clusterj.connection.pool.size";

  public static final String
    PROPERTY_DEFER_CHANGES
        =  "com.mysql.clusterj.defer.changes";

  public static final String
    PROPERTY_JDBC_DRIVER_NAME
        =  "com.mysql.clusterj.jdbc.driver";

  public static final String
    PROPERTY_JDBC_PASSWORD
        =  "com.mysql.clusterj.jdbc.password";

  public static final String
    PROPERTY_JDBC_URL
        =  "com.mysql.clusterj.jdbc.url";

  public static final String
    PROPERTY_JDBC_USERNAME
        =  "com.mysql.clusterj.jdbc.username";

  public static final String
    SESSION_FACTORY_SERVICE_CLASS_NAME
        =  "com.mysql.clusterj.SessionFactoryService";

  public static final String
    SESSION_FACTORY_SERVICE_FILE_NAME
        =  "META-INF/services/com.mysql.clusterj.SessionFactoryService";

}
```

Figure 4.11 Constants Class



DEFAULT_PROPERTY_CLUSTER_BYTE_BUFFER_POOL_SIZES

```
public static final String
  DEFAULT_PROPERTY_CLUSTER_BYTE_BUFFER_POOL_SIZES
      =  "256, 10240, 102400, 1048576";
```

The default value of the byte buffer pool sizes property: 256, 10K, 100K, 1M

DEFAULT_PROPERTY_CLUSTER_CONNECT_AUTO_INCREMENT_BATCH_SIZE

```
public static final int
  DEFAULT_PROPERTY_CLUSTER_CONNECT_AUTO_INCREMENT_BATCH_SIZE
```

```
      =  10;
```

The default value of the connection delay property

DEFAULT_PROPERTY_CLUSTER_CONNECT_AUTO_INCREMENT_START

```
public static final long
  DEFAULT_PROPERTY_CLUSTER_CONNECT_AUTO_INCREMENT_START
      =  1L;
```

The default value of the connection autoincrement offset property

DEFAULT_PROPERTY_CLUSTER_CONNECT_AUTO_INCREMENT_STEP

```
public static final long
  DEFAULT_PROPERTY_CLUSTER_CONNECT_AUTO_INCREMENT_STEP
      =  1L;
```

The default value of the connection delay property

DEFAULT_PROPERTY_CLUSTER_CONNECT_DELAY

```
public static final int
  DEFAULT_PROPERTY_CLUSTER_CONNECT_DELAY
      =  5;
```

The default value of the connection delay property

DEFAULT_PROPERTY_CLUSTER_CONNECT_RETRIES

```
public static final int
  DEFAULT_PROPERTY_CLUSTER_CONNECT_RETRIES
      =  4;
```

The default value of the connection retries property

DEFAULT_PROPERTY_CLUSTER_CONNECT_TIMEOUT_AFTER

```
public static final int
  DEFAULT_PROPERTY_CLUSTER_CONNECT_TIMEOUT_AFTER
      =  20;
```

The default value of the connection timeout after property

DEFAULT_PROPERTY_CLUSTER_CONNECT_TIMEOUT_BEFORE

```
public static final int
  DEFAULT_PROPERTY_CLUSTER_CONNECT_TIMEOUT_BEFORE
      =  30;
```

The default value of the connection timeout before property

DEFAULT_PROPERTY_CLUSTER_CONNECT_TIMEOUT_MGM

```
public static final int
  DEFAULT_PROPERTY_CLUSTER_CONNECT_TIMEOUT_MGM
      =  30000;
```

The default value of the connection timeout mgm property

DEFAULT_PROPERTY_CLUSTER_CONNECT_VERBOSE

```
public static final int
  DEFAULT_PROPERTY_CLUSTER_CONNECT_VERBOSE
      =  0;
```

The default value of the connection verbose property

DEFAULT_PROPERTY_CLUSTER_DATABASE

```
public static final String
  DEFAULT_PROPERTY_CLUSTER_DATABASE
      =  "test";
```

The default value of the database property

DEFAULT_PROPERTY_CLUSTER_MAX_TRANSACTIONS

```
public static final int
  DEFAULT_PROPERTY_CLUSTER_MAX_TRANSACTIONS
      =  4;
```

The default value of the maximum number of transactions property

DEFAULT_PROPERTY_CONNECTION_POOL_SIZE

```
public static final int
  DEFAULT_PROPERTY_CONNECTION_POOL_SIZE
      =  1;
```

The default value of the connection pool size property

ENV_CLUSTERJ_LOGGER_FACTORY_NAME

```
public static final String
  ENV_CLUSTERJ_LOGGER_FACTORY_NAME
      =  "CLUSTERJ_LOGGER_FACTORY";
```

The name of the environment variable to set the logger factory

PROPERTY_CLUSTER_BYTE_BUFFER_POOL_SIZES

```
public static final String
  PROPERTY_CLUSTER_BYTE_BUFFER_POOL_SIZES
      =  "com.mysql.clusterj.byte.buffer.pool.sizes";
```

The name of the byte buffer pool sizes property. To disable buffer pooling for blob objects, set the value of this property to "1". With this setting, buffers will be allocated and freed (and cleaned if possible) immediately after being used for blob data transfer.

PROPERTY_CLUSTER_CONNECT_AUTO_INCREMENT_BATCH_SIZE

```
public static final String
  PROPERTY_CLUSTER_CONNECT_AUTO_INCREMENT_BATCH_SIZE
      =  "com.mysql.clusterj.connect.autoincrement.batchsize";
```

The name of the connection autoincrement batch size property. For details, see Ndb_cluster_connection::connect()

PROPERTY_CLUSTER_CONNECT_AUTO_INCREMENT_START

```
public static final String
  PROPERTY_CLUSTER_CONNECT_AUTO_INCREMENT_START
      =  "com.mysql.clusterj.connect.autoincrement.offset";
```

The name of the connection autoincrement offset property. For details, see Ndb_cluster_connection::connect()

PROPERTY_CLUSTER_CONNECT_AUTO_INCREMENT_STEP

```
public static final String
  PROPERTY_CLUSTER_CONNECT_AUTO_INCREMENT_STEP
      =  "com.mysql.clusterj.connect.autoincrement.increment";
```

The name of the connection autoincrement increment property. For details, see Ndb_cluster_connection::connect()

PROPERTY_CLUSTER_CONNECT_DELAY

```
public static final String
  PROPERTY_CLUSTER_CONNECT_DELAY
      =  "com.mysql.clusterj.connect.delay";
```

The name of the connection delay property. For details, see Ndb_cluster_connection::connect()

PROPERTY_CLUSTER_CONNECT_RETRIES

```
public static final String
  PROPERTY_CLUSTER_CONNECT_RETRIES
      =  "com.mysql.clusterj.connect.retries";
```

The name of the connection retries property. For details, see Ndb_cluster_connection::connect()

PROPERTY_CLUSTER_CONNECT_TIMEOUT_AFTER

```
public static final String
  PROPERTY_CLUSTER_CONNECT_TIMEOUT_AFTER
      =  "com.mysql.clusterj.connect.timeout.after";
```

The name of the connection timeout after property. For details, see Ndb_cluster_connection::wait_until_ready()

PROPERTY_CLUSTER_CONNECT_TIMEOUT_BEFORE

```
public static final String
  PROPERTY_CLUSTER_CONNECT_TIMEOUT_BEFORE
      =  "com.mysql.clusterj.connect.timeout.before";
```

The name of the connection timeout before property. For details, see Ndb_cluster_connection::wait_until_ready()

PROPERTY_CLUSTER_CONNECT_TIMEOUT_MGM

```
public static final String
  PROPERTY_CLUSTER_CONNECT_TIMEOUT_MGM
      =  "com.mysql.clusterj.connect.timeout.mgm";
```

The name of the initial timeout for cluster connection to connect to MGM before connecting to data nodes Ndb_cluster_connection::set_timeout()

PROPERTY_CLUSTER_CONNECT_VERBOSE

```
public static final String
  PROPERTY_CLUSTER_CONNECT_VERBOSE
      =  "com.mysql.clusterj.connect.verbose";
```

The name of the connection verbose property. For details, see Ndb_cluster_connection::connect()

PROPERTY_CLUSTER_CONNECTION_SERVICE

```
public static final String
  PROPERTY_CLUSTER_CONNECTION_SERVICE
      =  "com.mysql.clusterj.connection.service";
```

The name of the connection service property

PROPERTY_CLUSTER_CONNECTSTRING

```
public static final String
  PROPERTY_CLUSTER_CONNECTSTRING
      =  "com.mysql.clusterj.connectstring";
```

The name of the connection string property. For details, see Ndb_cluster_connection constructor

PROPERTY_CLUSTER_DATABASE

```
public static final String
```

```
  PROPERTY_CLUSTER_DATABASE
      =  "com.mysql.clusterj.database";
```

The name of the database property. For details, see the catalogName parameter in the Ndb constructor. Ndb constructor

PROPERTY_CLUSTER_MAX_TRANSACTIONS

```
public static final String
  PROPERTY_CLUSTER_MAX_TRANSACTIONS
      =  "com.mysql.clusterj.max.transactions";
```

The name of the maximum number of transactions property. For details, see Ndb::init()

PROPERTY_CONNECTION_POOL_NODEIDS

```
public static final String
  PROPERTY_CONNECTION_POOL_NODEIDS
      =  "com.mysql.clusterj.connection.pool.nodeids";
```

The name of the connection pool node ids property. There is no default. This is the list of node ids to force the connections to be assigned to specific node ids. If this property is specified and connection pool size is not the default, the number of node ids of the list must match the connection pool size, or the number of node ids must be 1 and node ids will be assigned to connections starting with the specified node id.

PROPERTY_CONNECTION_POOL_SIZE

```
public static final String
  PROPERTY_CONNECTION_POOL_SIZE
      =  "com.mysql.clusterj.connection.pool.size";
```

The name of the connection pool size property. This is the number of connections to create in the connection pool. The default is 1 (all sessions share the same connection; all requests for a SessionFactory with the same connect string and database will share a single SessionFactory). A setting of 0 disables pooling; each request for a SessionFactory will receive its own unique SessionFactory.

PROPERTY_DEFER_CHANGES

```
public static final String
  PROPERTY_DEFER_CHANGES
      =  "com.mysql.clusterj.defer.changes";
```

The flag for deferred inserts, deletes, and updates

PROPERTY_JDBC_DRIVER_NAME

```
public static final String
  PROPERTY_JDBC_DRIVER_NAME
      =  "com.mysql.clusterj.jdbc.driver";
```

The name of the jdbc driver

PROPERTY_JDBC_PASSWORD

```
public static final String
  PROPERTY_JDBC_PASSWORD
      =  "com.mysql.clusterj.jdbc.password";
```

The jdbc password

PROPERTY_JDBC_URL

```
public static final String
  PROPERTY_JDBC_URL
      =  "com.mysql.clusterj.jdbc.url";
```

The jdbc url

PROPERTY_JDBC_USERNAME

```
public static final String
  PROPERTY_JDBC_USERNAME
      =  "com.mysql.clusterj.jdbc.username";
```

The jdbc username

SESSION_FACTORY_SERVICE_CLASS_NAME

```
public static final String
  SESSION_FACTORY_SERVICE_CLASS_NAME
      =  "com.mysql.clusterj.SessionFactoryService";
```

The name of the session factory service interface

SESSION_FACTORY_SERVICE_FILE_NAME

```
public static final String
  SESSION_FACTORY_SERVICE_FILE_NAME
      =  "META-INF/services/com.mysql.clusterj.SessionFactoryService";
```

The name of the files with names of implementation classes for session factory service

4.3.1.13 Dbug

Dbug allows clusterj applications to enable the DBUG functionality in cluster ndbapi library. The dbug state is a control string that consists of flags separated by colons. Flags are:

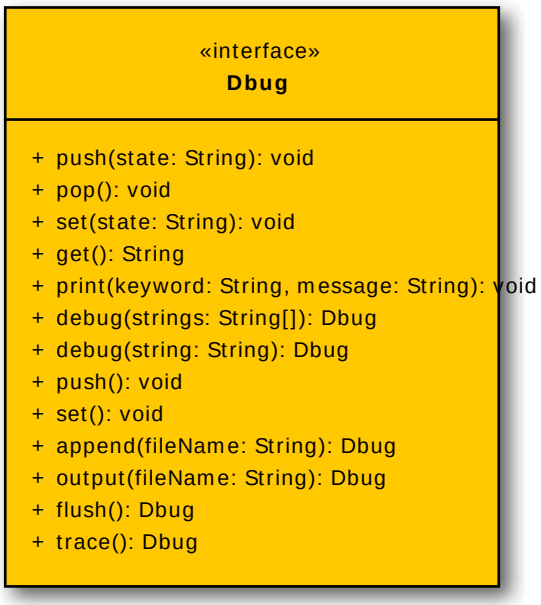
- d set the debug flag
- a[,filename] append debug output to the file
- A[,filename] like a[,filename] but flush the output after each operation
- d[,keyword[,keyword...]] enable output from macros with specified keywords
- D[,tenths] delay for specified tenths of a second after each operation
- f[,function[,function...]] limit output to the specified list of functions
- F mark each output with the file name of the source file
- i mark each output with the process id of the current process
- g[,function[,function...]] profile specified list of functions
- L mark each output with the line number of the source file
- n mark each output with the current function nesting depth
- N mark each output with a sequential number
- o[,filename] overwrite debug output to the file
- O[,filename] like o[,filename] but flush the output after each operation
- p[,pid[,pid...]] limit output to specified list of process ids
- P mark each output with the process name
- r reset the indentation level to zero
- t[,depth] limit function nesting to the specified depth
- T mark each output with the current timestamp

For example, the control string to trace calls and output debug information only for "jointx" and overwrite the contents of file "/tmp/dbug/jointx", use "t:d,jointx:o,/tmp/dbug/jointx". The above can be written as ClusterJHelper.newDbug().trace().debug("jointx").output("/tmp/dbug/jointx").set();

Synopsis

```
public interface Dbug {
// Public Methods

  public abstract Dbug append(
    String fileName);

  public abstract Dbug debug(
    String string);

  public abstract Dbug debug(
    String[] strings);

  public abstract Dbug flush();

  public abstract String get();

  public abstract Dbug output(
    String fileName);

  public abstract void pop();

  public abstract void print(
    String keyword,
    String message);

  public abstract void push();

  public abstract void push(
    String state);

  public abstract void set();

  public abstract void set(
    String state);

  public abstract Dbug trace();

}
```

Figure 4.12 Dbug Class



append(String)

```
public abstract Dbug append(
  String fileName);
```

Specify the file name for debug output (append).

Parameters	
fileName	the name of the file
return	this

debug(String)

```
public abstract Dbug debug(
  String string);
```

Set the list of debug keywords.

Parameters	
string	the comma separated debug keywords
return	this

debug(String[])

```
public abstract Dbug debug(
  String[] strings);
```

Set the list of debug keywords.

Parameters	
strings	the debug keywords
return	this

flush()

```
public abstract Dbug flush();
```

Force flush after each output operation.

Parameters	
return	this

get()

```
public abstract String get();
```

Return the current state.

Parameters	
return	the current state

output(String)

```
public abstract Dbug output(
  String fileName);
```

Specify the file name for debug output (overwrite).

Parameters	
fileName	the name of the file
return	this

pop()

```
public abstract void pop();
```

Pop the current state. The new state will be the previously pushed state.

print(String, String)

```
public abstract void print(
  String keyword,
  String message);
```

Print debug message.

push()

```
public abstract void push();
```

Push the current state as defined by the methods.

push(String)

```
public abstract void push(
  String state);
```

Push the current state and set the parameter as the new state.

Parameters	
state	the new state

set()

```
public abstract void set();
```

Set the current state as defined by the methods.

set(String)

```
public abstract void set(
  String state);
```

Set the current state from the parameter.

Parameters	
state	the new state

trace()

```
public abstract Dbug trace();
```

Set the trace flag.

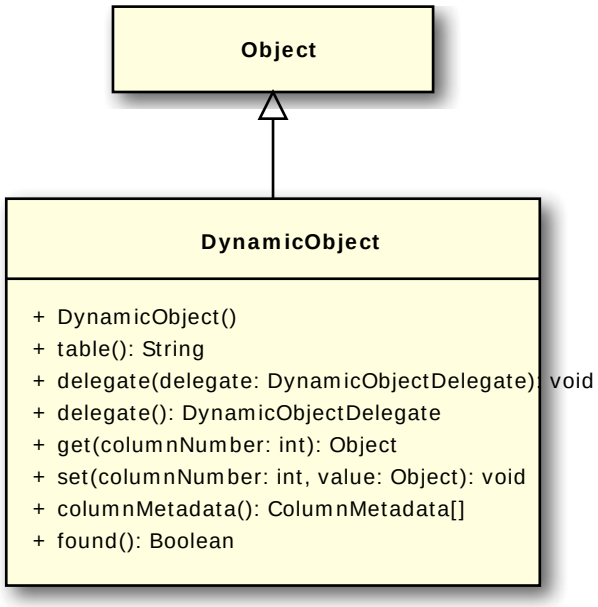
Parameters	
return	this

4.3.1.14 DynamicObject

```
public abstract class DynamicObject {
// Public Constructors

  public DynamicObject();

// Public Methods

  public final ColumnMetadata[] columnMetadata();

  public final DynamicObjectDelegate delegate();

  public final void delegate(
    DynamicObjectDelegate delegate);

  public Boolean found();

  public final Object get(
    int columnNumber);

  public final void set(
    int columnNumber,
    Object value);

  public String table();

}
```

Methods inherited from java.lang.Object: `equals` , `getClass` , `hashCode` , `notify` , `notifyAll` , `toString` , `wait`

Figure 4.13 DynamicObject Class

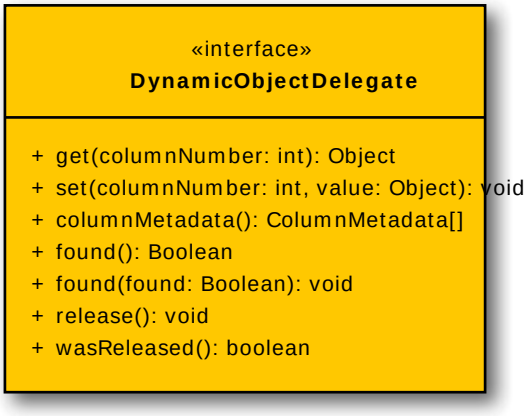


4.3.1.15 DynamicObjectDelegate

```
public interface DynamicObjectDelegate {
// Public Methods

  public abstract ColumnMetadata[] columnMetadata();

  public abstract Boolean found();

  public abstract void found(
```

```
    Boolean found);

  public abstract Object get(
    int columnNumber);

  public abstract void release();

  public abstract void set(
    int columnNumber,
    Object value);

  public abstract boolean wasReleased();

}
```

Figure 4.14 DynamicObjectDelegate Class



4.3.1.16 LockMode

Lock modes for read operations.

- SHARED: Set a shared lock on rows
- EXCLUSIVE: Set an exclusive lock on rows
- READ_COMMITTED: Set no locks but read the most recent committed values

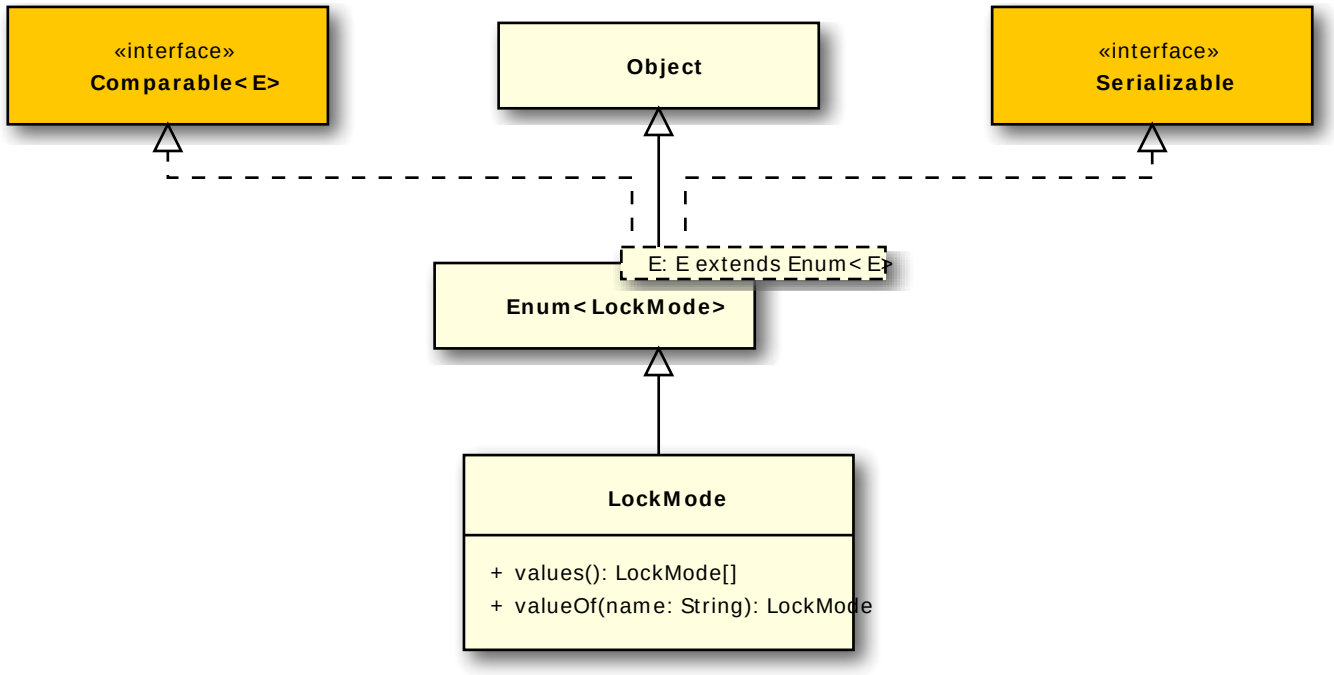
Synopsis

```
public final class LockMode,
  extends  Enum<LockMode> {
// Public Static Fields

  public static final LockMode
    EXCLUSIVE ;

  public static final LockMode
    READ_COMMITTED ;

  public static final LockMode
    SHARED ;

// Public Static Methods

  public static LockMode valueOf(
    String name);

  public static LockMode[] values();

}
```

Methods inherited from java.lang.Enum: `compareTo` , `equals` , `getDeclaringClass` , `hashCode` , `name` , `ordinal` , `toString` , `valueOf`

Methods inherited from java.lang.Object: `getClass` , `notify` , `notifyAll` , `wait`

Figure 4.15 LockMode Class



4.3.1.17 Query

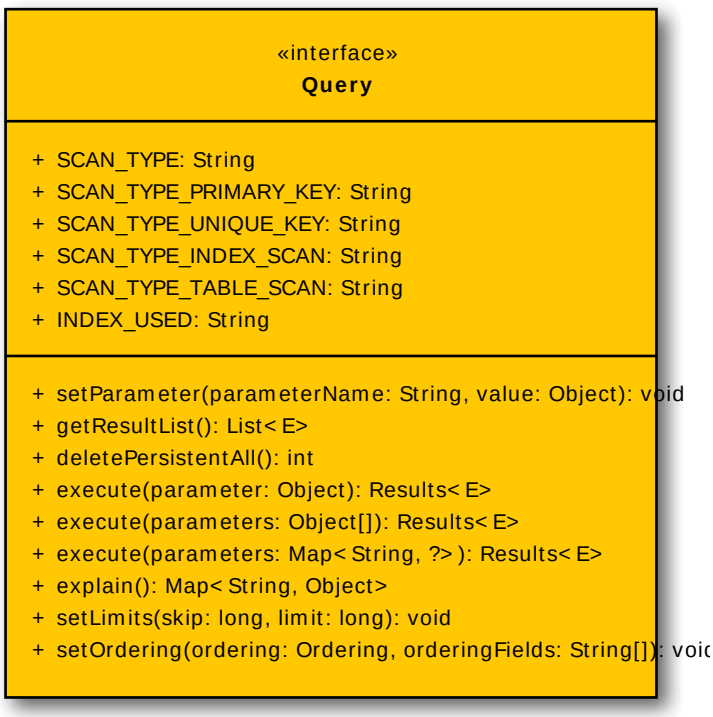
A Query instance represents a specific query with bound parameters. The instance is created by the method `com.mysql.clusterj.Session.<T>createQuery(com.mysql.clusterj.query.QueryDefinition`

Synopsis

```
public interface Query<E> {
// Public Static Fields

  public static final String
    INDEX_USED
        =  "IndexUsed";

  public static final String
    SCAN_TYPE
        =  "ScanType";

  public static final String
    SCAN_TYPE_INDEX_SCAN
        =  "INDEX_SCAN";

  public static final String
    SCAN_TYPE_PRIMARY_KEY
        =  "PRIMARY_KEY";

  public static final String
    SCAN_TYPE_TABLE_SCAN
        =  "TABLE_SCAN";

  public static final String
```

```
    SCAN_TYPE_UNIQUE_KEY
        =  "UNIQUE_KEY";

// Public Methods

  public abstract int deletePersistentAll();

  public abstract Results<E> execute(
    Object parameter);

  public abstract Results<E> execute(
    Object[] parameters);

  public abstract Results<E> execute(
    Map<String, ?> parameters);

  public abstract Map<String, Object> explain();

  public abstract List<E> getResultList();

  public abstract void setLimits(
    long skip,
    long limit);

  public abstract void setOrdering(
    Ordering ordering,
    String[] orderingFields);

  public abstract void setParameter(
    String parameterName,
    Object value);

}
```

Figure 4.16 Query Class



INDEX_USED

```
  public static final String
    INDEX_USED
```

```
      =  "IndexUsed";
```

The query explain index used key

SCAN_TYPE

```
public static final String
  SCAN_TYPE
      =  "ScanType";
```

The query explain scan type key

SCAN_TYPE_INDEX_SCAN

```
public static final String
  SCAN_TYPE_INDEX_SCAN
      =  "INDEX_SCAN";
```

The query explain scan type value for index scan

SCAN_TYPE_PRIMARY_KEY

```
public static final String
  SCAN_TYPE_PRIMARY_KEY
      =  "PRIMARY_KEY";
```

The query explain scan type value for primary key

SCAN_TYPE_TABLE_SCAN

```
public static final String
  SCAN_TYPE_TABLE_SCAN
      =  "TABLE_SCAN";
```

The query explain scan type value for table scan

SCAN_TYPE_UNIQUE_KEY

```
public static final String
  SCAN_TYPE_UNIQUE_KEY
      =  "UNIQUE_KEY";
```

The query explain scan type value for unique key

deletePersistentAll()

```
public abstract int deletePersistentAll();
```

Delete the instances that satisfy the query criteria.

Parameters	
return	the number of instances deleted

execute(Map<String, ?>)

```
public abstract Results<E> execute(
  Map<String, ?> parameters);
```

Execute the query with one or more named parameters. Parameters are resolved by name.

Parameters	
parameters	the parameters
return	the result

execute(Object...)

```
public abstract Results<E> execute(
  Object[] parameters);
```

Execute the query with one or more parameters. Parameters are resolved in the order they were declared in the query.

Parameters	
parameters	the parameters
return	the result

execute(Object)

```
public abstract Results<E> execute(
  Object parameter);
```

Execute the query with exactly one parameter.

Parameters	
parameter	the parameter
return	the result

explain()

```
public abstract Map<String, Object> explain();
```

Explain how this query will be or was executed. If called before binding all parameters, throws ClusterJUserException. Return a map of key:value pairs that explain how the query will be or was executed. Details can be obtained by calling toString on the value. The following keys are returned:

- ScanType: the type of scan, with values:
  - PRIMARY_KEY: the query used key lookup with the primary key
  - UNIQUE_KEY: the query used key lookup with a unique key
  - INDEX_SCAN: the query used a range scan with a non-unique key
  - TABLE_SCAN: the query used a table scan
- IndexUsed: the name of the index used, if any

Parameters	
return	the data about the execution of this query

Exceptions

`ClusterJUserException` if not all parameters are bound

getResultList()

```
public abstract List<E> getResultList();
```

Get the results as a list.

Parameters	
return	the result

Exceptions

`ClusterJUserException`	if not all parameters are bound
`ClusterJDatastoreException`	if an exception is reported by the datastore

setLimits(long, long)

```
public abstract void setLimits(
  long skip,
  long limit);
```

Set limits on results to return. The execution of the query is modified to return only a subset of results. If the filter would normally return 100 instances, skip is set to 50, and limit is set to 40, then the first 50 results that would have been returned are skipped, the next 40 results are returned and the remaining 10 results are ignored.

Skip must be greater than or equal to 0. Limit must be greater than or equal to 0. Limits may not be used with deletePersistentAll.

Parameters	
skip	the number of results to skip
limit	the number of results to return after skipping; use Long.MAX_VALUE for no limit.

setOrdering(Query.Ordering, String...)

```
public abstract void setOrdering(
  Ordering ordering,
  String[] orderingFields);
```

Set ordering for the results of this query. The execution of the query is modified to use an index previously defined.

- There must be an index defined on the columns mapped to the ordering fields, in the order of the ordering fields.
- There must be no gaps in the ordering fields relative to the index.
- All ordering fields must be in the index, but not all fields in the index need be in the ordering fields.
- If an "in" predicate is used in the filter on a field in the ordering, it can only be used with the first field.
- If any of these conditions is violated, ClusterJUserException is thrown when the query is executed.

If an "in" predicate is used, each element in the parameter defines a separate range, and ordering is performed within that range. There may be a better (more efficient) index based on the filter, but specifying the ordering will force the query to use an index that contains the ordering fields.

Parameters	
ordering	either Ordering.ASCENDING or Ordering.DESCENDING
orderingFields	the fields to order by

setParameter(String, Object)

```
public abstract void setParameter(
  String parameterName,
  Object value);
```

Set the value of a parameter. If called multiple times for the same parameter, silently replace the value.

Parameters	
parameterName	the name of the parameter
value	the value for the parameter

4.3.1.18 Query.Ordering

Ordering

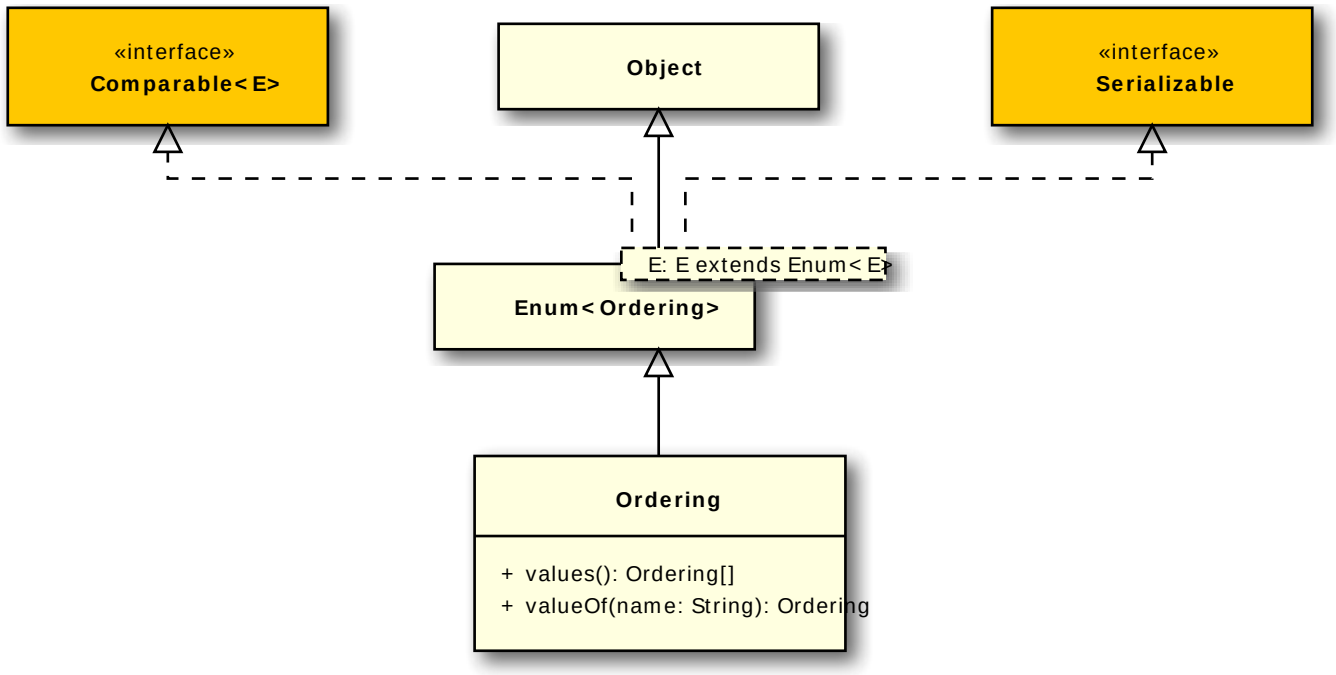
Synopsis

```
public static final class Query.Ordering,
  extends  Enum<Ordering> {
// Public Static Fields

  public static final Ordering
    ASCENDING ;

  public static final Ordering
    DESCENDING ;

// Public Static Methods

  public static Ordering valueOf(
    String name);

  public static Ordering[] values();

}
```

Methods inherited from java.lang.Enum: `compareTo` , `equals` , `getDeclaringClass` , `hashCode` , `name` , `ordinal` , `toString` , `valueOf`

Methods inherited from java.lang.Object: `getClass` , `notify` , `notifyAll` , `wait`

Figure 4.17 Query.Ordering Class



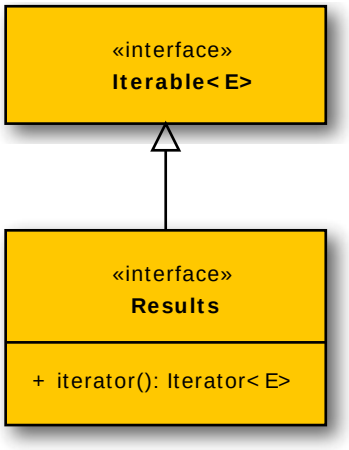
4.3.1.19 Results

Results of a query.

Synopsis

```
public interface Results<E>,
  extends  Iterable<E> {
// Public Methods

  public abstract Iterator<E> iterator();

}
```

Figure 4.18 Results Class



iterator()

```
  public abstract Iterator<E> iterator();
```

Specified by: Method `iterator` in interface `Iterable`

Get an iterator over the results of a query.

Parameters	
return	the iterator

4.3.1.20 Session

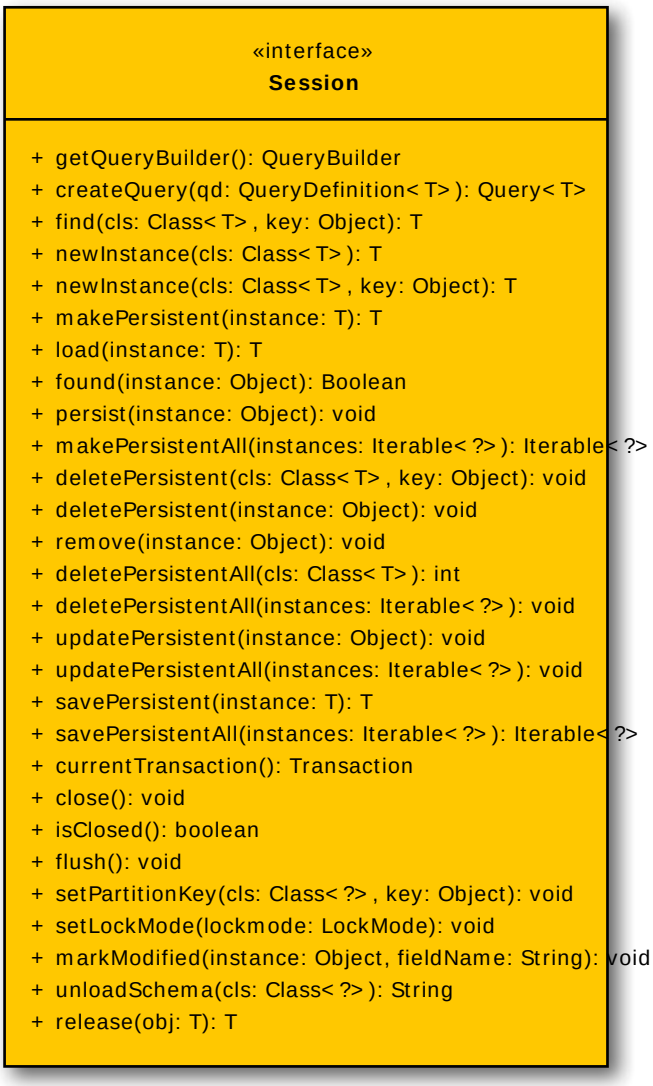
Session is the primary user interface to the cluster.

Synopsis

```
public interface Session {
// Public Methods

  public abstract void close();

  public abstract Query<T> createQuery(
    QueryDefinition<T> qd);

  public abstract Transaction currentTransaction();

  public abstract void deletePersistent(
    Class<T> cls,
    Object key);

  public abstract void deletePersistent(
    Object instance);
```

```
  public abstract int deletePersistentAll(
    Class<T> cls);

  public abstract void deletePersistentAll(
    Iterable<?> instances);

  public abstract T find(
    Class<T> cls,
    Object key);

  public abstract void flush();

  public abstract Boolean found(
    Object instance);

  public abstract QueryBuilder getQueryBuilder();

  public abstract boolean isClosed();

  public abstract T load(
    T instance);

  public abstract T makePersistent(
    T instance);

  public abstract Iterable<?> makePersistentAll(
    Iterable<?> instances);

  public abstract void markModified(
    Object instance,
    String fieldName);

  public abstract T newInstance(
    Class<T> cls);

  public abstract T newInstance(
    Class<T> cls,
    Object key);

  public abstract void persist(
    Object instance);

  public abstract T release(
    T obj);

  public abstract void remove(
    Object instance);

  public abstract T savePersistent(
    T instance);

  public abstract Iterable<?> savePersistentAll(
    Iterable<?> instances);

  public abstract void setLockMode(
    LockMode lockmode);

  public abstract void setPartitionKey(
    Class<?> cls,
    Object key);

  public abstract String unloadSchema(
    Class<?> cls);

  public abstract void updatePersistent(
    Object instance);

  public abstract void updatePersistentAll(
    Iterable<?> instances);

}
```

Figure 4.19 Session Class



close()

```
public abstract void close();
```

Close this session.

createQuery(QueryDefinition<T>)

```
public abstract Query<T> createQuery(
  QueryDefinition<T> qd);
```

Create a Query from a QueryDefinition.

Parameters	
qd	the query definition
return	the query instance

currentTransaction()

```
public abstract Transaction currentTransaction();
```

Get the current `com.mysql.clusterj.Transaction`.

Parameters	
return	the transaction

deletePersistent(Class<T>, Object)

```
public abstract void deletePersistent(
  Class<T> cls,
  Object key);
```

Delete an instance of a class from the database given its primary key. For single-column keys, the key parameter is a wrapper (e.g. Integer). For multi-column keys, the key parameter is an Object[] in which elements correspond to the primary keys in order as defined in the schema.

Parameters	
cls	the interface or dynamic class
key	the primary key

deletePersistent(Object)

```
public abstract void deletePersistent(
  Object instance);
```

Delete the instance from the database. Only the id field is used to determine which instance is to be deleted. If the instance does not exist in the database, an exception is thrown.

Parameters	
instance	the instance to delete

deletePersistentAll(Class<T>)

```
public abstract int deletePersistentAll(
  Class<T> cls);
```

Delete all instances of this class from the database. No exception is thrown even if there are no instances in the database.

Parameters	
cls	the interface or dynamic class
return	the number of instances deleted

deletePersistentAll(Iterable<?>)

```
public abstract void deletePersistentAll(
  Iterable<?> instances);
```

Delete all parameter instances from the database.

Parameters	
instances	the instances to delete

find(Class<T>, Object)

```
public abstract T find(
  Class<T> cls,
  Object key);
```

Find a specific instance by its primary key. The key must be of the same type as the primary key defined by the table corresponding to the cls parameter. The key parameter is the wrapped version

of the primitive type of the key, e.g. Integer for INT key types, Long for BIGINT key types, or String for char and varchar types. For multi-column primary keys, the key parameter is an Object[], each element of which is a component of the primary key. The elements must be in the order of declaration of the columns (not necessarily the order defined in the CONSTRAINT ... PRIMARY KEY clause) of the CREATE TABLE statement.

Parameters	
cls	the interface or dynamic class to find an instance of
key	the key of the instance to find
return	the instance of the interface or dynamic class with the specified key

flush()

```
public abstract void flush();
```

Flush deferred changes to the back end. Inserts, deletes, loads, and updates are sent to the back end.

found(Object)

```
public abstract Boolean found(
  Object instance);
```

Was the row corresponding to this instance found in the database?

Parameters	
instance	the instance corresponding to the row in the database
return	• null if the instance is null or was created via newInstance and never loaded; • true if the instance was returned from a find or query or created via newInstance and successfully loaded; • false if the instance was created via newInstance and not found.

See Also load(T) , newInstance(java.lang.Class<T>, java.lang.Object)

getQueryBuilder()

```
public abstract QueryBuilder getQueryBuilder();
```

Get a QueryBuilder.

Parameters	
return	the query builder

isClosed()

```
public abstract boolean isClosed();
```

Is this session closed?

Parameters	
return	true if the session is closed

load(T)

```
public abstract T load(
  T instance);
```

Load the instance from the database into memory. Loading is asynchronous and will be executed when an operation requiring database access is executed: find, flush, or query. The instance must have been returned from find or query; or created via session.newInstance and its primary key initialized.

Parameters	
instance	the instance to load
return	the instance

See Also found(java.lang.Object)

makePersistent(T)

```
public abstract T makePersistent(
  T instance);
```

Insert the instance into the database. If the instance already exists in the database, an exception is thrown.

Parameters	
instance	the instance to insert
return	the instance

See Also savePersistent(T)

makePersistentAll(Iterable<?>)

```
public abstract Iterable<?> makePersistentAll(
  Iterable<?> instances);
```

Insert the instances into the database.

Parameters	
instances	the instances to insert.
return	the instances

markModified(Object, String)

```
public abstract void markModified(
  Object instance,
  String fieldName);
```

Mark the field in the object as modified so it is flushed.

Parameters	
instance	the persistent instance
fieldName	the field to mark as modified

newInstance(Class<T>)

```
public abstract T newInstance(
  Class<T> cls);
```

Create an instance of an interface or dynamic class that maps to a table.

Parameters	
cls	the interface for which to create an instance
return	an instance that implements the interface

newInstance(Class<T>, Object)

```
public abstract T newInstance(
  Class<T> cls,
  Object key);
```

Create an instance of an interface or dynamic class that maps to a table and set the primary key of the new instance. The new instance can be used to create, delete, or update a record in the database.

Parameters	
cls	the interface for which to create an instance
return	an instance that implements the interface

persist(Object)

```
public abstract void persist(
  Object instance);
```

Insert the instance into the database. This method has identical semantics to makePersistent.

Parameters	
instance	the instance to insert

release(T)

```
public abstract T release(
  T obj);
```

Release resources associated with an instance. The instance must be a domain object obtained via session.newInstance(T.class), find(T.class), or query; or Iterable, or array T[]. Resources released can include direct buffers used to hold instance data. Released resources may be returned to a pool.

Parameters	
obj	a domain object of type T, an Iterable, or array T[]
return	the input parameter

Exceptions

`ClusterJUserException`	if the instance is not a domain object T, Iterable, or array T[], or if the object is used after calling this method.

remove(Object)

```
public abstract void remove(
  Object instance);
```

Delete the instance from the database. This method has identical semantics to deletePersistent.

Parameters	
instance	the instance to delete

savePersistent(T)

```
public abstract T savePersistent(
  T instance);
```

Save the instance in the database without checking for existence. The id field is used to determine which instance is to be saved. If the instance exists in the database it will be updated. If the instance does not exist, it will be created.

Parameters	
instance	the instance to update

savePersistentAll(Iterable<?>)

```
public abstract Iterable<?> savePersistentAll(
  Iterable<?> instances);
```

Update all parameter instances in the database.

Parameters	
instances	the instances to update

setLockMode(LockMode)

```
public abstract void setLockMode(
  LockMode lockmode);
```

Set the lock mode for read operations. This will take effect immediately and will remain in effect until this session is closed or this method is called again.

Parameters	
lockmode	the LockMode

setPartitionKey(Class<?>, Object)

```
public abstract void setPartitionKey(
  Class<?> cls,
  Object key);
```

Set the partition key for the next transaction. The key must be of the same type as the primary key defined by the table corresponding to the cls parameter. The key parameter is the wrapped version of the primitive type of the key, e.g. Integer for INT key types, Long for BIGINT key types, or String for char and varchar types. For multi-column primary keys, the key parameter is an Object[], each element of which is a component of the primary key. The elements must be in the order of declaration of the columns (not necessarily the order defined in the CONSTRAINT ... PRIMARY KEY clause) of the CREATE TABLE statement.

Parameters	
key	the primary key of the mapped table

Exceptions

`ClusterJUserException`	if a transaction is enlisted
`ClusterJUserException`	if a partition key is null
`ClusterJUserException`	if called twice in the same transaction
`ClusterJUserException`	if a partition key is the wrong type

unloadSchema(Class<?>)

```
public abstract String unloadSchema(
  Class<?> cls);
```

Unload the schema definition for a class. This must be done after the schema definition has changed in the database due to an alter table command. The next time the class is used the schema will be reloaded.

Parameters	
cls	the class for which the schema is unloaded
return	the name of the schema that was unloaded

updatePersistent(Object)

```
  public abstract void updatePersistent(
    Object instance);
```

Update the instance in the database without necessarily retrieving it. The id field is used to determine which instance is to be updated. If the instance does not exist in the database, an exception is thrown. This method cannot be used to change the primary key.

Parameters	
instance	the instance to update

updatePersistentAll(Iterable<?>)

```
  public abstract void updatePersistentAll(
    Iterable<?> instances);
```

Update all parameter instances in the database.

Parameters	
instances	the instances to update

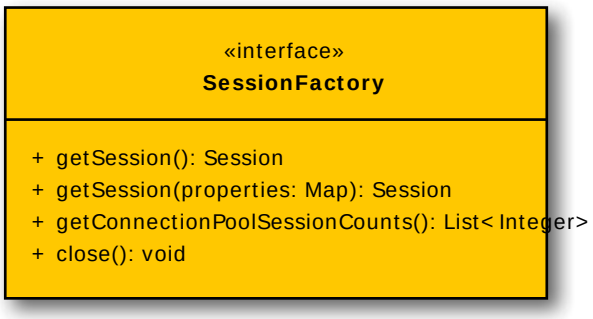
4.3.1.21 SessionFactory

SessionFactory represents a cluster.

Synopsis

```
public interface SessionFactory {
// Public Methods

  public abstract void close();

  public abstract List<Integer> getConnectionPoolSessionCounts();

  public abstract Session getSession();

  public abstract Session getSession(
    Map properties);

}
```

Figure 4.20 SessionFactory Class



close()

```
  public abstract void close();
```

Close this session factory. Release all resources.

getConnectionPoolSessionCounts()

```
  public abstract List<Integer> getConnectionPoolSessionCounts();
```

Get a list containing the number of open sessions for each connection in the connection pool.

Since 7.3.14, 7.4.12, 7.5.2

getSession()

```
  public abstract Session getSession();
```

Create a Session to use with the cluster, using all the properties of the SessionFactory.

Parameters	
return	the session

getSession(Map)

```
  public abstract Session getSession(
    Map properties);
```

Create a session to use with the cluster, overriding some properties. Properties PROPERTY_CLUSTER_CONNECTSTRING, PROPERTY_CLUSTER_DATABASE, and PROPERTY_CLUSTER_MAX_TRANSACTIONS may not be overridden.

Parameters	
properties	overriding some properties for this session
return	the session

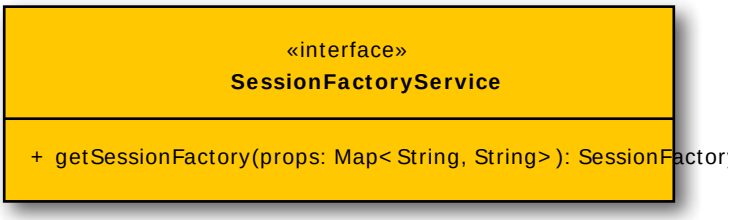
4.3.1.22 SessionFactoryService

This interface defines the service to create a SessionFactory from a Map<String, String> of properties.

Synopsis

```
public interface SessionFactoryService {
// Public Methods

  public abstract SessionFactory getSessionFactory(
    Map<String, String> props);

}
```

Figure 4.21 SessionFactoryService Class



getSessionFactory(Map<String, String>)

```
  public abstract SessionFactory getSessionFactory(
    Map<String, String> props);
```

Create or get a session factory. If a session factory with the same value for PROPERTY_CLUSTER_CONNECTSTRING has already been created in the VM, the existing factory is returned, regardless of whether other properties of the factory are the same as specified in the Map.

Parameters	
props	the properties for the session factory, in which the keys are defined in Constants and the values describe the environment
return	the session factory

See Also com.mysql.clusterj.Constants

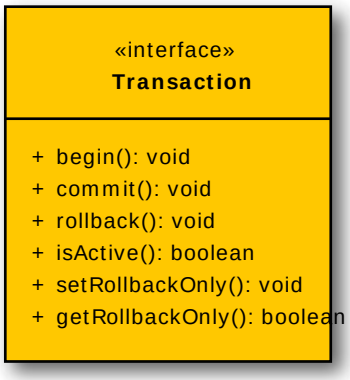
4.3.1.23 Transaction

Transaction represents a user transaction active in the cluster.

Synopsis

```
public interface Transaction {
// Public Methods

  public abstract void begin();

  public abstract void commit();

  public abstract boolean getRollbackOnly();

  public abstract boolean isActive();

  public abstract void rollback();

  public abstract void setRollbackOnly();

}
```

Figure 4.22 Transaction Class



begin()

```
  public abstract void begin();
```

Begin a transaction.

commit()

```
  public abstract void commit();
```

Commit a transaction.

getRollbackOnly()

```
  public abstract boolean getRollbackOnly();
```

Has this transaction been marked for rollback only?

Parameters	
return	true if the transaction has been marked for rollback only

isActive()

```
  public abstract boolean isActive();
```

Is there a transaction currently active?

Parameters	
return	true if a transaction is active

rollback()

```
  public abstract void rollback();
```

Roll back a transaction.

setRollbackOnly()

```
  public abstract void setRollbackOnly();
```

Mark this transaction as rollback only. After this method is called, commit() will roll back the transaction and throw an exception; rollback() will roll back the transaction and not throw an exception.

4.3.2 com.mysql.clusterj.annotation

This package provides annotations for domain object model interfaces mapped to database tables.

4.3.2.1 Column

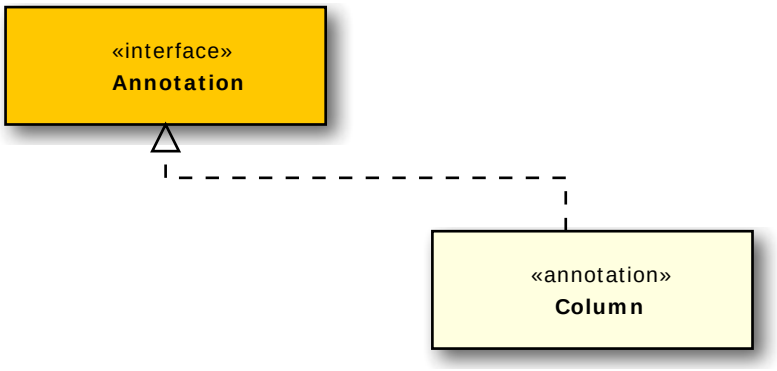
Annotation for a column in the database.

Synopsis

```
@Target(value={java.lang.annotation.ElementType.FIELD,
 java.lang.annotation.ElementType.METHOD}) @Retention(value=java.lang.annotation.RetentionPolicy.RUNTIME) p

  public String
    name ;

  public String
    allowsNull ;

  public String
```

```
    defaultValue ;

}
```

Figure 4.23 Column Class



allowsNull

Whether the column allows null values to be inserted. This overrides the database definition and requires that the application provide non-null values for the database column.

Parameters	
return	whether the column allows null values to be inserted

defaultValue

Default value for this column.

Parameters	
return	the default value for this column

name

Name of the column.

Parameters	
return	the name of the column

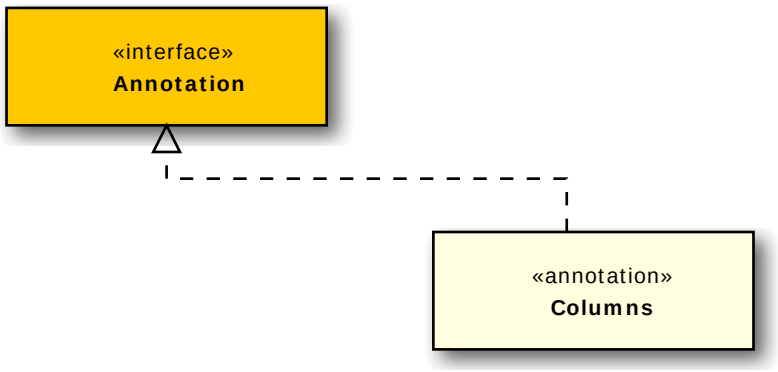
4.3.2.2 Columns

Annotation for a group of columns. This annotation is used for multi-column structures such as indexes and keys.

Synopsis

```
@Target(value={java.lang.annotation.ElementType.FIELD, java.lang.annotation.ElementType.METHOD,
 java.lang.annotation.ElementType.TYPE}) @Retention(value=java.lang.annotation.RetentionPolicy.RUNTIME)

  public Column[]
    value ;

}
```

Figure 4.24 Columns Class



value

The columns annotation information.

Parameters	
return	the columns

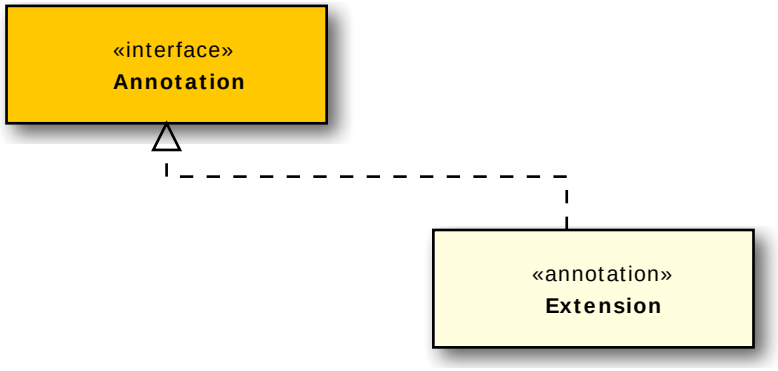
4.3.2.3 Extension

Annotation for a non-standard extension.

Synopsis

```
@Target(value={java.lang.annotation.ElementType.TYPE, java.lang.annotation.ElementType.FIELD,
 java.lang.annotation.ElementType.METHOD}) @Retention(value=java.lang.annotation.RetentionPolicy.RUNTIME) p

  public String
    vendorName ;

  public String
    key ;

  public String
    value ;

}
```

Figure 4.25 Extension Class



key

The key for the extension (required).

Parameters	
return	the key

value

The value for the extension (required).

Parameters	
return	the value

vendorName

Vendor that the extension applies to (required to make the key unique).

Parameters	
return	the vendor

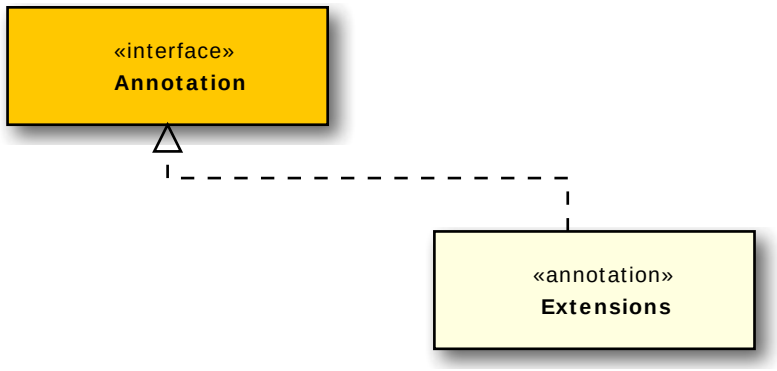
4.3.2.4 Extensions

Annotation for a group of extensions.

Synopsis

```
@Target(value={java.lang.annotation.ElementType.TYPE, java.lang.annotation.ElementType.FIELD,
 java.lang.annotation.ElementType.METHOD}) @Retention(value=java.lang.annotation.RetentionPolicy.RUNTIM

  public Extension[]
    value ;

}
```

Figure 4.26 Extensions Class



value

The extensions.

Parameters	
return	the extensions

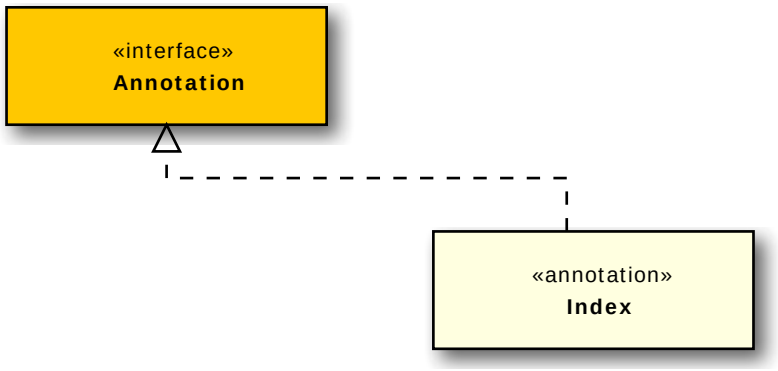
4.3.2.5 Index

Annotation for a database index.

Synopsis

```
@Target(value={java.lang.annotation.ElementType.TYPE, java.lang.annotation.ElementType.FIELD,
 java.lang.annotation.ElementType.METHOD}) @Retention(value=java.lang.annotation.RetentionPolicy.RUNTIME) p

  public String
    name ;

  public String
    unique ;

  public Column[]
    columns ;

}
```

Figure 4.27 Index Class



columns

Columns that compose this index.

Parameters	
return	columns that compose this index

name

Name of the index

Parameters	
return	the name of the index

unique

Whether this index is unique

Parameters	
return	whether this index is unique

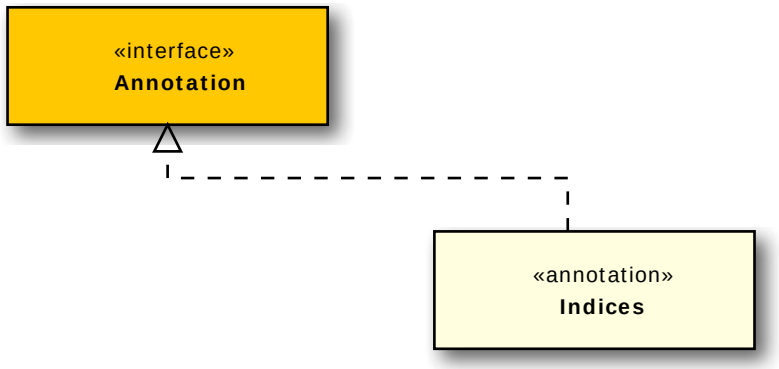
4.3.2.6 Indices

Annotation for a group of indices. This is used on a class where there are multiple indices defined.

Synopsis

```
@Target(value=java.lang.annotation.ElementType.TYPE) @Retention(value=java.lang.annotation.RetentionPol

  public Index[]
    value ;

}
```

Figure 4.28 Indices Class



value

The indices.

Parameters	
return	The indices

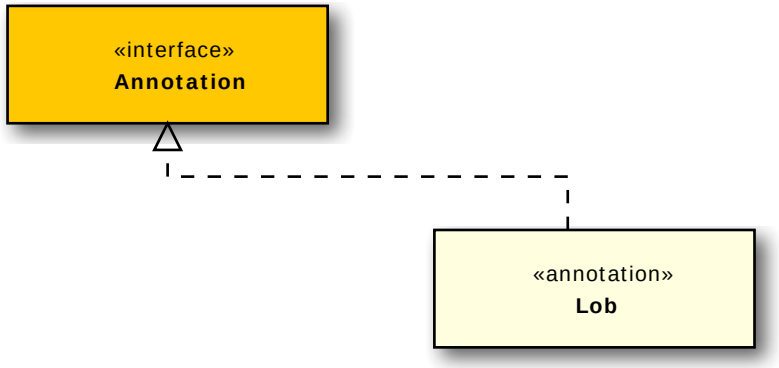
4.3.2.7 Lob

Annotation for a Large Object (lob). This annotation can be used with byte[] and InputStream types for binary columns; and with String and InputStream types for character columns.

Synopsis

```
@Target(value={java.lang.annotation.ElementType.FIELD,
 java.lang.annotation.ElementType.METHOD}) @Retention(value=java.lang.annotation.RetentionPolicy.RUNTIM
}
```

Figure 4.29 Lob Class



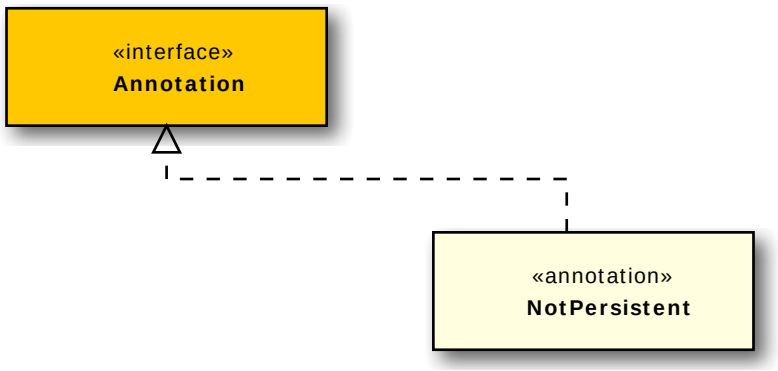
4.3.2.8 NotPersistent

Annotation to specify that the member is not persistent. If used, this is the only annotation allowed on a member.

Synopsis

```
@Target(value={java.lang.annotation.ElementType.FIELD,
 java.lang.annotation.ElementType.METHOD}) @Retention(value=java.lang.annotation.RetentionPolicy.RUNTIME) p
}
```

Figure 4.30 NotPersistent Class



4.3.2.9 NullValue

Enumeration of the "null-value" behavior values. This behavior is specified in the @Persistent annotation.

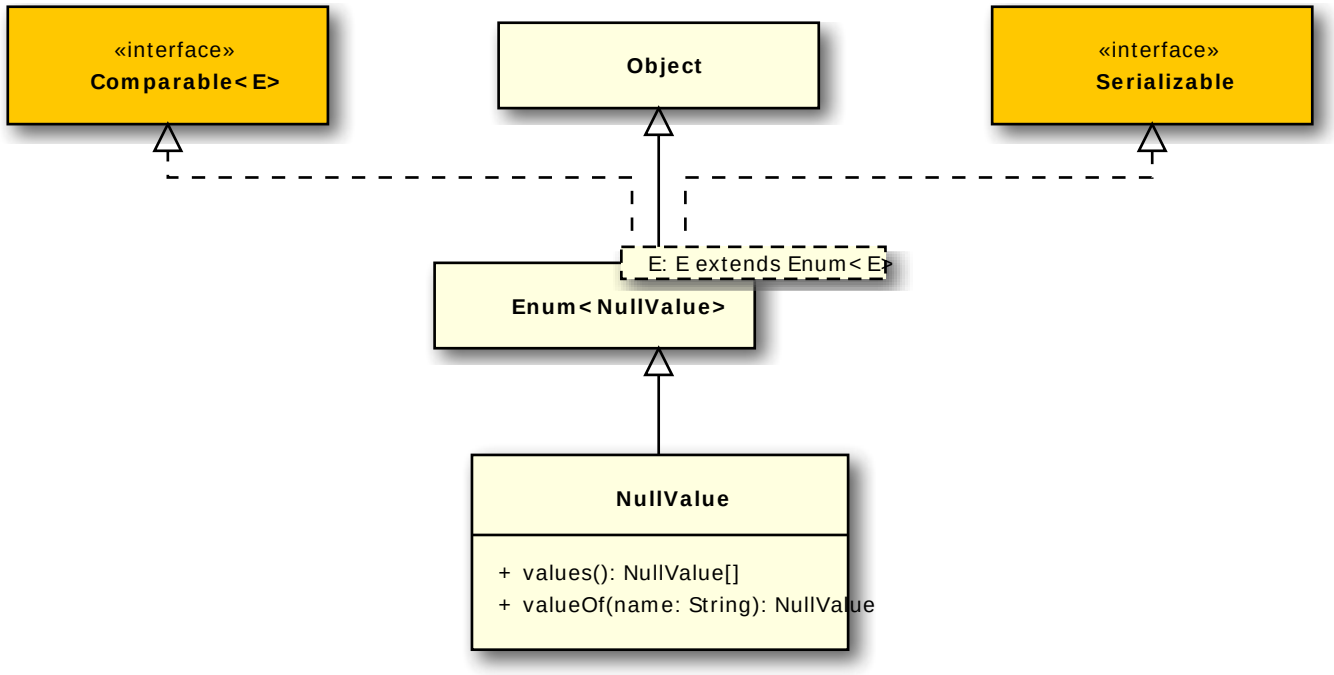
Synopsis

```
public final class NullValue,
  extends  Enum<NullValue> {
// Public Static Fields

  public static final NullValue
    DEFAULT ;

  public static final NullValue
    EXCEPTION ;

  public static final NullValue
    NONE ;

// Public Static Methods

  public static NullValue valueOf(
    String name);

  public static NullValue[] values();

}
```

Methods inherited from java.lang.Enum: `compareTo` , `equals` , `getDeclaringClass` , `hashCode` , `name` , `ordinal` , `toString` , `valueOf`

Methods inherited from java.lang.Object: `getClass` , `notify` , `notifyAll` , `wait`

Figure 4.31 NullValue Class



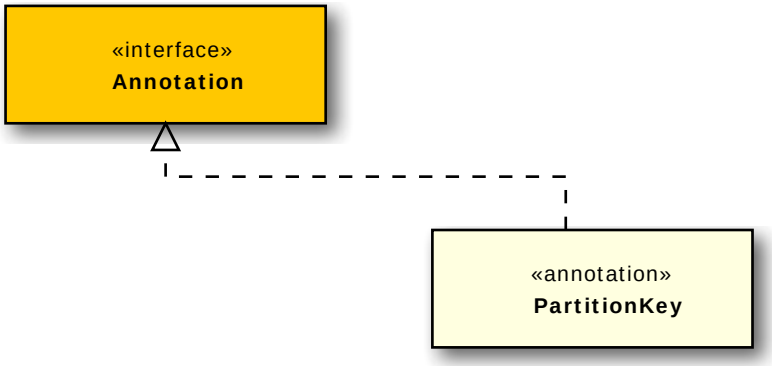
4.3.2.10 PartitionKey

Annotation on a class or member to define the partition key. If annotating a class or interface, either a single column or multiple columns can be specified. If annotating a member, neither column nor columns should be specified.

Synopsis

```
@Target(value={java.lang.annotation.ElementType.TYPE, java.lang.annotation.ElementType.FIELD,
 java.lang.annotation.ElementType.METHOD}) @Retention(value=java.lang.annotation.RetentionPolicy.RUNTIM

  public String
    column ;

  public Column[]
    columns ;

}
```

Figure 4.32 PartitionKey Class



column

Name of the column to use for the partition key

Parameters	
return	the name of the column to use for the partition key

columns

The column(s) for the partition key

Parameters	
return	the column(s) for the partition key

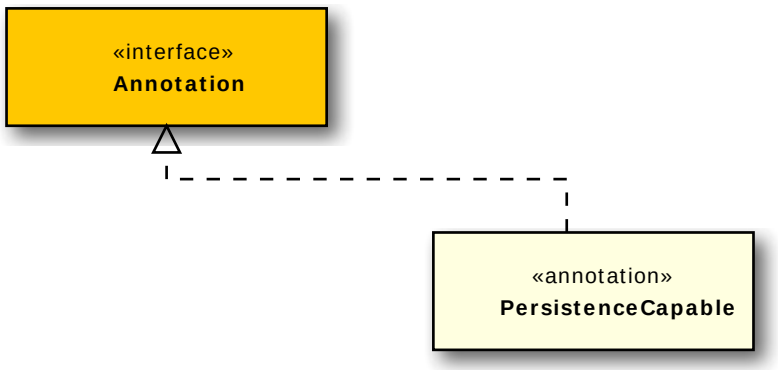
4.3.2.11 PersistenceCapable

Annotation for whether the class or interface is persistence-capable.

Synopsis

```
@Target(value=java.lang.annotation.ElementType.TYPE) @Retention(value=java.lang.annotation.RetentionPolicy.

  public String
    table ;

  public String
    database ;

  public String
    schema ;

}
```

Figure 4.33 PersistenceCapable Class



4.3.2.12 PersistenceModifier

Enumeration of the persistence-modifier values for a member.

Synopsis

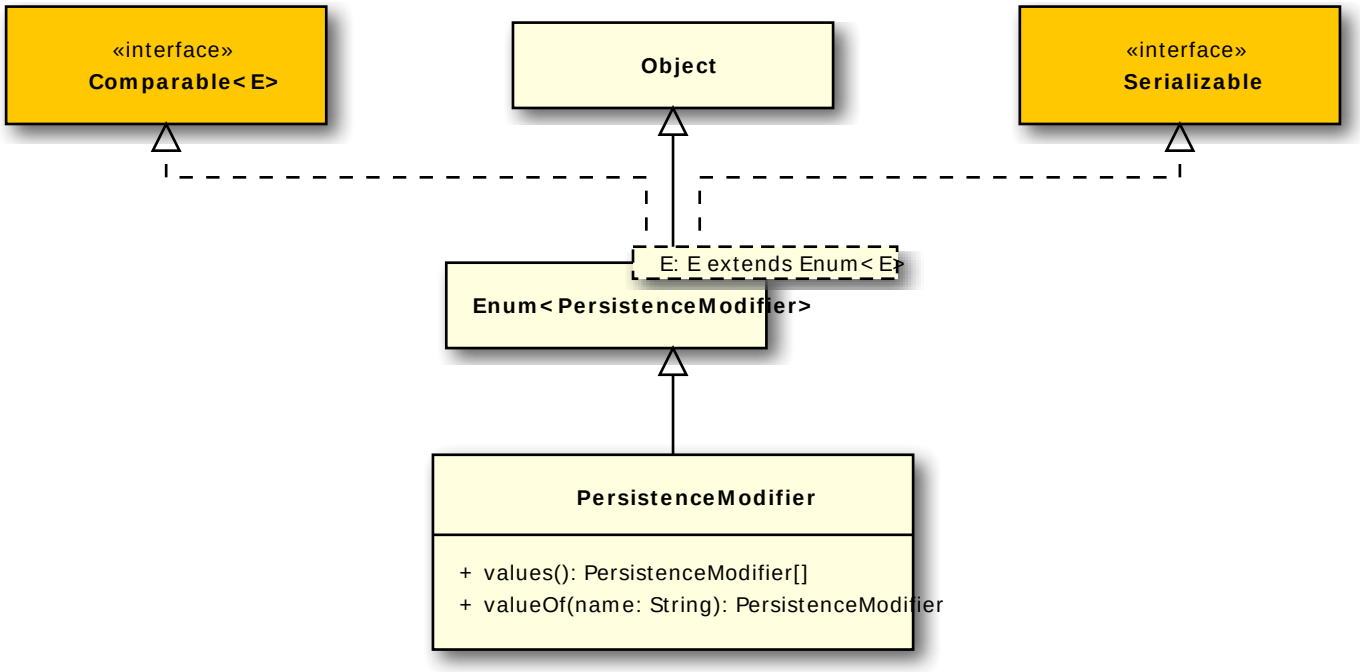
```
public final class PersistenceModifier,
  extends  Enum<PersistenceModifier> {
// Public Static Fields

  public static final PersistenceModifier
    NONE ;

  public static final PersistenceModifier
    PERSISTENT ;
```

```
  public static final PersistenceModifier
    UNSPECIFIED ;

// Public Static Methods

  public static PersistenceModifier valueOf(
    String name);

  public static PersistenceModifier[] values();

}
```

Methods inherited from java.lang.Enum: `compareTo` , `equals` , `getDeclaringClass` , `hashCode` , `name` , `ordinal` , `toString` , `valueOf`

Methods inherited from java.lang.Object: `getClass` , `notify` , `notifyAll` , `wait`

Figure 4.34 PersistenceModifier Class



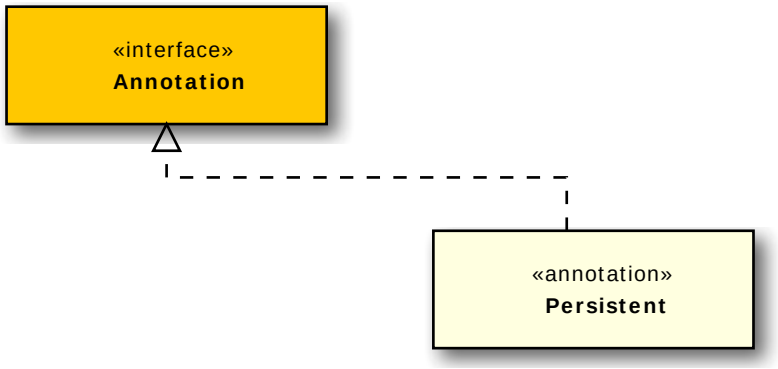
4.3.2.13 Persistent

Annotation for defining the persistence of a member.

Synopsis

```
@Target(value={java.lang.annotation.ElementType.FIELD,
 java.lang.annotation.ElementType.METHOD}) @Retention(value=java.lang.annotation.RetentionPolicy.RUNTIM

  public NullValue
    nullValue ;

  public String
    primaryKey ;

  public String
    column ;

  public Extension[]
    extensions ;

}
```

Figure 4.35 Persistent Class



column

Column name where the values are stored for this member.

Parameters	
return	the name of the column

extensions

Non-standard extensions for this member.

Parameters	
return	the non-standard extensions

nullValue

Behavior when this member contains a null value.

Parameters	
return	the behavior when this member contains a null value

primaryKey

Whether this member is part of the primary key for the table. This is equivalent to specifying @PrimaryKey as a separate annotation on the member.

Parameters	
return	whether this member is part of the primary key

4.3.2.14 PrimaryKey

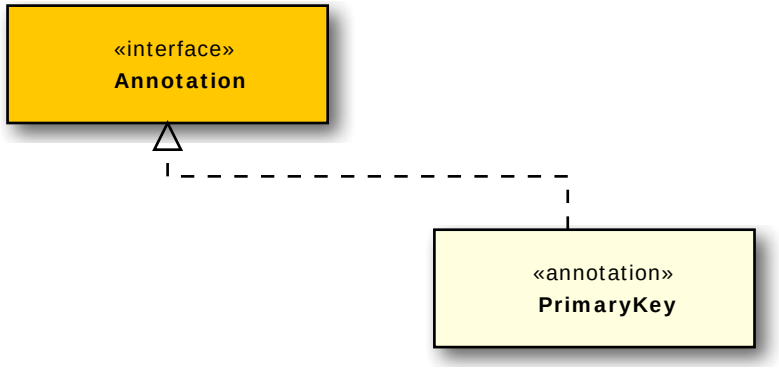
Annotation on a member to define it as a primary key member of a class or persistent interface.

Synopsis

```
@Target(value={java.lang.annotation.ElementType.TYPE, java.lang.annotation.ElementType.FIELD,
 java.lang.annotation.ElementType.METHOD}) @Retention(value=java.lang.annotation.RetentionPolicy.RUNTIME) p

  public String
    name ;

  public String
    column ;

  public Column[]
```

```
    columns ;
}
```

Figure 4.36 PrimaryKey Class



column

Name of the column to use for the primary key

Parameters	
return	the name of the column to use for the primary key

columns

The column(s) for the primary key

Parameters	
return	the column(s) for the primary key

name

Name of the primary key constraint

Parameters	
return	the name of the primary key constraint

4.3.3 com.mysql.clusterj.query

Provides interfaces for building queries for ClusterJ.

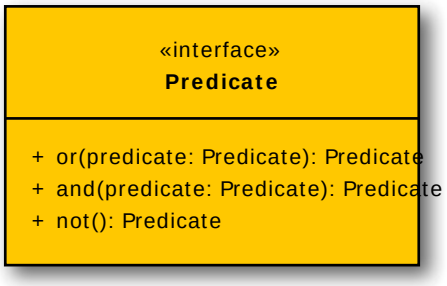
4.3.3.1 Predicate

Used to combine multiple predicates with boolean operations.

Synopsis

```
public interface Predicate {
// Public Methods

  public abstract Predicate and(
    Predicate predicate);

  public abstract Predicate not();

  public abstract Predicate or(
    Predicate predicate);

}
```

Figure 4.37 Predicate Class



and(Predicate)

```
  public abstract Predicate and(
    Predicate predicate);
```

Combine this Predicate with another, using the "and" semantic.

Parameters	
predicate	the other predicate
return	a new Predicate combining both Predicates

not()

```
  public abstract Predicate not();
```

Negate this Predicate.

Parameters	
return	this predicate

or(Predicate)

```
  public abstract Predicate or(
    Predicate predicate);
```

Combine this Predicate with another, using the "or" semantic.

Parameters	
predicate	the other predicate
return	a new Predicate combining both Predicates

4.3.3.2 PredicateOperand

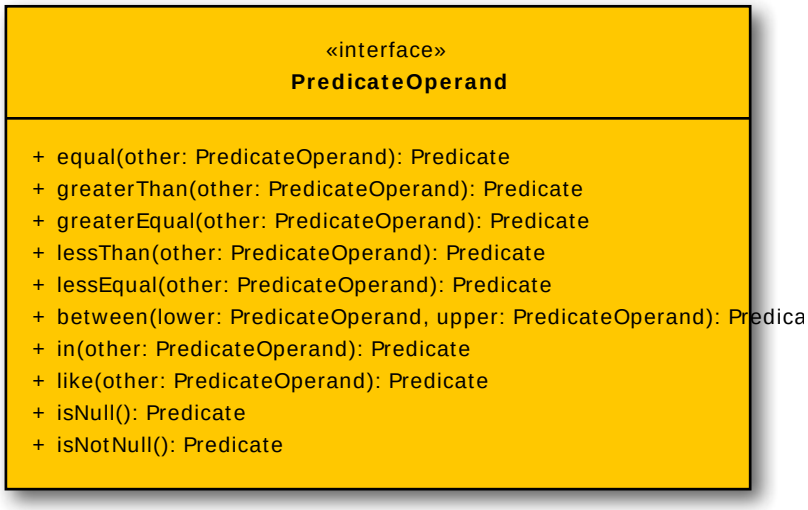
PredicateOperand represents a column or parameter that can be compared to another

Synopsis

```
public interface PredicateOperand {
// Public Methods

  public abstract Predicate between(
    PredicateOperand lower,
    PredicateOperand upper);

  public abstract Predicate equal(
    PredicateOperand other);

  public abstract Predicate greaterEqual(
    PredicateOperand other);
```

```
  public abstract Predicate greaterThan(
    PredicateOperand other);

  public abstract Predicate in(
    PredicateOperand other);

  public abstract Predicate isNotNull();

  public abstract Predicate isNull();

  public abstract Predicate lessEqual(
    PredicateOperand other);

  public abstract Predicate lessThan(
    PredicateOperand other);

  public abstract Predicate like(
    PredicateOperand other);

}
```

Figure 4.38 PredicateOperand Class



between(PredicateOperand, PredicateOperand)

```
  public abstract Predicate between(
    PredicateOperand lower,
    PredicateOperand upper);
```

Return a Predicate representing comparing this to another using "between" semantics.

Parameters	
lower	another PredicateOperand
upper	another PredicateOperand
return	a new Predicate

equal(PredicateOperand)

```
  public abstract Predicate equal(
    PredicateOperand other);
```

Return a Predicate representing comparing this to another using "equal to" semantics.

Parameters	
other	the other PredicateOperand

return	a new Predicate

greaterEqual(PredicateOperand)

```
public abstract Predicate greaterEqual(
  PredicateOperand other);
```

Return a Predicate representing comparing this to another using "greater than or equal to" semantics.

Parameters	
other	the other PredicateOperand
return	a new Predicate

greaterThan(PredicateOperand)

```
public abstract Predicate greaterThan(
  PredicateOperand other);
```

Return a Predicate representing comparing this to another using "greater than" semantics.

Parameters	
other	the other PredicateOperand
return	a new Predicate

in(PredicateOperand)

```
public abstract Predicate in(
  PredicateOperand other);
```

Return a Predicate representing comparing this to a collection of values using "in" semantics.

Parameters	
other	another PredicateOperand
return	a new Predicate

isNotNull()

```
public abstract Predicate isNotNull();
```

Return a Predicate representing comparing this to not null.

Parameters	
return	a new Predicate

isNull()

```
public abstract Predicate isNull();
```

Return a Predicate representing comparing this to null.

Parameters	
return	a new Predicate

lessEqual(PredicateOperand)

```
public abstract Predicate lessEqual(
  PredicateOperand other);
```

Return a Predicate representing comparing this to another using "less than or equal to" semantics.

Parameters

other	the other PredicateOperand
return	a new Predicate

lessThan(PredicateOperand)

```
  public abstract Predicate lessThan(
    PredicateOperand other);
```

Return a Predicate representing comparing this to another using "less than" semantics.

Parameters	
other	the other PredicateOperand
return	a new Predicate

like(PredicateOperand)

```
  public abstract Predicate like(
    PredicateOperand other);
```

Return a Predicate representing comparing this to another using "like" semantics.

Parameters	
other	another PredicateOperand
return	a new Predicate

4.3.3.3 QueryBuilder

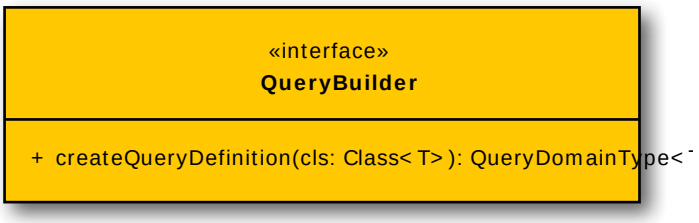
QueryBuilder represents a factory for queries.

Synopsis

```
public interface QueryBuilder {
// Public Methods

  public abstract QueryDomainType<T> createQueryDefinition(
    Class<T> cls);

}
```

See Also getQueryBuilder()

Figure 4.39 QueryBuilder Class



createQueryDefinition(Class<T>)

```
  public abstract QueryDomainType<T> createQueryDefinition(
    Class<T> cls);
```

Create a QueryDefinition to define queries.

Parameters	
cls	the class of the type to be queried

return	the QueryDomainType to define the query

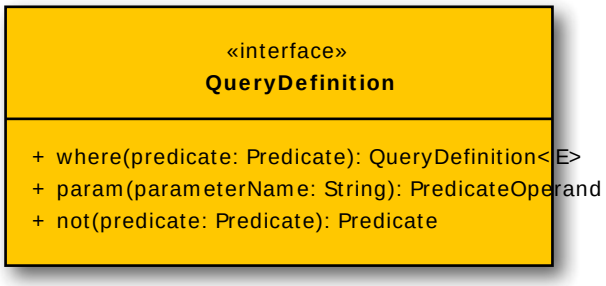
4.3.3.4 QueryDefinition

QueryDefinition allows users to define queries.

Synopsis

```
public interface QueryDefinition<E> {
// Public Methods

  public abstract Predicate not(
    Predicate predicate);

  public abstract PredicateOperand param(
    String parameterName);

  public abstract QueryDefinition<E> where(
    Predicate predicate);

}
```

Figure 4.40 QueryDefinition Class



not(Predicate)

```
  public abstract Predicate not(
    Predicate predicate);
```

Convenience method to negate a predicate.

Parameters	
predicate	the predicate to negate
return	the inverted predicate

param(String)

```
  public abstract PredicateOperand param(
    String parameterName);
```

Specify a parameter for the query.

Parameters	
parameterName	the name of the parameter
return	the PredicateOperand representing the parameter

where(Predicate)

```
  public abstract QueryDefinition<E> where(
    Predicate predicate);
```

Specify the predicate to satisfy the query.

Parameters	
predicate	the Predicate
return	this query definition

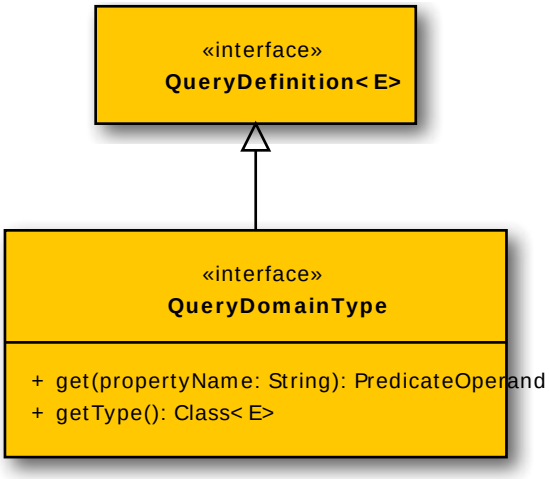
4.3.3.5 QueryDomainType

QueryDomainType represents the domain type of a query. The domain type validates property names that are used to filter results.

Synopsis

```
public interface QueryDomainType<E>,
  extends  QueryDefinition<E> {
// Public Methods

  public abstract PredicateOperand get(
    String propertyName);

  public abstract Class<E> getType();

}
```

Figure 4.41 QueryDomainType Class



get(String)

```
  public abstract PredicateOperand get(
    String propertyName);
```

Get a PredicateOperand representing a property of the domain type.

Parameters	
propertyName	the name of the property
return	a representation the value of the property

getType()

```
  public abstract Class<E> getType();
```

Get the domain type of the query.

Parameters	
return	the domain type of the query

4.3.4 Constant field values

4.3.4.1 com.mysql.clusterj.*

Table 4.1 com.mysql.clusterj.* Constants and Default Values

Name: **Default Value:**
Name: DEFAULT_PROPERTY_CLUSTER_BYTE_BUFFER_POOL_SIZES **Default Value**: "256, 10240, 102400, 1048576"
Name: DEFAULT_PROPERTY_CLUSTER_CONNECT_AUTO_INCREMENT_BATCH_SIZE **Default Value**: 10
Name: DEFAULT_PROPERTY_CLUSTER_CONNECT_AUTO_INCREMENT_START **Default Value**: 1
Name: DEFAULT_PROPERTY_CLUSTER_CONNECT_AUTO_INCREMENT_STEP **Default Value**: 1
Name: DEFAULT_PROPERTY_CLUSTER_CONNECT_DELAY **Default Value**: 5
Name: DEFAULT_PROPERTY_CLUSTER_CONNECT_RETRIES **Default Value**: 4
Name: DEFAULT_PROPERTY_CLUSTER_CONNECT_TIMEOUT_AFTER **Default Value**: 20
Name: DEFAULT_PROPERTY_CLUSTER_CONNECT_TIMEOUT_BEFORE **Default Value**: 30
Name: DEFAULT_PROPERTY_CLUSTER_CONNECT_TIMEOUT_MGM **Default Value**: 30000
Name: DEFAULT_PROPERTY_CLUSTER_CONNECT_VERBOSE **Default Value**: 0
Name: DEFAULT_PROPERTY_CLUSTER_DATABASE **Default Value**: "test"
Name: DEFAULT_PROPERTY_CLUSTER_MAX_TRANSACTIONS **Default Value**: 4
Name: DEFAULT_PROPERTY_CONNECTION_POOL_SIZE **Default Value**: 1
Name: ENV_CLUSTERJ_LOGGER_FACTORY_NAME **Default Value**: "CLUSTERJ_LOGGER_FACTORY"
Name: PROPERTY_CLUSTER_BYTE_BUFFER_POOL_SIZES **Default Value**: "com.mysql.clusterj.byte.buffer.pool.sizes"
Name: PROPERTY_CLUSTER_CONNECTION_SERVICE

Name: **Default Value:**
Default Value: "com.mysql.clusterj.connection.service"
Name: PROPERTY_CLUSTER_CONNECTSTRING **Default Value**: "com.mysql.clusterj.connectstring"
Name: PROPERTY_CLUSTER_CONNECT_AUTO_INCREMENT_BATCH_SIZE **Default Value**: "com.mysql.clusterj.connect.autoincrement.batchsize"
Name: PROPERTY_CLUSTER_CONNECT_AUTO_INCREMENT_START **Default Value**: "com.mysql.clusterj.connect.autoincrement.offset"
Name: PROPERTY_CLUSTER_CONNECT_AUTO_INCREMENT_STEP **Default Value**: "com.mysql.clusterj.connect.autoincrement.increment"
Name: PROPERTY_CLUSTER_CONNECT_DELAY **Default Value**: "com.mysql.clusterj.connect.delay"
Name: PROPERTY_CLUSTER_CONNECT_RETRIES **Default Value**: "com.mysql.clusterj.connect.retries"
Name: PROPERTY_CLUSTER_CONNECT_TIMEOUT_AFTER **Default Value**: "com.mysql.clusterj.connect.timeout.after"
Name: PROPERTY_CLUSTER_CONNECT_TIMEOUT_BEFORE **Default Value**: "com.mysql.clusterj.connect.timeout.before"
Name: PROPERTY_CLUSTER_CONNECT_TIMEOUT_MGM **Default Value**: "com.mysql.clusterj.connect.timeout.mgm"
Name: PROPERTY_CLUSTER_CONNECT_VERBOSE **Default Value**: "com.mysql.clusterj.connect.verbose"
Name: PROPERTY_CLUSTER_DATABASE **Default Value**: "com.mysql.clusterj.database"
Name: PROPERTY_CLUSTER_MAX_TRANSACTIONS **Default Value**: "com.mysql.clusterj.max.transactions"
Name: PROPERTY_CONNECTION_POOL_NODEIDS **Default Value**: "com.mysql.clusterj.connection.pool.nodeids"
Name: PROPERTY_CONNECTION_POOL_SIZE **Default Value**: "com.mysql.clusterj.connection.pool.size"
Name: PROPERTY_DEFER_CHANGES **Default Value**: "com.mysql.clusterj.defer.changes"
Name: PROPERTY_JDBC_DRIVER_NAME **Default Value**: "com.mysql.clusterj.jdbc.driver"
Name: PROPERTY_JDBC_PASSWORD

Name: **Default Value:**
Default Value: "com.mysql.clusterj.jdbc.password"
Name: PROPERTY_JDBC_URL **Default Value**: "com.mysql.clusterj.jdbc.url"
Name: PROPERTY_JDBC_USERNAME **Default Value**: "com.mysql.clusterj.jdbc.username"
Name: SESSION_FACTORY_SERVICE_CLASS_NAME **Default Value**: "com.mysql.clusterj.SessionFactoryService"
Name: SESSION_FACTORY_SERVICE_FILE_NAME **Default Value**: "META-INF/services/com.mysql.clusterj.SessionFactoryService"

Table 4.2 Query Constants and Default Values

Name: **Default Value:**
Name: INDEX_USED **Default Value**: "IndexUsed"
Name: SCAN_TYPE **Default Value**: "ScanType"
Name: SCAN_TYPE_INDEX_SCAN **Default Value**: "INDEX_SCAN"
Name: SCAN_TYPE_PRIMARY_KEY **Default Value**: "PRIMARY_KEY"
Name: SCAN_TYPE_TABLE_SCAN **Default Value**: "TABLE_SCAN"
Name: SCAN_TYPE_UNIQUE_KEY **Default Value**: "UNIQUE_KEY"

4.4 MySQL Cluster Connector for Java: Limitations and Known Issues

This section discusses the limitations and known issues in the MySQL Cluster Connector for Java APIs.

Known issues in ClusterJ:

- *Joins:* With ClusterJ, queries are limited to single tables. This is not a problem with JPA or JDBC, both of which support joins.
- *Database views:* Because MySQL database views do not use the `NDB` storage engine, ClusterJ applications cannot “see” views, and thus cannot access them. To work with views using Java, you should use JPA or JDBC.

- *Relations and inheritance:* ClusterJ does not support relations or inheritance. Tables are mapped one-to-one onto domain classes, and only single-table operations are supported. `NDB` tables for NDB Cluster 7.3 and later support foreign keys, and foreign key constraints are enforced when using ClusterJ for inserts, updates, and deletes.

- *TIMESTAMP:* Currently, ClusterJ does not support the TIMESTAMP data type for a primary key field.

Known issues in JPA: For information about limitations and known issues with OpenJPA, see the OpenJPA documentation.

Known issues in JDBC and Connector/J: For information about limitations and known issues with JDBC and Connector/J, see JDBC API Implementation Notes, and Troubleshooting Connector/J Applications.

Known issues in NDB Cluster: For information about limitations and other known issues with NDB Cluster, see Known Limitations of NDB Cluster.

Chapter 5 MySQL NoSQL Connector for JavaScript

Table of Contents

5.1 MySQL NoSQL Connector for JavaScript Overview 621
5.2 Installing the JavaScript Connector 621
5.3 Connector for JavaScript API Documentation 622
5.3.1 Batch 623
5.3.2 Context 623
5.3.3 Converter 625
5.3.4 Errors 626
5.3.5 Mynode 626
5.3.6 Session 629
5.3.7 SessionFactory 630
5.3.8 TableMapping and FieldMapping 630
5.3.9 TableMetadata 631
5.3.10 Transaction 632
5.4 Using the MySQL JavaScript Connector: Examples 633
5.4.1 Requirements for the Examples 633
5.4.2 Example: Finding Rows 637
5.4.3 Inserting Rows 639
5.4.4 Deleting Rows 640

This section provides information about the MySQL NoSQL Connector for JavaScript, a set of Node.js adapters for NDB Cluster and MySQL Server available beginning with NDB 7.3.1, which make it possible to write JavaScript applications for Node.js using MySQL data.

5.1 MySQL NoSQL Connector for JavaScript Overview

This connector differs in a number of key respects from most other MySQL Connectors and APIs. The interface is asynchronous, following the built-in Node.js event model. In addition, it employs a domain object model for data storage. Applications retrieve data in the form of fully-instantiated objects, rather than as rows and columns.

The MySQL Node.js adapter includes 2 drivers. The `ndb` driver accesses the `NDB` storage engine directly, using the NDB API (see Chapter 2, *The NDB API*). No MySQL Server is required for the `ndb` driver. The `mysql` driver uses a MySQL Server for its data source, and depends on the `node-mysql` Node.js module from https://github.com/felixge/node-mysql/. Regardless of the driver in use, no SQL statements are required; when using the Connector for JavaScript, Node.js applications employ data objects for all requests made to the database.

5.2 Installing the JavaScript Connector

This section covers basic installation and setup of the MySQL JavaScript Connector and its prerequites. The Connector requires both Node.js and NDB Cluster to be installed first; you can install these in either order. In addition, the `mysql-js` adapter requires the `node-mysql` driver. Building the Connector also requires that your system have a working C++ compiler such as `gcc` or Microsoft Visual Studio.

To install all of the prerequisites for the JavaScript Connector, including `node-mysql`, you should perform the following steps:

1. **Node.js.** If you do not already have Node.js installed on your system, you can obtain it from http://nodejs.org/download/. In addition to source code, prebuilt binaries and installers are available

for a number of platforms. Many Linux distributions also have Node.js in their repositories (you may need to add an alternative repository in your package manager).

NDB 7.3.1 requires Node.js version 0.7.9 or earlier, due to dependency on `node-waf`. NDB 7.3.2 and later use `node-gyp` (see https://npmjs.org/package/node-gyp), and should work with Node.js 0.8.0 and later.

Regardless of the method by which you obtain Node.js, keep in mind that the architecture of the version you install must match that of the NDB Cluster binaries you intend to use; you cannot, for example, install the JavaScript Connector using 64-bit Node.js and 32-bit NDB Cluster. If you do not know the architecture of your existing Node.js installation, you can determine this by checking the value of `global.process.arch`.

2. **NDB Cluster.** If NDB Cluster, including all header and library files, is not already installed on the system, install it (see NDB Cluster Installation).

   As mentioned previously, you must make sure that the architecture (32-bit or 64-bit) is the same for both NDB Cluster and Node.js. You can check the architecture of an existing NDB Cluster installation in the output of `ndb_mgm -V`.

3. **node-mysql driver.** The `mysql-js` adapter also requires a working installation of the `node-mysql` driver from https://github.com/felixge/node-mysql/. You can install the driver using the Node.js `npm install` command; see the project web site for the recommended version and package identifier.

Once the requirements just listed are met, you can find the files needed to install the MySQL Connector for JavaScript in `share/nodejs` in the NDB Cluster installation directory. (If you installed NDB Cluster as an RPM, this is `/usr/share/mysql/nodejs`.) To use the Node.js `npm` tool to perform a "best-guess" installation without any user intervention, change to the `share/nodejs` directory, then use `npm` as shown here:

```
shell> npm install .
```

The final period (`.`) character is required. Note that you must run this command in `share/node.js` in the NDB Cluster installation directory.

You can test your installation using the supplied test program. This requires a running NDB Cluster, including a MySQL Server with a database named `test`. The `mysql` client executable must be in the path.

To run the test suite, change to the `test` directory, then execute command shown here:

```
shell> node driver
```

By default, all servers are run on the local machine using default ports; this can be changed by editing the file `test/test_connection.js`, which is generated by running the test suite. If this file is not already present (see Bug #16967624), you can copy `share/nodejs/test/lib/test_connection_js` to the `test` directory for this purpose.

If you istalled NDB Cluster to a nondefault location, you may need to export the `LD_LIBRARY_PATH` to enable the test suite. The test suite also requires that the `test` database be available on the MySQL server.

NDB 7.3.1 also provided an alternative build script in `share/node.js/setup`; this was removed in NDB 7.3.2 and later NDB Cluster 7.3 releases.

5.3 Connector for JavaScript API Documentation

This section contains prototype descriptions and other information for the MySQL Connector for JavaScript.

5.3.1 Batch

This class represents a batch of operations.

```
Batch extends Context
```

```
execute(Function(Object error) callback);
```

Execute this batch. When a batch is executed, all operations are executed; the callback for each operation is called when that operation is executed (operations are not performed in any particular order). The `execute()` function's *`callback`* is also called.

A batch is executed in the context of the session's current state: this is autocommit if a transaction has not been started; this also includes the default lock mode and the partition key.

```
clear();
```

Clear this batch without affecting the transaction state. After being cleared, the batch is still valid, but all operations previously defined are removed; this restores the batch to a clean state.

The callbacks for any operations that are defined for this batch are called with an error indicating that the batch has been cleared.

This function requires no arguments.

```
getSession();
```

Get the session from which this batch was created.

This function requires no arguments.

5.3.2 Context

`Context` is the supertype of `Session` and `Batch`. It contains functions that are executed immediately if called from a session, or when the batch is executed.

The `Mynode` implementation does have any concept of a user and does not define any such property.

```
find(Function constructor, Object keys, Function(Object error, Object instance[, ...]) callback[, ...])

find(String tableName, Object keys, Function(Object error, Object instance[, ...]) callback[, ...]);
```

Find a specific instance based on a primary key or unique key value.

You can use either of two versions of this function. In the first version, the *`constructor`* parameter is the constructor function of a mapped domain object. Alternatively, you can use the *`tableName`* instead, in the second variant of the function.

For both versions of `find()`, the *`keys`* may be of any type. A key must uniquely identify a single row in the database. If *`keys`* is a simple type (number or string), then the parameter type must be the same type as or compatible with the primary key type of the mapped object. Otherwise, properties are taken from the parameter and matched against property names in the mapping. Primary key properties are used if all are present, and other properties ignored. If *`keys`* cannot be used identify the primary key, property names corresponding to unique key columns are used instead. If no complete primary or unique key properties are found, an error is reported. The returned object is loaded based on the mapping and the current values in the database.

For multi-column primary or unique keys, all key fields must be set.

```
load(Object instance, Function(Object error) callback);
```

Load a specific instance by matching its primary or unique key with a database row, without creating a new domain object. (This is unlike `find()`, which creates a new, mapped domain object.)

The *`instance`* must have its primary or unique key value or values set. The mapped values in the object are loaded based on the current values in the database. Unmapped properties in the object are not changed.

Primary key properties are used if all are present, and all other properties are ignored; otherwise, property names corresponding to unique key columns are used. If no complete primary or unique key properties can be found, an error is reported.

The *`callback`* function is called with the parameters provided when the operation has completed. The *`error`* is the Node.js `Error` object; see Section 5.3.4, “Errors”, for more information.

```
persist(Object instance, Function(Object error) callback);

persist(Function constructor, Object values, Function(Object error) callback);

persist(String tableName, Object values, Function(Object error) callback);
```

Insert an instance into the database, unless the instance already exists in the database, in which case an exception is reported to a *`callback`* function. Autogenerated values are present in the instance when the *`callback`* is executed.

The role of an instance to be persisted can be fulfilled in any of three ways: by an instance object; by a constructor, with parameters, for a mapped domain object; or by table name and values to be inserted.

In all three cases, the *`callback`* function is called with the parameters provided, if any, when the operation has completed. The *`error`* is the Node.js `Error` object; see Section 5.3.4, “Errors”, for more information.

```
remove(Object instance, Function(Object error) callback);

remove(Function constructor, Object keys, Function(Object error) callback);

remove(String tableName, Object keys, Function(Object error) callback);
```

Delete an instance of a class from the database by a primary or unique key.

There are three versions of `remove()`; these allow you to delete an instance by referring to the *`instance`* object, to a *`constructor`* function, or by name of the table. The *`instance`* object must contain key values that uniquely identify a single row in the database. Otherwise, if the *`keys`* supplied with the function constructor or table name is a simple type (`Number` or `String`), then the parameter type must be of either the same type as or a type compatible with the primary key type of the mapped object. If *`keys`* is not a simple type, properties are taken from the parameter and matched against property names in the mapping. Primary key properties are used if all are present, and other properties ignored. If *`keys`* does not identify the primary key, property names corresponding to unique key columnsare used instead. If no complete primary or unique key properties are found, an error is reported to the *`callback`*.

All three versions of `remove()` call the *`callback`* function with the parameters provided, if any, when the operation is complete. The *`error`* object is a Node.js `Error`; see Section 5.3.4, “Errors”, for error codes.

```
update(Object instance, Function(Object error) callback);

update(Function constructor, keys, values, Function(Object error) callback);

update(String tableName, keys, values, Function(Object error) callback);
```

Update an instance in the database with the supplied *values* without retrieving it. The primary key is used to determine which instance is updated. If the instance does not exist in the database, an exception is reported in the *callback*.

As with the methods previously shown for persisting instances in and removing them from the database, `update()` exists in three variations, which allow you to use the *instance* as an object, an object *constructor* with *keys*, or by *tableName* and *keys*.

Unique key fields of the *keys* object determine which *instance* is to be updated. The *values* object provides values to be updated. If the *keys* object contains all fields corresponding to the primary key, the primary key identifies the instance. If not, unique keys are chosen is a nondeterministic manner.

Note

`update()` cannot be used to change the primary key.

```
save(Object instance, Function(Object error) callback);

save(Function constructor, Object values, Function(Object error) callback);

save(String tableName, Object values, Function(Object error) callback);
```

Save an instance in the database without checking for its existence. If the instance already exists, it is updated (as if you had used `update()`); otherwise, it is created (as if `persist()` had been used). The instance `id` property is used to determine which instance should be saved. As with `update()`, `persist()`, and `remove()`, this method allows you to specify the instance using an object, object constructor, or table name.

All three versions of the `save()` method call the *callback* function with any parameters provided when the operation has been completed. The *error* is a Node.js `Error` object; see Section 5.3.4, “Errors”, for error codes and messages.

```
Boolean isBatch()
```

`Context` also exposes an `isBatch()` instance method, which returns true if this `Context` is a `Batch`, and false if it is a `Session`. `isBatch()` takes no arguments.

5.3.3 Converter

Converter classes convert between JavaScript types and MySQL types. If the user supplies a JavaScript converter, it used to read and write to the database.

Converters have several purposes, including the following:

- To convert between MySQL `DECIMAL` types and a user's preferred JavaScript fixed-precision utility library
- To convert between MySQL `BIGINT` types and a user's preferred JavaScript big number utility library
- To serialize arbitrary application objects into character or binary columns

The ndb backend also uses converters to support `SET` and `ENUM` columns. (The mysql backend does not use these.)

A `Converter` class has the interface defined here:

```
function Converter() {}:

Converter.prototype = {
  "toDB"    : function(obj) {  },
  "fromDB"  : function(val) {  }
```

```
};
```

The `Converter` *must* implement the following two functions:

1. `toDB(obj)`: Convert an application object *`obj`* into a form that can be stored in the database.
2. `fromDB(val)`: Convert a value *`val`* read from the database into application object format.

Each function returns the result of the conversion.

Converter invocations are chained in the following ways:

- When writing to the database, first the registered `FieldConverter`, if any, is invoked. Later, any registered `TypeConverter` is invoked.
- When reading from the database, first the registered `TypeConverter`, if any, is invoked. Later, any registered `FieldConverter` is invoked.

5.3.4 Errors

The `Errors` object contains the error codes and message exposed by the MySQL Node.js adapters.

```
var Errors;

Errors = {
  /* Standard-defined classes, SQL-99 */
  "02000" : "No Data",

  // connection errors
  "08000" : "Connection error",
  "08001" : "Unable to connect to server",
  "08004" : "Connection refused",

  // data errors
  "22000" : "Data error",
  "22001" : "String too long",
  "22003" : "Numeric value out of range",
  "22008" : "Invalid datetime",

  // Constraint violations
  // 23000 includes both duplicate primary key and duplicate unique key
  "23000" : "Integrity Constraint Violation",

  // misc. errors
  "25000" : "Invalid Transaction State",
  "2C000" : "Invalid character set name",
  "42S02" : "Table not found",
  "IM001" : "Driver does not support this function",

  /* Implementation-defined classes (NDB) */
  "NDB00" : "Refer to ndb_error for details"
};
```

5.3.5 Mynode

This class is used to generate and obtain information about sessions (`Session` objects). To create an instance, use the Node.js `require()` function with the driver name, like this:

```
var nosql = require("mysql-js");
```

`ConnectionProperties` can be used to retrieve or set the connection properties for a given session. You can obtain a complete set of of default connection properties for a given adapter using the `ConnectionProperties` constructor, shown here, with the name of the adapter (a string) used as the value of *`nameOrProperties`*:

```
```

```
ConnectionProperties(nameOrProperties);
```

You can also create your own `ConnectionProperties` object by supplying a list of property names and values to a new `ConnectionProperties` object in place of the adapter name. Then you can use this object to set the connection properties for a new session, as shown here:

```
var NdbConnectionProperties = {
  "implementation" : "ndb",

  "ndb_connectstring" : "localhost:1186",
  "database"          : "test",
  "mysql_user"        : "root",

  "ndb_connect_retries" : 4,
  "ndb_connect_delay"   : 5,
  "ndb_connect_verbose" : 0,

  "linger_on_close_msec": 500,
  "use_ndb_async_api"   : false,

  "ndb_session_pool_min" : 4,
  "ndb_session_pool_max" : 100,
};

var sharePath = '/usr/local/mysql/share/nodejs';      // path to share/nodejs
var nosql = require(sharePath);
var dbProperties = nosql.ConnectionProperties(NdbConnectionProperties);
```

It is also possible to obtain an object with the adapter's default connection properties, after which you can update a selected number of these properties, then use the modified object to set connection properties for the session, as shown here:

```
var sharePath = '/usr/local/mysql/share/nodejs';      // path to share/nodejs
var spi   = require(sharePath + "/Adapter/impl/SPI"); // under share/nodejs

var serviceProvider         = spi.getDBServiceProvider('ndb');
var NdbConnectionProperties = serviceProvider.getDefaultConnectionProperties();

NdbConnectionProperties.mysql_user = 'nodejs_user';
NdbConnectionProperties.database   = 'my_nodejs_db';

var dbProperties = nosql.ConnectionProperties(NdbConnectionProperties);
```

The `ConnectionProperties` object includes the following properties:

- `implementation`: For Node.js applications using NDB Cluster, this is always “ndb”.
- `ndb_connectstring`: NDB Cluster connection string used to connect to the management server.
- `database`: Name of the MySQL database to use.
- `mysql_user`: MySQL user name.
- `ndb_connect_retries`: Number of times to retry a failed connection before timing out; use a number less than 0 for this to keep trying the connection without ever stopping.
- `ndb_connect_delay`: Interval in seconds between connection retries.
- `ndb_connect_verbose`: 1 or 0; 1 enables extra console output during connection.
- `linger_on_close_msec`: When a client closes a `DBConnectionPool`, the underlying connection is kept open for this many milliseconds in case another client tries to reuse it.
- `use_ndb_async_api`: If true, some operations are executed using asynchronous calls for improved concurrency. If false, the number of operations in transit is limited to one per worker thread.
- `ndb_session_pool_min`: Minimum number of `DBSession` objects per `NdbConnectionPool`.

- `ndb_session_pool_max`: Maximum number of `DBSession` objects per `NdbConnectionPool`.

  Each `NdbConnectionPool` maintains a pool of `DBSession` objects, along with their underlying `Ndb` objects. This parameter, together with `ndb_session_pool_min`, sets guidelines for the size of that pool.

The `TableMapping` constructor is also visible as a top-level function. You can get the mapping either by name, or by using an existing mapping:

```
TableMapping(tableName);

TableMapping(tableMapping);
```

```
openSession(properties, mappings, Function(err, Session) callback);
```

Connect to the data source and get a `Session` in the *`callback`* function. This is equivalent to calling `connect()` (see later in this section), and then calling `getSession()` on the `SessionFactory` that is returned in the callback function.

Note

Executing this method could result in connections being made to many other nodes on the network, waiting for them to become ready, and making multiple requests to them. You should avoid opening new sessions unnecessarily for this reason.

The implementation member of the *`properties`* object determines the implementation of the `Session`.

If *`mappings`* is undefined, null, or an empty array, no mappings are loaded or validated. In this case, any required mappings are loaded and validated when needed during execution. If *`mappings`* contains a string or a constructor function, the metadata for the table (or mapped table) is loaded from the database and validated against the requirements of the mapping.

Multiple tables and constructors may be passed to `openSession()` as elements in an array.

```
connect(properties, mappings, Function(err, SessionFactory) callback);
```

Connect to the data source to obtain a `SessionFactory` in the *`callback`* function. In order to obtain a `Session`, you must then call `getSession()` on this `SessionFactory`, whose implementation is determined by the implementation member of the *`properties`* object.

If *`mappings`* is undefined, null, or an empty array, no mappings are loaded or validated. In this case, any required mappings are loaded and validated when needed. If *`mappings`* contains a string or a constructor function, the metadata for the table (or mapped table) is loaded from the database and validated against the requirements of the mapping.

Multiple tables and constructors may be passed as elements in an array.

```
Array getOpenSessionFactories()
```

Get an array of all the `SessionFactory` objects that have been created by this module.

Note

The following functions are part of the public API but are not intended for application use. They form part of the contract between `Mynode` and `SessionFactory`.

- `Connection()`

- `getConnectionKey()`
- `getConnection()`
- `newConnection()`
- `deleteFactory()`

5.3.6 Session

A session is the main user access path to the database. The `Session` class models such a session.

```
Session extends Context
```

```
getMapping(Object parameter, Function(Object err, Object mapping) callback);
```

Get the mappings for a table or class.

The *`parameter`* may be a table name, a mapped constructor function, or a domain object. This function returns a fully resolved `TableMapping` object.

```
Batch createBatch()
```

Creates a new, empty batch for collecting multiple operations to be executed together. In an application, you can invoke this function similarly to what is shown here:

```
var nosql = require("mysql-js");

var myBatch = nosql.createBatch();
```

```
Array listBatches():
```

Return an array whose elements consist of all current batches belonging to this session.

```
Transaction currentTransaction();
```

Get the current `Transaction`.

```
void close(Function(Object error) callback);
```

Close this session. Must be called when the session is no longer needed.

```
boolean isClosed();
```

Returns true if this session is closed.

```
void setLockMode(String lockMode);
```

Set the lock mode for read operations. This takes effect immediately and remains in effect until the session is closed or this method is called again. *`lockMode`* must be one of `'EXCLUSIVE'`, `'SHARED'`, OR `'NONE'`.

```
Array listTables(databaseName, callback);
```

List all tables in database *`databaseName`*.

```
TableMetadata getTableMetadata(String databaseName, String tableName, callback);
```

Fetch metadata for table *tableName* in database *databaseName*.

5.3.7 SessionFactory

This class is used to generate and manage sessions. A `Session` provides a context for database transactions and operations. Each independent user should have its own session.

```
openSession(Object mappings, Function(Object error, Session session) callback);
```

Open a database session object. Table *mappings* are validated at the beginning of the session. Resources required for sessions are allocated in advance; if those resources are not available, the method returns an error in the callback.

```
Array getOpenSessions();
```

Get all open sessions that have been created by this `SessionFactory`.

```
close(Function(Error err));
```

Close the connection to the database. This ensures proper disconnection. The function passed in is called when the close operation is complete.

5.3.8 TableMapping and FieldMapping

A `TableMapping` describes the mapping of a domain object in the application to a table stored in the database. A *default* table mapping is one which maps each column in a table to a field of the same name.

```
TableMapping = {
  String table                  :  "" ,
  String database               :  "" ,
  boolean mapAllColumns         : true,
  Array fields                  : null
};
```

The `table` and `data` members are the names of the table and database, respectively. `mapAllColumns`, if true, creates a default `FieldMapping` for all columns not listed in `fields`, such that that all columns not explicitly mapped are given a default mapping to a field of the same name. `fields` holds an array of `FieldMapping` objects;this can also be a single `FieldMapping`.

A `FieldMapping` describes a single field in a domain object. There is no public constructor for this object; you can create a FieldMapping using `TableMapping.mapField()`, or you can use `FieldMapping` literals can be used directly in the `TableMapping` constructor.

```
FieldMapping = {
  String fieldName     :  "" ,
  String columnName    :  "" ,
  Boolean persistent    : true,
  Converter converter     : null
};
```

`fieldName` and `columnName` are the names of the field and the column where this field are stored, respectively, in the domain object. If `persistent` is true (the default), the field is stored in the database. `converter` specifies a `Converter` class, if any, to use with this field (defaults to null). };

The `TableMapping` constructor can take either the name of a table (possibly qualified with the database name) or a `TableMapping` literal.

```
TableMapping mapField(String fieldName, [String columnName], [Converter converter], [Boolean persistent])
```

Create a field mapping for a named field of a mapped object. The only mandatory parmeter is *fieldName*, which provides the name a field in a JavaScript application object. The remaining parameters are optional, and may appear in any order. The cyrrent TableMapping object is returned.

columnName specifies the name of the database column that maps to this object field. If omitted, *columnName* defaults to the same value as *fieldName*. A *converter* can be used to supply a Converter class that performs custom conversion between JavaScript and database data types. The default is null. *persistent* specifies whether the field is persisted to the database, and defaults to true.

Important

If *persistent* is false, then the *columnName* and *converter* parameters may not be used.

```
TableMapping applyToClass(Function constuctor)
```

Attach a TableMapping to a *constructor* for mapped objects. After this is done, any object created from the constructor will qualify as a mapped instance, which several forms of the relevant Session and Batch methods can be used.

For example, an application can construct an instance that is only partly complete, then use Session.load() to populate it with all mapped fields from the database. After the application modifies the instance, Session.save() saves it back. Similarly, Session.find() can take the mapped constructor, retrieve an object based on keys, and then use the constructor to create a fully-fledged domain object.

5.3.9 TableMetadata

A TableMetadata object represents a table. This is the object returned in the getTable() callback. indexes[0] represents the table's intrinsic primary key.

```
TableMetadata = {
  database         : ""    ,  // Database name
  name             : ""    ,  // Table Name
  columns          : {}    ,  // ordered array of ColumnMetadata objects
  indexes          : {}    ,  // array of IndexMetadata objects
  partitionKey     : {}    ,  // ordered array of column numbers in the partition key
};
```

ColumnMetadata object represents a table column.

```
ColumnMetadata = {
  /* Required Properties */
  name             : ""    ,  // column name
  columnNumber     : -1    ,  // position of column in table, and in columns array
  columnType       : ""    ,  // a ColumnTypes value
  isIntegral       : false ,  // true if column is some variety of INTEGER type
  isNullable       : false ,  // true if NULLABLE
  isInPrimaryKey   : false ,  // true if column is part of PK
  isInPartitionKey : false ,  // true if column is part of partition key
  columnSpace      : 0     ,  // buffer space required for encoded stored value
  defaultValue     : null  ,  // default value for column: null for default NULL;
                              // undefined for no default; or a type-appropriate
                              // value for column

  /* Optional Properties, depending on columnType */
  /* Group A: Numeric */
  isUnsigned       : false ,  //  true for UNSIGNED
  intSize          : null  ,  //  1,2,3,4, or 8 if column type is INT
  scale            : 0     ,  //  DECIMAL scale
  precision        : 0     ,  //  DECIMAL precision
  isAutoincrement  : false ,  //  true for AUTO_INCREMENT columns
```

```
  /* Group B: Non-numeric */
  length           : 0     ,  //  CHAR or VARCHAR length in characters
  isBinary         : false ,  //  true for BLOB/BINARY/VARBINARY
  charsetNumber    : 0     ,  //  internal number of charset
  charsetName      : ""    ,  //  name of charset
};
```

An `IndexMetadata` object represents a table index. The `indexes` array of `TableMetadata` contains one `IndexMetadata` object per table index.

`NDB` implements a primary key as both an ordered index and a unique index, and might be viewed through the NDB API adapter as two indexes, but through a MySQL adapter as a single index that is both unique and ordered. We tolerate this discrepancy and note that the implementation in `Adapter/api` must treat the two descriptions as equivalent.

```
IndexMetadata = {
  name             : ""    ,  // Index name; undefined for PK
  isPrimaryKey     : true  ,  // true for PK; otherwise undefined
  isUnique         : true  ,  // true or false
  isOrdered        : true  ,  // true or false; can scan if true
  columns          : null  ,  // an ordered array of column numbers
};
```

The `ColumnMetaData` object's `columnType` must be a valid `ColumnTypes` value, as shown in this object's definition here:

```
ColumnTypes =  [
  "TINYINT",
  "SMALLINT",
  "MEDIUMINT",
  "INT",
  "BIGINT",
  "FLOAT",
  "DOUBLE",
  "DECIMAL",
  "CHAR",
  "VARCHAR",
  "BLOB",
  "TEXT",
  "DATE",
  "TIME",
  "DATETIME",
  "YEAR",
  "TIMESTAMP",
  "BIT",
  "BINARY",
  "VARBINARY"
];
```

5.3.10 Transaction

A transaction is always either automatic or explicit. If it is automatic, (autocommit), every operation is performed as part of a new transaction that is automatically committed.

Beginning, committing, and rolling back a transaction

```
begin();
```

Begin a transaction. No arguments are required. If a transaction is already active, an exception is thrown.

```
commit(Function(Object error) callback);
```

Commit a transaction.

This method takes as its sole argument a *`callback`* function that returns an error object.

```
rollback(Function(Object error) callback);
```

Roll back a transaction. Errors are reported in the *`callback`* function.

Transaction information methods

```
Boolean isActive();
```

Determine whether or not a given transaction is currently active. Returns true if a transaction is active, and false otherwise.

`isActive()` requires no arguments.

```
setRollbackOnly();
```

Mark the transaction as rollback-only. Once this is done, `commit()` rolls back the transaction and throws an exception; `rollback()` rolls the transaction back, but does not throw an exception. To mark a transaction as rollback-only, call the `setRollbackOnly()` method, as shown here.

This method is one-way; a transaction marked as rollback-only cannot be unmarked. Invoking `setRollbackOnly()` while in autocommit mode throws an exception. This method requires no arguments.

```
boolean getRollbackOnly();
```

Determine whether a transaction has been marked as rollback-only. Returns true if the transaction has been so marked. `setRollbackOnly()` takes no arguments.

5.4 Using the MySQL JavaScript Connector: Examples

This section contains a number of examples performing basic database operations such as retrieving, inserting, or deleting rows from a table. The source for these files ca also be found in `share/nodejs/samples`, under the NDB Cluster installation directory.

5.4.1 Requirements for the Examples

The software requirements for running the examples found in the next few sections are as follows:

- A working Node.js installation
- Working installations of the `ndb` and `mysql-js` adapters
- The `mysql-js` adapter also requires a working installation of the `node-mysql` driver from https://github.com/felixge/node-mysql/.

Section 5.2, “Installing the JavaScript Connector”, describes the installation process for all three of these requirements.

Sample database, table, and data. All of the examples use a sample table named `tweet`, in the `test` database. This table is defined as in the following `CREATE TABLE` statement:

```
CREATE TABLE IF NOT EXISTS tweet  (
    id CHAR(36) NOT NULL PRIMARY KEY,
    author VARCHAR(20),
    message VARCHAR(140),
    date_created TIMESTAMP,

    KEY idx_btree_date_created (date_created),
    KEY idx_btree_author(author)
)
```

```
ENGINE=NDB;
```

The `tweet` table can be created by running the included SQL script `create.sql` in the `mysql` client. You can do this by invoking `mysql` in your system shell, as shown here:

```
shell> mysql < create.sql
```

All of the examples also make use of two modules defined in the file `lib.js`, whose contents are reproduced here:

```
# FILE: lib.js

"use strict";

var udebug = unified_debug.getLogger("samples/lib.js");
var exec = require("child_process").exec;
var SQL = {};

/* Pseudo random UUID generator */

var randomUUID = function() {
  return 'xxxxxxxx-xxxx-4xxx-yxxx-xxxxxxxxxxxx'.replace(/[xy]/g, function(c) {
    var r = Math.random()*16|0, v = c == 'x' ? r : (r&0x3|0x8);
    return v.toString(16);
    });
};

/* Tweet domain object model */

var Tweet = function(author, message) {
  this.id = randomUUID();
  this.date_created = new Date();
  this.author = author;
  this.message = message;
};

/* SQL DDL Utilities */

var runSQL = function(sqlPath, source, callback) {

  function childProcess(error, stdout, stderr) {
    udebug.log('harness runSQL process completed.');
    udebug.log(source + ' stdout: ' + stdout);
    udebug.log(source + ' stderr: ' + stderr);
    if (error !== null) {
      console.log(source + 'exec error: ' + error);
    } else {
      udebug.log(source + ' exec OK');
    }
    if(callback) {
      callback(error);
    }
  }

  var p = mysql_conn_properties;
  var cmd = 'mysql';
  if(p) {
    if(p.mysql_socket)     { cmd += " --socket=" + p.mysql_socket; }
    else if(p.mysql_port)  { cmd += " --port=" + p.mysql_port; }
    if(p.mysql_host)     { cmd += " -h " + p.mysql_host; }
    if(p.mysql_user)     { cmd += " -u " + p.mysql_user; }
    if(p.mysql_password) { cmd += " --password=" + p.mysql_password; }
  }
  cmd += ' <' + sqlPath;
  udebug.log('harness runSQL forking process...');
  var child = exec(cmd, childProcess);
};

SQL.create =  function(suite, callback) {
  var sqlPath = path.join(suite.path, 'create.sql');
```

```
  udebug.log_detail("createSQL path: " + sqlPath);
  runSQL(sqlPath, 'createSQL', callback);
};

SQL.drop = function(suite, callback) {
  var sqlPath = path.join(suite.path, 'drop.sql');
  udebug.log_detail("dropSQL path: " + sqlPath);
  runSQL(sqlPath, 'dropSQL', callback);
};


/* Exports from this module */
exports.SQL               = SQL;
exports.Tweet             = Tweet;
```

Finally, a module used for random data generation is included in the file `ndb_loader/lib/RandomData.js`, shown here:

```
# FILE: RandomData.js

var assert = require("assert");


function RandomIntGenerator(min, max) {
  assert(max > min);
  var range = max - min;
  this.next = function() {
    var x = Math.floor(Math.random() * range);
    return min + x;
  };
}


function SequentialIntGenerator(startSeq) {
  var seq = startSeq - 1;
  this.next = function() {
    seq += 1;
    return seq;
  };
}


function RandomFloatGenerator(min, max, prec, scale) {
  assert(max > min);
  this.next = function() {
    var x = Math.random();
    /* fixme! */
    return 100 * x;
  };
}


function RandomCharacterGenerator() {
  var intGenerator = new RandomIntGenerator(32, 126);
  this.next = function() {
    return String.fromCharCode(intGenerator.next());
  };
}


function RandomVarcharGenerator(length) {
  var lengthGenerator = new RandomIntGenerator(0, length),
      characterGenerator = new RandomCharacterGenerator();
  this.next = function() {
    var i = 0,
        str = "",
        len = lengthGenerator.next();
    for(; i < len ; i++) str += characterGenerator.next();
    return str;
  }
}
```

```
function RandomCharGenerator(length) {
  var characterGenerator = new RandomCharacterGenerator();
  this.next = function() {
    var i = 0,
        str = "";
    for(; i < length ; i++) str += characterGenerator.next();
    return str;
  };
}


function RandomDateGenerator() {
  var generator = new RandomIntGenerator(0, Date.now());
  this.next = function() {
    return new Date(generator.next());
  };
}


function RandomGeneratorForColumn(column) {
  var g = {},
      min, max, bits;

  switch(column.columnType.toLocaleUpperCase()) {
    case "TINYINT":
    case "SMALLINT":
    case "MEDIUMINT":
    case "INT":
    case "BIGINT":
      if(column.isInPrimaryKey) {
        g = new SequentialIntGenerator(0);
      }
      else {
        bits = column.intSize * 8;
        max = column.isUnsigned ? Math.pow(2,bits)-1 : Math.pow(2, bits-1);
        min = column.isUnsigned ?                  0 : 1 - max;
        g = new RandomIntGenerator(min, max);
      }
      break;
    case "FLOAT":
    case "DOUBLE":
    case "DECIMAL":
      g = new RandomFloatGenerator(0, 100000); // fixme
      break;
    case "CHAR":
      g = new RandomCharGenerator(column.length);
      break;
    case "VARCHAR":
      g = new RandomVarcharGenerator(column.length);
      break;
    case "TIMESTAMP":
      g = new RandomIntGenerator(0, Math.pow(2,32)-1);
      break;
    case "YEAR":
      g = new RandomIntGenerator(1900, 2155);
      break;
    case "DATE":
    case "TIME":
    case "DATETIME":
      g = new RandomDateGenerator();
      break;
    case "BLOB":
    case "TEXT":
    case "BIT":
    case "BINARY":
    case "VARBINARY":
    default:
      throw("UNSUPPORTED COLUMN TYPE " + column.columnType);
      break;
  }
```

```
  return g;
}


function RandomRowGenerator(table) {
  var i = 0,
      generators = [];
  for(; i < table.columns.length ; i++) {
    generators[i] = RandomGeneratorForColumn(table.columns[i]);
  }

  this.newRow = function() {
    var n, col, row = {};
    for(n = 0; n < table.columns.length ; n++) {
      col = table.columns[n];
      row[col.name] = generators[n].next();
    }
    return row;
  };
}

exports.RandomRowGenerator = RandomRowGenerator;
exports.RandomGeneratorForColumn = RandomGeneratorForColumn;
```

5.4.2 Example: Finding Rows

```
# FILE: find.js

var nosql = require('..');
var lib = require('./lib.js');
var adapter = 'ndb';
global.mysql_conn_properties = {};

var user_args = [];

// *** program starts here ***

// analyze command line

var usageMessage =
  "Usage: node find key\n" +
  "          -h or --help: print this message\n" +
  "         -d or --debug: set the debug flag\n" +
  "  --mysql_socket=value: set the mysql socket\n" +
  "    --mysql_port=value: set the mysql port\n" +
  "    --mysql_host=value: set the mysql host\n" +
  "    --mysql_user=value: set the mysql user\n" +
  "--mysql_password=value: set the mysql password\n" +
  "              --detail: set the detail debug flag\n" +
  "   --adapter=<adapter>: run on the named adapter (e.g. ndb or mysql)\n"
  ;

// handle command line arguments
var i, exit, val, values;

for(i = 2; i < process.argv.length ; i++) {
  val = process.argv[i];
  switch (val) {
  case '--debug':
  case '-d':
    unified_debug.on();
    unified_debug.level_debug();
    break;
  case '--detail':
    unified_debug.on();
    unified_debug.level_detail();
    break;
  case '--help':
  case '-h':
    exit = true;
```

```
    break;
  default:
    values = val.split('=');
    if (values.length === 2) {
      switch (values[0]) {
      case '--adapter':
        adapter = values[1];
        break;
      case '--mysql_socket':
        mysql_conn_properties.mysql_socket = values[1];
        break;
      case '--mysql_port':
        mysql_conn_properties.mysql_port = values[1];
        break;
      case '--mysql_host':
        mysql_conn_properties.mysql_host = values[1];
        break;
      case '--mysql_user':
        mysql_conn_properties.mysql_user = values[1];
        break;
      case '--mysql_password':
        mysql_conn_properties.mysql_password = values[1];
        break;
      default:
        console.log('Invalid option ' + val);
        exit = true;
      }
    } else {
      user_args.push(val);
   }
  }
}

if (user_args.length !== 1) {
  console.log(usageMessage);
  process.exit(0);
};

if (exit) {
  console.log(usageMessage);
  process.exit(0);
}

console.log('Running find with adapter', adapter, user_args);
//create a database properties object

var dbProperties = nosql.ConnectionProperties(adapter);

// create a basic mapping
var annotations = new nosql.TableMapping('tweet').applyToClass(lib.Tweet);

//check results of find
var onFind = function(err, object) {
  console.log('onFind.');
  if (err) {
    console.log(err);
  } else {
    console.log('Found: ' + JSON.stringify(object));
  }
  process.exit(0);
};

// find an object
var onSession = function(err, session) {
  if (err) {
    console.log('Error onSession.');
    console.log(err);
    process.exit(0);
  } else {
    session.find(lib.Tweet, user_args[0], onFind);
  }
};
```

```
// connect to the database
nosql.openSession(dbProperties, annotations, onSession);
```

5.4.3 Inserting Rows

```
# FILE: insert.js


var nosql = require('..');
var lib = require('./lib.js');
var adapter = 'ndb';
global.mysql_conn_properties = {};

var user_args = [];
// *** program starts here ***

// analyze command line

var usageMessage =
  "Usage: node insert author message\n" +
  "          -h or --help: print this message\n" +
  "         -d or --debug: set the debug flag\n" +
  "  --mysql_socket=value: set the mysql socket\n" +
  "    --mysql_port=value: set the mysql port\n" +
  "    --mysql_host=value: set the mysql host\n" +
  "    --mysql_user=value: set the mysql user\n" +
  "--mysql_password=value: set the mysql password\n" +
  "              --detail: set the detail debug flag\n" +
  "   --adapter=<adapter>: run on the named adapter (e.g. ndb or mysql)\n"
  ;

// handle command line arguments
var i, exit, val, values;

for(i = 2; i < process.argv.length ; i++) {
  val = process.argv[i];
  switch (val) {
  case '--debug':
  case '-d':
    unified_debug.on();
    unified_debug.level_debug();
    break;
  case '--detail':
    unified_debug.on();
    unified_debug.level_detail();
    break;
  case '--help':
  case '-h':
    exit = true;
    break;
  default:
    values = val.split('=');
    if (values.length === 2) {
      switch (values[0]) {
      case '--adapter':
        adapter = values[1];
        break;
      case '--mysql_socket':
        mysql_conn_properties.mysql_socket = values[1];
        break;
      case '--mysql_port':
        mysql_conn_properties.mysql_port = values[1];
        break;
      case '--mysql_host':
        mysql_conn_properties.mysql_host = values[1];
        break;
      case '--mysql_user':
        mysql_conn_properties.mysql_user = values[1];
        break;
      case '--mysql_password':
```

```
        mysql_conn_properties.mysql_password = values[1];
        break;
      default:
        console.log('Invalid option ' + val);
        exit = true;
      }
    } else {
      user_args.push(val);
   }
  }
}

if (user_args.length !== 2) {
  console.log(usageMessage);
  process.exit(0);
};

if (exit) {
  console.log(usageMessage);
  process.exit(0);
}

console.log('Running insert with adapter', adapter, user_args);
//create a database properties object

var dbProperties = nosql.ConnectionProperties(adapter);

// create a basic mapping
var annotations = new nosql.TableMapping('tweet').applyToClass(lib.Tweet);

//check results of insert
var onInsert = function(err, object) {
  console.log('onInsert.');
  if (err) {
    console.log(err);
  } else {
    console.log('Inserted: ' + JSON.stringify(object));
  }
  process.exit(0);
};

// insert an object
var onSession = function(err, session) {
  if (err) {
    console.log('Error onSession.');
    console.log(err);
    process.exit(0);
  } else {
    var data = new lib.Tweet(user_args[0], user_args[1]);
    session.persist(data, onInsert, data);
  }
};

// connect to the database
nosql.openSession(dbProperties, annotations, onSession);
```

5.4.4 Deleting Rows

```
FILE: delete.js


var nosql = require('..');
var lib = require('./lib.js');
var adapter = 'ndb';
global.mysql_conn_properties = {};

var user_args = [];
// *** program starts here ***

// analyze command line
```

```
var usageMessage =
  "Usage: node delete message-id\n" +
  "          -h or --help: print this message\n" +
  "         -d or --debug: set the debug flag\n" +
  "  --mysql_socket=value: set the mysql socket\n" +
  "    --mysql_port=value: set the mysql port\n" +
  "    --mysql_host=value: set the mysql host\n" +
  "    --mysql_user=value: set the mysql user\n" +
  "--mysql_password=value: set the mysql password\n" +
  "              --detail: set the detail debug flag\n" +
  "   --adapter=<adapter>: run on the named adapter (e.g. ndb or mysql)\n"
  ;

// handle command line arguments
var i, exit, val, values;

for(i = 2; i < process.argv.length ; i++) {
  val = process.argv[i];
  switch (val) {
  case '--debug':
  case '-d':
    unified_debug.on();
    unified_debug.level_debug();
    break;
  case '--detail':
    unified_debug.on();
    unified_debug.level_detail();
    break;
  case '--help':
  case '-h':
    exit = true;
    break;
  default:
    values = val.split('=');
    if (values.length === 2) {
      switch (values[0]) {
      case '--adapter':
        adapter = values[1];
        break;
      case '--mysql_socket':
        mysql_conn_properties.mysql_socket = values[1];
        break;
      case '--mysql_port':
        mysql_conn_properties.mysql_port = values[1];
        break;
      case '--mysql_host':
        mysql_conn_properties.mysql_host = values[1];
        break;
      case '--mysql_user':
        mysql_conn_properties.mysql_user = values[1];
        break;
      case '--mysql_password':
        mysql_conn_properties.mysql_password = values[1];
        break;
      default:
        console.log('Invalid option ' + val);
        exit = true;
      }
    } else {
      user_args.push(val);
   }
  }
}

if (user_args.length !== 1) {
  console.log(usageMessage);
  process.exit(0);
};

if (exit) {
  console.log(usageMessage);
  process.exit(0);
```

```
}

console.log('Running delete with adapter', adapter, user_args);
//create a database properties object

var dbProperties = nosql.ConnectionProperties(adapter);

// create a basic mapping
var annotations = new nosql.TableMapping('tweet').applyToClass(lib.Tweet);

// check results of delete
var onDelete = function(err, object) {
  console.log('onDelete.');
  if (err) {
    console.log(err);
  } else {
    console.log('Deleted: ' + JSON.stringify(object));
  }
  process.exit(0);
};

// delete an object
var onSession = function(err, session) {
  if (err) {
    console.log('Error onSession.');
    console.log(err);
    process.exit(0);
  } else {
    var tweet = new lib.Tweet();
    tweet.id = user_args[0];
    session.remove(tweet, onDelete, user_args[0]);
  }
};

// connect to the database
nosql.openSession(dbProperties, annotations, onSession);
```

Chapter 6 ndbmemcache—Memcache API for NDB Cluster

Table of Contents

6.1 Overview 643
6.2 Compiling NDB Cluster with Memcache Support 643
6.3 `memcached` command line options 643
6.4 NDB Engine Configuration 644
6.5 Memcache protocol commands 650
6.6 The memcached log file 652
6.7 Known Issues and Limitations of ndbmemcache 654

This section provides information about the Memcache API for NDB Cluster, available beginning with NDB 7.2.2, which makes it possible to access NDB Cluster data using the memcache protocol.

6.1 Overview

Memcached is a distributed in-memory caching server using a simple text-based protocol, commonly used for key-value data stores, with clients available for many platforms and programming languages. The most recent release of the `memcached` server is available from memcached.org.

The Memcache API for NDB Cluster is available beginning with NDB 7.2.2. This API is implemented as a loadable storage engine for memcached version 1.6 and later, which employs a storage engine architecture. This API can be used to provide a persistent NDB Cluster data store which is accessible employing the memcache protocol. It is also possible for the `memcached` server to provide a strictly defined interface to existing NDB Cluster tables such that an administrator can control exactly which tables and columns are referenced by particular memcache keys and values, and which operations are allowed on these keys and values.

The standard `memcached` caching engine is included in the NDB Cluster distribution. Each memcache server, in addition to providing direct access to data stored in NDB Cluster, is able to cache data locally and serve (some) requests from this local cache. As with table and column mappings, cache policies are configurable based on a prefix of a memcache key.

6.2 Compiling NDB Cluster with Memcache Support

Support for the Memcache API is built automatically using the memcached and libevent sources included in the NDB Cluster sources when compiling NDB 7.2.3 or later from source. By default, `make install` places the `memcached` binary in the NDB Cluster installation `bin` directory, and the ndbmemcache engine shared object file `ndb_engine.so` in the installation `lib` directory.

You can disable use of the bundled memcached when building ndbmemcache, by using `-DWITH_BUNDLED_MEMCACHED=OFF`; you can instead use your own system's memcached server and sources, installed in *`path`*, with `-DWITH_BUNDLED_MEMCACHED=OFF -DMEMCACHED_HOME=`*`path`*. You can also cause your system's version of libevent to be used, rather than the version bundled with NDB Cluster, by using the `-DWITH_BUNDLED_LIBEVENT=OFF` option.

For additional information about `CMake` options relating to ndbmemcache support, see Options for Compiling NDB Cluster.

For general information about building NDB Cluster, see Building NDB Cluster from Source on Linux, and Compiling and Installing NDB Cluster from Source on Windows. For information about building MySQL Server from source, see Installing MySQL from Source, as well as MySQL Source-Configuration Options.

6.3 `memcached` command line options

The following list contains `memcached` command line options that are of particular interest or usefulness when working with ndbmemcache.

- `-E so_file`

  Specifies an engine (module) to be dynamically loaded on startup by memcached (version 1.6 or later).

  If this option is not specified, memcached tries to load the default engine, which provides the same caching engine as used in memcached 1.4 and previous versions

  To load the NDB engine, use this option as shown here:

  ```
  -E /path/to/ndb_engine.so
  ```

- `-e "configuration_string"`

  Specifies options for use by the loaded engine. Options are given as *`option=value`* pairs separated by semicolons. The complete string should be quoted to prevent the possibility that the shell might interpret the semicolon as a command separator. All options to be passed to the NDB memcached engine must be specified in this fashion, as shown in the following example:

  ```
  shell> memcached -E lib/ndb_engine.so -e "connectstring=maddy:1186;role=dev"
  ```

  See Section 6.4, “NDB Engine Configuration” for a list of NDB memcached engine configuration options.

- `-t number_of_worker_threads`

  Sets the number of worker threads to be used by memcached. Because memcached uses an event-driven model in which each worker thread should be able to saturate a CPU core, the number of worker threads should be approximately the same as the number of CPU cores that memcached is to use.

  In some cases, adding worker threads does not improve performance unless you also provide additional connections to NDB Cluster. The default (4 memcached threads and 2 cluster connections) should work in most cases.

- `-p tcp_port`

  The default TCP port is port 11211.

- `-U udb_port`

  The default UDP port is port 11211. Setting this option to 0 disables UDP support.

- `-h`

  Causes `memcached` to print help information.

For general information `memcached` command line options, see the documentation at http://code.google.com/p/memcached/wiki/NewStart.

6.4 NDB Engine Configuration

NDB memcache engine configuration options. The NDB engine supports the following configuration options for use with `memcache -e` (see Section 6.3, “`memcached` command line options”):

- `debug={true|false}`

Enables writing of debug tracing output to `stderr` or the memcached log file, as shown in this example:

```
shell> memcached -E lib/ndb_engine.so -e "debug=true"
```

Because the debug output can be quite large, you should enable this option as a diagnostic tool only, and not in production.

By default, this option is `false`.

- `connectstring=connect_string`

  This option takes as its value an NDB Cluster connection string (see NDB Cluster Connection Strings) pointing to the primary NDB Cluster—that is, the NDB Cluster in which the `ndbmemcache` configuration database is stored, as shown here:

  ```
  shell > memcached -E lib/ndb_engine.so -e "connectstring=sam:1186;debug=true"
  ```

  The default value is `localhost:1186`.

- `reconf={true|false}`

  Enables online reconfiguration (reloading of the configuration stored in the `ndbmemcache` information database).

  This option is enabled (`true`) by default.

- `role=role_name`

  Sets the role assumed by this memcached server. A role corresponds to a set of key-prefix mappings described in the `ndbmemcache` configuration database, identified by a *`role_name`* found in the `ndbmemcache.memcache_server_roles` table.

  The default role is `default_role`.

  An example is shown here:

  ```
  shell> memcached -E lib/ndb_engine.so -e "role=db-only"
  ```

- `scheduler=scheduler_name:scheduler_options`

  This option controls some advanced aspects of how the NDB engine sends requests to NDB Cluster. The *`scheduler_name`* of the default scheduler or *S-scheduler* is `S`. An S-scheduler option takes the form of a single letter followed by a number; multiple S-scheduler options are separated by commas. In most cases, the default value `S:c0,f0,t1` is sufficient.

  These S-scheduler options are described in the following list:

  - `c`: Number of connections to `NDB`. Possible values are in the range 0-4 inclusive, with 0 (the default) causing this number to be calculated automatically. Using 1, 2, 3, or 4 causes that number of connections to be created.
  - `f`: Can be either 0 or 1; setting to 1 enables force-send. The default is 0 (force-send disabled).
  - `t`: Sets the send-thread timer to 1-10 milliseconds (inclusive). The default is 1.

Initial Configuration.
When a the NDB engine starts up, its most important command-line arguments are the cluster connection string and server role. The connection string is used to connect to a particular cluster, called the primary cluster, which contains a configuration schema. The tables in the configuration schema

are read to retrieve a set of key-prefix mappings for the given server role (see the ndbmemcache configuration schema). Those mappings instruct the server how to respond to memcache operations on particular keys, based on the leftmost part of the key. For instance, they may specify that data is stored in particular columns of a certain table. This table may be stored in the same cluster as the configuration schema, or in a different cluster. A memcache server may have connections to several different clusters, and many memcache servers may connect to a single cluster but with a variety of roles.

The ndbmemcache configuration schema. When the memcache NDB engine starts up, it connects to a cluster, and looks for the ndbmemcache configuration schema there. If the schema is not found, it shuts down.

The schema is described (with full comments) in the file ndb_memcache_metadata.sql

The main concept of the schema is a key-prefix mapping. This takes a prefix of a memcache key and maps it to a specific container table, on a particular cluster, with a particular cache policy.

A server role is defined as a set of key-prefix mappings that a memcached server will implement.

Whenever a memcached server is started with a particular server role (from the command-line arguments), that server role must exist in the ndbmemcache.server_roles table.

The following table lists table names and descriptions for tables that belong to the `ndbmemcache` configuration schema.

Table Name	Description
`meta`	The meta table describes the version number of the ndbmemcache tables. It should be considered as a read-only table.
`ndb_clusters`	For each cluster, this table holds a numeric cluster-id and a connection string. The microsec_rtt column is used for performance tuning. It is recommended to use the default value of this column. See Autotuning.
`cache_policies`	This table maps a policy name to a set of get, set, delete, and flush policies. The `policy_name` column is used as the key (there is no numeric policy id). Additional information about cache policies can found in the text following the table.
`containers`	The containers table describes how the memcached server can use a database table to store data. Additional information about containers can found in the text following the table.
`memcache_server_roles`	The `memcache_server_roles` table maps a role name to a numeric ID and a `max_tps` specifier, which is used for performance tuning. See Autotuning. It is recommended to use the default value. This table also has an update_timestamp column. This column can be updated to enable online reconfiguration. See Online reconfiguration. Additional information about server roles can found in the text following the table.
`key_prefixes`	In this table, the leftmost part of a memcache key is paired with a cluster ID, container, and cache policy to make a *key prefix mapping*. Additional information about key prefix mappings can found in the text following the table.

Cache policies. There are four policy types: `get_policy`, `set_policy`, `delete_policy`, and `flush_from_db`. These are described in the following paragraphs.

`get_policy` determines how the memcached server interprets `GET` commands. Possible values and their meanings are shown in the following list:

- `cache_only`: The server searches in its local cache only.
- `ndb_only`: The server searches in the NDB Cluster database only.
- `caching`: The server searches the local cache first, then the NDB Cluster database.
- `disabled`: `GET` commands are not permitted.

The `set_policy` determines how the memcached server interprets `SET`, `INSERT`, and `REPLACE` commands. Possible `set_policy` values and their meanings are listed here:

- `cache_only`: The server updates the value in its local cache only.
- `ndb_only`: The server updates the value stored in NDB Cluster only.
- `caching`: The server updates the value stored in NDB Cluster, and then stores a copy of that value in its local cache.
- `disabled`: `SET`, `INSERT`, and `REPLACE` commands are not allowed.

`delete_policy` describes how the memcached server interprets `DELETE` commands. It can take on the values shown and described in the following list:

- `cache_only`: The server deletes the value from its local cache only.
- `ndb_only`: The server deletes the value from the NDB Cluster database only.
- `caching`: The server deletes the value from both the database and its local cache.
- `disabled`: `DELETE` operations are not allowe.

`flush_from_db` determines how the memcached server interprets a `FLUSH_ALL` command with regard to data stored in the NDB Cluster database, as shown here:

- `true`: `FLUSH_ALL` commands cause data to be deleted from the NDB Cluster database.
- `false`: `FLUSH_ALL` commands do not affect the NDB Cluster database.

containers table columns. The columns in the `containers` table are described in the following list:

- `name`: Name of container; primary key of table.
- `db_schema`: Name of database (schema) holding container table.
- `db_table`: table name of container table.
- `key_columns`: List of columns that map to the memcache key. Most keys are one-part keys, but a key can have up to four parts, in which case multiple columns are listed and separated by commas.
- `value_columns`: List of columns that map to the memcache value. It can also contain a comma-separated list of up to 16 value columns.
- `flags`: Currently unimplemented; it is intended hold either a numeric value which is used as the memcache `FLAGS` value for the entire container, or the name of that column of the container table used to store this value.
- `increment_column`: Name of the column in the container table which stores the numeric value used in memcached `INCR` and `DECR` operations. If set, this must be a `BIGINT UNSIGNED` column.

- `cas_column` Name of the column in the container table storing the memcache CAS value. If set, it must be a `BIGINT UNSIGNED` column.
- `expire_time_column`: Currently unimplemented.

Key mappings.

- server_role_id is a numeric server role identifier which references the memcache_server_roles table
- key_prefix is a string that corresponds to the leftmost part of the memcache key. If this string is empty, then the defined prefix will be the "default prefix". The default prefix matches any memcache key that does not match some more specific prefix.
- cluster_id is an int that references the ndb_clusters table
- policy is a string that references a policy name in the cache_policies table
- container is a container name that references the containers table

The following table lists table names and descriptions for non-configuration `ndbmemcache` logging and container tables.

Table Name	Description
`last_memcached_signon`	This table is not part of the configuration schema, but is an informative logging table. It records the most recent login time of each memcached server using the configuration. • ndb_node_id is an int recording the API node id of the server • hostname is the hostname of the memcached server • server_role is the server role of the server at signon time • signon_time is a timestamp recording the memcached startup time In the case of online reconfiguration, signon_time records the time of the latest reconfiguration, not the time of startup. This is an unintended consequence and might be considered a bug.
`demo_table`	demo_table is the container table used with default key prefix in the default server role. It is used to demonstrate SET and GET operations as well as INCR, DECR, and CAS, with one key column and one value column.
`demo_table_tabs`	demo_table_tabs is the container table for the "demo_tabs" container, which is used with the key prefix "t:" in the default server role. It is used to demonstrate one key column with multiple value columns. In memcache operations, the value columns are represented as a tab-separated list of values.

Predefined configuration objects

Predefined clusters. A single ndb_cluster record is predefined, referring to the primary cluster (the one where configuration data is stored) as cluster id 0. Id 0 should always be reserved for the primary cluster.

Predefined cache policies

- "memcache-only" : a policy in which all memcache operations are to use local cache only
- "ndb-only" : a policy in which all memcache operations use the NDB Cluster database, except for FLUSH_ALL, which is disabled
- "caching" : a policy with get_policy, set_policy, and delete_policy all set to "caching". FLUSH_ALL is disabled.

- "caching-with-local-deletes": a policy in which get_policy and set_policy are set to caching, but delete_policy is set to "cache-only", and FLUSH_ALL is disabled.
- "ndb-read-only": a policy in which get_policy is set to ndb_only, so that memcache GET operations use the database, but all other memcache operations are disabled
- "ndb-test": a policy like "ndb-only" with the difference that FLUSH_ALL is allowed (flush_from_db) is true. This is the only predefined policy with flush_from_db enabled. This policy is enabled by default for the default server role, so taht the entire memcache command set can be demonstrated.

Predefined containers

- "demo_table": a container using the table ndbmemcache.demo_table as a container table
- "demo_tabs": a container using the table ndbmemcache.demo_table_tabs as a container table

Predefined memcache server roles and their key prefixes

- "default_role" (role id 0)

  "": The empty (default) prefix uses the ndb-test policy and the demo_table container

  "mc:" Memcache keys beginning with "mc:" are treated according to the memcache-only cache policy

  "t:" Memcache keys beginning with "t:" use the ndb-test cache policy and the demo_tabs container

- The "db-only" role (role id 1)

  "": the empty (default) prefix uses the ndb-only role and demo_table container

  The "t:" prefix uses the ndb-only role and demo_tabs container

- The "mc-only" role (role id 2)

  "": The empty (default) prefix uses local caching only for all keys

- The "ndb-caching" role (role id 3)

  "": The empty (default) prefix uses the "caching" cache policy and "demo_table" container for all keys

Configuration versioning and upgrade.
The configuration schema is versioned, and the version number is stored in the ndbmemcache.meta table. The NDB Engine begins the configuration process by reading the schema version number from this table. As a rule, newer versions of the NDB engine will remain compatible with older versions of the configuration schema.

STABILITY NOTE: consider this section "unstable" & subject to change

Performance Tuning.
Two parameters are used to tune performance of the NDB memcache engine. The parameters are stored in the configuration schema: the "usec_rtt" value of a particular cluster, and the "max_tps" value of a memcache server role. These values are currently used in two ways: to configure the number of connections to each cluster, and to configure a particular fixed number of concurrent operations supported from each connection.

Autotuning. Autotuning uses an estimated round trip time between cluster data nodes and a target rate of throughput to determine the ideal number of cluster connections and transactions per connection for a given workload. Autotuning parameters are described in the next few paragraphs.

- `usec_rtt`: The round trip time, in microseconds, between cluster nodes. The default value is 250, which is typical for an NDB Cluster on a local switched ethernet. To represent a cluster with higher inter-node latency (wider area), a higher value should be used.

- `max_tps`: The desired throughput from a server. This value is a heuristic, and does not in any way express either a floor or a ceiling on the actual throughput obtained. The default value (100000) is reasonable in most cases.

These values are used, as described in the next few paragraphs, to calculate an optimum number of cluster connections with a given transactions-per-second capacity..

Number of cluster connections. The NDB Engine scheduler attempts to open 1 cluster connection per 50000 transactions per second (TPS). This behavior can be overridden by using a scheduler configuration string (see Section 6.4, “NDB Engine Configuration”.) If the scheduler fails to open a second or subsequent connection to a cluster—for example, because a node id is not available—this is not a fatal error; it will run with only the connections actually opened.

Number of transactions per connection. We assume that a transaction takes 5 times the cluster round trip time to complete. We can obtain the total number of in-flight transactions by dividing the server's `max_tps` by `5 * rtt` (in seconds). These in-flight transaction objects are evenly distributed among the cluster connections.

Tuning example. The following example starts with the default values `usec_rtt` = 250 and `max_tps` = 100000, and assumes a memcached server with 4 worker threads.

- 100000 TPS divided by 50000 is 2, and the server opens two NDB cluster connections.
- Transaction time in microseconds = 250 µs round trip time * 5 round trips = 1250 µs.
- Transactions per connection per second = 1000000 / tx_time_in_µsec = 1000000 / 1250 = 800.
- Total Ndb objects = max_tps / tx_per_ndb_per_sec = 100000 / 800 = 125.
- 125 Ndb objects / 2 connections = 63 Ndb objects per connection (rounding upward).
- (Rounding upward once more) each of 4 worker threads gets 32 Ndb objects

Online reconfiguration.
It is possible to reconfigure the key-prefix mappings of a running NDB engine without restarting it. This is done by committing a change to the configuration schema, and then updating the `update_timestamp` column of a particular server role in the memcache server roles table. The updating of the timestamp causes an event trigger to fire, so that the memcache server receives notification of the event.

Online reconfiguration can be disabled by using the `-e reconf=false` option on the command line.

Online reconfiguration can be used to connect to new clusters and to create new key-prefix mappings. However, it cannot be used to reset autotuning values on existing connections.

Online reconfiguration is a risky operation that could result in memcache server crashes or data corruption, and is used extensively in the mysql test suite. However, it is not recommended for reconfiguring a production server under load.

The `stats reconf` command can be run before and after online reconfiguration to verify that the version number of the running configuration has increased. Verification of reconfiguration is also written into the memcached log file.

6.5 Memcache protocol commands

The NDB engine supports the complete set of memcache protocol commands. When a newly installed server is started with the default server role and configuration schema, you should be able to run `memcapable`, a memcache-server verification tool, and see all tests pass. After a configuration has been customized, however—for instance, by disabling the FLUSH_ALL command—some `memcapable` tests are expected to fail.

GET, SET, ADD, REPLACE, and DELETE operations. Each of these operations is always performed according to a cache policy associated with the memcache key prefix. It may operate on a locally cached item, an item stored in the database, or both. If an operation has been disabled for the prefix, the developer should be sure to test the disabled operation, since it may fail silently, or with a misleading response code.

CAS. *CAS*, in the memcache protocol, refers to a “compare and set” value, which is used as a sort of version number on a cached value, and enables some optimistic application behavior

If a container includes a CAS column, the ndb engine will generate a unique CAS ID every time it writes a data value, and store it in the CAS column.

Some memcache operations include CAS checks, such as the ASCII CAS update which has the semantics “update this value, but only if its CAS id matches the CAS id in the request”. These operations are supported by the NDB engine. The check of the stored CAS ID against the application's CAS ID is performed in an atomic operation on the NDB data node. This allows CAS checks to work correctly even when multiple memcached servers access the same key-value pair.

If CAS ID checks are in use, and additional NDB Cluster APIs other than memcached are being used to manipulate the data, then the applications using those APIs are responsible for invalidating the stored CAS IDs whenever they update data. They can do this by setting the stored CAS ID value to 0 or `NULL`.

The CAS ID is generated using a scheme that attempts to prevent different servers from generating overlapping IDs. This scheme can be considered a best effort, but not a guarantee, of uniqueness. The scheme constructs an initial CAS as follows:

Part of the 32-bit Cluster GCI from the primary cluster at memcached startup time is used for the high-order bits of the 64-bit CAS ID

Part of the unique cluster node id in the primary cluster used when fetching configuration is used for middle-order bits of the CAS ID

An incrementing counter in the low-order bits of the CAS ID is at least 28-bits wide.

While the NDB engine generates one sequence of CAS IDs, the default engine—used for caching values in local memcached servers—generates a different sequence. Not all combinations of CAS behavior and cache policies have been tested, so any application developer wishing to use CAS should thoroughly test whether a particular configuration behaves as desired.

FLUSH_ALL. FLUSH_ALL is implemented as follows: First, the NDB engine iterates over all configured key-prefixes. For any prefix whose cache policy enables a database flush (`flush_from_db` is `true`), it performs a scanning delete of every row in that prefix's container table. Other prefixes are ignored. This can be a slow operation if the table is large, and some memcache clients may time out before the `DELETE` operation is complete. After all database deletes are complete, the `FLUSH_ALL` command is forwarded to the standard caching engine, which sets a flag invalidating all cached data.

INCR and DECR. All `INCR` and `DECR` operations are pushed down to the NDB data nodes and performed atomically there. This allows multiple memcached servers to increment or decrement the same key and be guaranteed a unique value each time.

The `INCR` and `DECR` operations have clearer and more useful semantics in the binary memcache protocol than in the ASCII protocol. The binary protocol is recommended.

The memcached ASCII protocol introduces some ambiguities in the handling of `INCR` and `DECR`, and forces the NDB engine to work in `dup_numbers` mode, in which the `value_column` and the `math_column` must mirror each other.

`dup_numbers` mode is enabled for key prefixes that meet all of the following conditions:

- The container includes a math column, AND
- The container includes a single value column, AND
- The data type of the value column is non-numeric

In `dup_numbers` mode, the following special behavior applies:

- Whenever an ASCII `SET`, `ADD`, or `REPLACE` command sets a value that could be interpreted as numeric, and the container defines a math_column, then the text value is stored in the value column and the numeric value is also stored in the math column.
- Whenever an ASCII `INCR` or `DECR` command is performed, the text value in that container's value column is set to `NULL`.
- Whenever a memcached `GET` command is issued, and the container's value column is `NULL`, but the container's math column is not `NULL`, then the math value is returned to the client.

APPEND and PREPEND. The memcache `APPEND` and `PREPEND` operations are implemented as a single transaction which involves a read of the existing value with an exclusive lock, followed by a write of the new value. The read and write are grouped atomically into a transaction, but unlike `INCR` and `DECR`, which can run natively on the data nodes, `APPEND` and `PREPEND` are executed inside the memcached server. This means that multiple memcached servers can contend to `APPEND` and `PREPEND` the same value, and that no updates will be lost, but this contention relies on locking behavior that could cause noticably increased latency.

STATS. A memcached server can provide many sets of statistics; use `STATS` *`KEYWORD`* from a login shell.

All statistics usually available from the memcached 1.6 core and the default engine are available. For instance, `STATS`, `STATS SLABS`, and `STATS SETTINGS` are all currently supported as described in the memcached documentation. Some special sets of statistics are available from the `NDB` engine, using the `STATS` commands described in the following list:

- `STATS NDB`: Returns NDB API statistics for each NDB cluster connection. These are the same internal statistics which are available as system status variables from the MySQL Server. See NDB API Statistics Counters and Variables, for more information.
- `STATS SCHEDULER`: Returns statistics for the S scheduler. All of these statistics are reported on the cluster connection level.
  - `cl%d.conn%d.sent_operations`: Records the number of operations sent from the connection's send thread to the cluster data nodes.
  - `cl%d.conn%d.batches`: Records the number of operation batches sent from the send thread to the data nodes. Each batch contains one or more operations. `sent_operations` / `batches` can be used to compute the average batch size.
  - `cl%d.conn%d.timeout_races`: This records a rare race condition that may occur in the send thread. It is expected to be 0, or to be a very low number compared to sent_operations.
- `stats reconf`: If the `NDB` engine is currently loading a new configuration, command returns the single-line message `Loading` *`revno`*, where *`revno`* is the version number of the configuration being loaded.

  Otherwise, this command returns the statistical message `Running` *`revno`*.

  `revno` starts at 1 when the memcached server begins running, and is incremented by 1 for each online reconfiguration.

6.6 The memcached log file

Whenever the NDB memcache engine is initialized, it writes a message including a timestamp and version number to its log file, as shown here:

```
12-Oct-2011 13:40:00 PDT NDB Memcache 5.5.15-ndb-7.2.1 started
[NDB 7.2.1; MySQL 5.5.15]
```

It also logs its attempt to connect to a primary cluster:

```
Contacting primary management server (localhost:1186) ...
·Connected to "localhost:1186" as node id 4.
```

Upon successfully fetching initial configuration data, the memcache engine logs a summary message describing the configuration similar to what is shown here:

```
Retrieved 3 key prefixes for server role "default_role"
The default behavior is that:
 GET uses NDB only
 SET uses NDB only
 DELETE uses NDB only
The 2 explicitly defined key prefixes are "mc:" () and "t:" (demo_table_tabs)
Server started with 4 threads.
```

The memcache engine also logs the establishment of each additional cluster connection, as shown here:

```
Connected to "" as node id 5.
```

A `priming the pump...` message indicates that the engine is about to prefetch a pool of transaction objects (API Connect Records). It is followed by a `done ...` message indicating how much time was used by prefetching. The server is not ready to respond to clients until after the prefetching is completed.

```
Priming the pump ...
Scheduler: using 2 connections to cluster 0
Scheduler: starting for 1 cluster; c0,f0,t1
done [0.579 sec].
```

Once the NDB engine has finished initializing, memcached prints a message verifying that the engine was loaded, and enumerating some of its features:

```
Loaded engine: NDB Memcache 5.5.15-ndb-7.2.1
Supplying the following features: compare and swap, persistent storage, LRU
```

If online reconfiguration is enabled, the NDB engine logs each reconfiguration, along with a summary of the new configuration, similar to what is shown here:

```
Received update to server role default_role
Retrieved 3 key prefixes for server role "default_role".
The default behavior is that:
 GET uses NDB only
 SET uses NDB only
 DELETE uses NDB only.
The 2 explicitly defined key prefixes are "mc:" () and "t:" (demo_table_tabs)
ONLINE RECONFIGURATION COMPLETE
```

On shutdown, memcached logs the shutdown sequence'a initialization and completion, and the NDB engine's scheduler logs its own shutdown as well:

```
Initiating shutdown
Shutting down scheduler.
```

```
Shutdown completed.
```

6.7 Known Issues and Limitations of ndbmemcache

This section provides information about known issues with and design limitations of the Memcache API for NDB Cluster.

Problems with AUTO_INCREMENT. ndbmemcache bypasses the `NDB` storage engine's mechanism for handling `AUTO_INCREMENT` columns. This means that, when you insert rows using ndbmemcache into a table having an `AUTO_INCREMENT` column, this column is not automatically updated. This can lead to duplicate key errors when inserts are performed later using SQL in a MySQL client application such as `mysql`.

To work around this issue, you can employ a sequence generator as described here.

Online schema changes not supported. The memcached daemon does not detect online schema changes; after making such changes, you must restart the memcached daemon before the updated schema can be used by your application.

Fractional seconds. Beginning with NDB 7.2.3, ndbmemcache supports the use of fractional seconds with the `TIME`, `DATE`, and `DATETIME` data types as implemented in MySQL 5.6.4 and later. ndbmemcache in previous versions of NDB Cluster does not support fractional seconds. (Bug #16593493)

Index

A

AbortOption (NdbOperation data type), 195
ACC
  and NDB Kernel, 12
  defined, 3
Access Manager
  defined, 3
ActiveHook (NdbBlob data type), 134
addColumn() (method of Index), 92
addColumn() (method of Table), 267
addColumnName() (method of Index), 92
addColumnNames() (method of Index), 93
addEventColumn() (method of Event), 70
addEventColumns() (method of Event), 70
addTableEvent() (method of Event), 71
add_reg() (method of NdbInterpretedCode), 176
add_val() (method of NdbInterpretedCode), 176
aggregate() (method of Table), 267
allowsNull (ClusterJ), 599
and (ClusterJ), 610
Annotations (ClusterJ)
  Column, 598
  Columns, 599
  Extension, 600
  Extensions, 601
  Index, 602
  Indices, 602
  Lob, 603
  NotPersistent, 604
  PartitionKey, 605
  PersistenceCapable, 606
  Persistent, 607
  PrimaryKey, 608
API documentation
  JavaScript, 622
API node
  defined, 3
append (ClusterJ), 577
application-level partitioning, 121
applications
  structure, 4
applyToClass() (method of TableMapping), 631
aRef() (method of NdbRecAttr), 208
ArrayType (Column data type), 27
AutoGrowSpecification
  NDB API structure, 23
AUTO_INCREMENT
  ndbmemcache, 654

B

backup
  defined, 2
Batch class (Connector for JavaScript), 623
Batch.clear(), 623
Batch.execute(), 623
Batch.getSession(), 623
begin (ClusterJ), 597
begin() (method of NdbScanFilter), 216
begin() (method of Transaction), 632
beginSchemaTrans() (method of Dictionary), 51
between (ClusterJ), 611
BinaryCondition (NdbScanFilter data type), 217
BLOB handling
  example, 444
  example (using NdbRecord), 452
blobsFirstBlob() (method of NdbBlob), 134
blobsNextBlob() (method of NdbBlob), 134
BoundType (NdbIndexScanOperation data type), 163
branch_col_and_mask_eq_mask() (method of NdbInterpretedCode), 177
branch_col_and_mask_eq_zero() (method of NdbInterpretedCode), 177
branch_col_and_mask_ne_mask() (method of NdbInterpretedCode), 177
branch_col_and_mask_ne_zero() (method of NdbInterpretedCode), 178
branch_col_eq() (method of NdbInterpretedCode), 178
branch_col_eq_null() (method of NdbInterpretedCode), 179
branch_col_ge() (method of NdbInterpretedCode), 179
branch_col_gt() (method of NdbInterpretedCode), 179
branch_col_le() (method of NdbInterpretedCode), 180
branch_col_like() (method of NdbInterpretedCode), 180
branch_col_lt() (method of NdbInterpretedCode), 181
branch_col_ne() (method of NdbInterpretedCode), 181
branch_col_ne_null() (method of NdbInterpretedCode), 182
branch_col_notlike() (method of NdbInterpretedCode), 182
branch_eq() (method of NdbInterpretedCode), 182
branch_eq_null() (method of NdbInterpretedCode), 183
branch_ge() (method of NdbInterpretedCode), 183
branch_gt() (method of NdbInterpretedCode), 183
branch_label() (method of NdbInterpretedCode), 184
branch_le() (method of NdbInterpretedCode), 184
branch_lt() (method of NdbInterpretedCode), 184
branch_ne() (method of NdbInterpretedCode), 185
branch_ne_null() (method of NdbInterpretedCode), 185

C

call_sub() (method of NdbInterpretedCode), 185
charsetName (ClusterJ), 563
char_value() (method of NdbRecAttr), 208
checkpoint
  defined, 2
Classes (ClusterJ)
  ClusterJDatastoreException.Classification, 548
  ClusterJHelper, 556
  ColumnType, 565
  DynamicObject, 579
  LockMode, 580
  NullValue, 604
  PersistenceModifier, 606

Query.Ordering, 586
Classification (NdbError data type), 149
clear() (method of Batch), 623
clearError() (method of NdbEventOperation), 152
clone() (method of NdbRecAttr), 208
close (ClusterJ), 589, 596
close() (method of NdbScanOperation), 225
close() (method of NdbTransaction), 234
close() (method of Session), 629
close() (method of SessionFactory), 630
closeTransaction() (method of Ndb), 106
ClusterJ
defined, 529
ClusterJDatastoreException (ClusterJ), 546
ClusterJDatastoreException.Classification (ClusterJ), 548
ClusterJException (ClusterJ), 550
ClusterJFatalException (ClusterJ), 551
ClusterJFatalInternalException (ClusterJ), 553
ClusterJFatalUserException (ClusterJ), 554
ClusterJHelper (ClusterJ), 556
ClusterJPA
defined, 529
ClusterJUserException (ClusterJ), 561
cmp() (method of NdbScanFilter), 217
coding examples
MGM API, 523
NDB API, 359
Column
and NDB Cluster replication, 26
NDB API class, 24
Column (ClusterJ), 598
column (ClusterJ), 606, 608, 609
Column::ArrayType, 27
Column::equal(), 30
Column::getArrayType(), 30
Column::getCharset(), 30
Column::getColumnNo(), 31
Column::getDefaultValue(), 31
Column::getInlineSize(), 32
Column::getLength(), 32
Column::getName(), 32
Column::getNullable(), 33
Column::getPartitionKey(), 33
Column::getPartSize(), 33
Column::getPrecision(), 34
Column::getPrimaryKey(), 34
Column::getSizeInBytesForRecord(), 35
Column::getStorageType(), 36
Column::getStripeSize(), 36
Column::getType(), 36
Column::setArrayType(), 37
Column::setCharset(), 37
Column::setDefaultValue(), 37
Column::setLength(), 38
Column::setName(), 38
Column::setNullable(), 39
Column::setPartitionKey(), 39
Column::setPartSize(), 39
Column::setPrecision(), 40
Column::setPrimaryKey(), 40
Column::setScale(), 41
Column::setStorageType(), 41
Column::setStripeSize(), 41
Column::setType(), 42
Column::StorageType, 28
Column::Type, 28
ColumnMetadata (ClusterJ), 562
ColumnMetadata class (Connector for JavaScript), 631
Columns (ClusterJ), 599
columns (ClusterJ), 602, 606, 609
columnType (ClusterJ), 563
ColumnType (ClusterJ), 565
Commit
defined, 5
commit (ClusterJ), 598
commit() (method of Transaction), 632
commitStatus() (method of NdbTransaction), 235
CommitStatusType (NdbTransaction data type), 235
computeHash() (method of Ndb), 106
concurrency control, 13
connect() (method of Mynode), 628
connect() (method of Ndb_cluster_connection), 124
connecting to multiple clusters
example, 364, 525
ConnectionProperties() object (Node.js), 626
Connector for JavaScript, 621
Batch class, 623
ColumnMetadata class, 631
concepts, 621
Context class, 623
Converter class, 625
CPU architecture, 622
Errors, 626
examples, 633
FieldMapping class, 630
installing, 621
Mynode, 626
prerequisites, 621
Session class, 629
SessionFactory class, 630
TableMapping class, 630
TableMetadata class, 631
test program, 622
Transaction class, 632
Connector/J
known issues, 619
Constants (ClusterJ), 567
containers table (ndbmemcache), 647
Context class (Connector for JavaScript), 623
Context.find(), 623
Context.isBatch(), 625
Context.load(), 623
Context.persist(), 624
Context.remove(), 624
Context.save(), 625

Context.update(), 624
Converter class (Connector for JavaScript), 625
copy() (method of NdbInterpretedCode), 185
CPU architecture
  Connector for JavaScript, 622
createBatch() (method of Session), 629
createDatafile() (method of Dictionary), 52
createEvent() (method of Dictionary), 52
createForeignKey() (method of Dictionary), 52
createHashMap() (method of Dictionary), 52
createIndex() (method of Dictionary), 53
createLogfileGroup() (method of Dictionary), 53
createQuery (ClusterJ), 589
createQueryDefinition (ClusterJ), 613
createRecord() (method of Dictionary), 53
createTable() (method of Dictionary), 54
createTablespace() (method of Dictionary), 55
createUndofile() (method of Dictionary), 55
currentTransaction (ClusterJ), 589
currentTransaction() (method of Session), 629

D

data node
  defined, 3
Datafile
  NDB API class, 42
Datafile::getFileNo(), 44
Datafile::getFree(), 44
Datafile::getNode(), 44
Datafile::getObjectId(), 45
Datafile::getObjectStatus(), 45
Datafile::getObjectVersion(), 45
Datafile::getPath(), 45
Datafile::getSize(), 46
Datafile::getTablespace(), 46
Datafile::getTablespaceId(), 46
Datafile::setNode(), 47
Datafile::setPath(), 47
Datafile::setSize(), 47
Datafile::setTablespace(), 47
Dbug (ClusterJ), 575
debug (ClusterJ), 577, 577
defaultValue (ClusterJ), 599
DEFAULT_PROPERTY_CLUSTER_BYTE_BUFFER_POOL_SIZES (ClusterJ), 570
DEFAULT_PROPERTY_CLUSTER_CONNECT_AUTO_INCREMENT_BATCH_SIZE (ClusterJ), 570
DEFAULT_PROPERTY_CLUSTER_CONNECT_AUTO_INCREMENT_START (ClusterJ), 571
DEFAULT_PROPERTY_CLUSTER_CONNECT_AUTO_INCREMENT_STEP (ClusterJ), 571
DEFAULT_PROPERTY_CLUSTER_CONNECT_DELAY (ClusterJ), 571
DEFAULT_PROPERTY_CLUSTER_CONNECT_RETRIES (ClusterJ), 571
DEFAULT_PROPERTY_CLUSTER_CONNECT_TIMEOUT_AFTER (ClusterJ), 571
DEFAULT_PROPERTY_CLUSTER_CONNECT_TIMEOUT_BE (ClusterJ), 571
DEFAULT_PROPERTY_CLUSTER_CONNECT_TIMEOUT_MG (ClusterJ), 571
DEFAULT_PROPERTY_CLUSTER_CONNECT_VERBOSE (ClusterJ), 571
DEFAULT_PROPERTY_CLUSTER_DATABASE (ClusterJ), 571
DEFAULT_PROPERTY_CLUSTER_MAX_TRANSACTIONS (ClusterJ), 572
DEFAULT_PROPERTY_CONNECTION_POOL_SIZE (ClusterJ), 572
def_label() (method of NdbInterpretedCode), 186
def_sub() (method of NdbInterpretedCode), 186
deleteCurrentTuple() (method of NdbScanOperation), 225
deletePersistent (ClusterJ), 590, 590
deletePersistentAll (ClusterJ), 583, 590, 590
deleteTuple() (method of NdbIndexOperation), 161
deleteTuple() (method of NdbOperation), 195
deleteTuple() (method of NdbTransaction), 235
Dictionary
  NDB API class, 48
Dictionary::beginSchemaTrans(), 51
Dictionary::create*() methods
  and NDB Cluster replication, 49
Dictionary::createDatafile(), 52
Dictionary::createEvent(), 52
Dictionary::createForeignKey(), 52
Dictionary::createHashMap(), 52
Dictionary::createIndex(), 53
Dictionary::createLogfileGroup(), 53
Dictionary::createRecord(), 53
Dictionary::createTable(), 54
Dictionary::createTablespace(), 55
Dictionary::createUndofile(), 55
Dictionary::dropDatafile(), 55
Dictionary::dropEvent(), 55
Dictionary::dropForeignKey(), 56
Dictionary::dropIndex(), 56
Dictionary::dropLogfileGroup(), 56
Dictionary::dropTable(), 57
Dictionary::dropTablespace(), 57
Dictionary::dropUndofile(), 57
Dictionary::endSchemaTrans(), 57
Dictionary::getDatafile(), 58
Dictionary::getDefaultHashMap(), 58
Dictionary::getEvent(), 59
Dictionary::getForeignKey(), 59
Dictionary::getHashMap(), 59
Dictionary::getIndex(), 60
Dictionary::getLogfileGroup(), 60
Dictionary::getNdbError(), 61
Dictionary::getTable(), 61
Dictionary::getTablespace(), 61
Dictionary::getUndofile(), 62
Dictionary::hasSchemaTrans(), 62
Dictionary::initDefaultHashMap(), 62

Dictionary::invalidateIndex(), 63
Dictionary::invalidateTable(), 63
Dictionary::List
  NDB API class, 102
Dictionary::List::Element
  NDB API structure, 66
Dictionary::listIndexes(), 64
Dictionary::listObjects(), 64
Dictionary::prepareHashMap(), 65
Dictionary::releaseRecord(), 65
Dictionary::removeCachedIndex(), 66
Dictionary::removeCachedTable(), 65
Dictionary::SchemaTransFlag, 58
distribution awareness, 119
double_value() (method of NdbRecAttr), 209
dropDatafile() (method of Dictionary), 55
dropEvent() (method of Dictionary), 55
dropEventOperation() (method of Ndb), 107
dropForeignKey() (method of Dictionary), 56
dropIndex() (method of Dictionary), 56
dropLogfileGroup() (method of Dictionary), 56
dropTable() (method of Dictionary), 57
dropTablespace() (method of Dictionary), 57
dropUndofile() (method of Dictionary), 57
DynamicObject (ClusterJ), 578
DynamicObjectDelegate (ClusterJ), 579

E

Element
  NDB API structure, 66
Elements (ClusterJ)
  allowsNull, 599
  column, 606, 608, 609
  columns, 602, 606, 609
  defaultValue, 599
  extensions, 608
  key, 601
  name, 599, 602, 609
  nullValue, 608
  primaryKey, 608
  unique, 602
  value, 600, 601, 601, 603
  vendorName, 601
end() (method of NdbScanFilter), 219
endSchemaTrans() (method of Dictionary), 58
end_of_bound() (method of NdbIndexScanOperation), 164
ENV_CLUSTERJ_LOGGER_FACTORY_NAME (ClusterJ), 572
eq() (method of NdbScanFilter), 219
equal (ClusterJ), 611
equal() (method of Column), 30
equal() (method of HashMap), 89
equal() (method of NdbOperation), 195
equal() (method of Table), 268
error classification (defined), 146
error classifications, 359
error code (defined), 146
Error code types, 300
Error codes, 300
error detail message (defined), 147
error handling
  example, 369
  overview, 10
error message (defined), 147
Error status, 146
error types
  in applications, 369
errors
  classifying, 359
  MGM API, 521
  NDB API, 296
Errors (Connector for JavaScript), 626
ER_DDL, 76
Event
  NDB API class, 66
event reporting
  DDL, 76
event subscriptions
  lifetime, 107
Event::addEventColumn(), 70
Event::addEventColumns(), 70
Event::addTableEvent(), 71
Event::EventDurability, 71
Event::EventReport, 72
Event::getDurability(), 72
Event::getEventColumn(), 73
Event::getName(), 73
Event::getNoOfEventColumns(), 73
Event::getObjectId(), 74
Event::getObjectStatus(), 73
Event::getObjectVersion(), 73
Event::getReport(), 74
Event::getTable(), 74
Event::getTableEvent(), 74
Event::getTableName(), 75
Event::mergeEvents(), 75
Event::setDurability(), 75
Event::setName(), 76
Event::setReport(), 76
Event::setTable(), 76
Event::TableEvent, 77
EventBufferMemoryUsage
  NDB API structure, 77
EventDurability (Event data type), 71
EventReport (Event data type), 72
events
  example, 525
  handling
    example, 440
examples
  Connector for JavaScript, 633
Exceptions (ClusterJ)
  ClusterJDatastoreException, 546
  ClusterJException, 550
  ClusterJFatalException, 551

ClusterJFatalInternalException, 553
ClusterJFatalUserException, 554
ClusterJUserException, 561
ExecType (NdbTransaction data type), 236
execute (ClusterJ), 583, 584, 584
execute() (method of Batch), 623
execute() (method of NdbEventOperation), 153
execute() (method of NdbTransaction), 236
executePendingBlobOps() (method of NdbTransaction), 237
explain (ClusterJ), 584
Extension (ClusterJ), 600
Extensions (ClusterJ), 601
extensions (ClusterJ), 608

F

FieldMapping class (Connector for JavaScript), 630
Fields (ClusterJ)
DEFAULT_PROPERTY_CLUSTER_BYTE_BUFFER_POOL_SIZES, 570
DEFAULT_PROPERTY_CLUSTER_CONNECT_AUTO_INCREMENT_BATCH_SIZE, 570
DEFAULT_PROPERTY_CLUSTER_CONNECT_AUTO_INCREMENT_START, 571
DEFAULT_PROPERTY_CLUSTER_CONNECT_AUTO_INCREMENT_STEP, 571
DEFAULT_PROPERTY_CLUSTER_CONNECT_DELAY, 571
DEFAULT_PROPERTY_CLUSTER_CONNECT_RETRIES, 571
DEFAULT_PROPERTY_CLUSTER_CONNECT_TIMEOUT_AFTER, 571
DEFAULT_PROPERTY_CLUSTER_CONNECT_TIMEOUT_BEFORE, 571
DEFAULT_PROPERTY_CLUSTER_CONNECT_TIMEOUT_MGM, 571
DEFAULT_PROPERTY_CLUSTER_CONNECT_VERBOSE, 571
DEFAULT_PROPERTY_CLUSTER_DATABASE, 571
DEFAULT_PROPERTY_CLUSTER_MAX_TRANSACTIONS, 572
DEFAULT_PROPERTY_CONNECTION_POOL_SIZE, 572
ENV_CLUSTERJ_LOGGER_FACTORY_NAME, 572
INDEX_USED, 582
PROPERTY_CLUSTER_BYTE_BUFFER_POOL_SIZES, 572
PROPERTY_CLUSTER_CONNECTION_SERVICE, 573
PROPERTY_CLUSTER_CONNECTSTRING, 573
PROPERTY_CLUSTER_CONNECT_AUTO_INCREMENT_BATCH_SIZE, 572
PROPERTY_CLUSTER_CONNECT_AUTO_INCREMENT_START, 572
PROPERTY_CLUSTER_CONNECT_AUTO_INCREMENT_STEP, 572
PROPERTY_CLUSTER_CONNECT_DELAY, 572
PROPERTY_CLUSTER_CONNECT_RETRIES, 573
PROPERTY_CLUSTER_CONNECT_TIMEOUT_AFTER, 573
PROPERTY_CLUSTER_CONNECT_TIMEOUT_BEFORE, 573
PROPERTY_CLUSTER_CONNECT_TIMEOUT_MGM, 573
PROPERTY_CLUSTER_CONNECT_VERBOSE, 573
PROPERTY_CLUSTER_DATABASE, 573
PROPERTY_CLUSTER_MAX_TRANSACTIONS, 574
PROPERTY_CONNECTION_POOL_NODEIDS, 574
PROPERTY_CONNECTION_POOL_SIZE, 574
PROPERTY_DEFER_CHANGES, 574
PROPERTY_JDBC_DRIVER_NAME, 574
PROPERTY_JDBC_PASSWORD, 574
PROPERTY_JDBC_URL, 574
PROPERTY_JDBC_USERNAME, 575
SCAN_TYPE, 583
SCAN_TYPE_INDEX_SCAN, 583
SCAN_TYPE_PRIMARY_KEY, 583
SCAN_TYPE_TABLE_SCAN, 583
SCAN_TYPE_UNIQUE_KEY, 583
SESSION_FACTORY_SERVICE_CLASS_NAME, 575
SESSION_FACTORY_SERVICE_FILE_NAME, 575
finalise() (method of NdbInterpretedCode), 186
find (ClusterJ), 590
find() (method of Context), 623
float_value() (method of NdbRecAttr), 209
flush (ClusterJ), 577, 591
ForeignKey
NDB API class, 78
ForeignKey(), 79
ForeignKey() (ForeignKey constructor), 79
ForeignKey::getChildColumnCount(), 81
ForeignKey::getChildColumnNo(), 82
ForeignKey::getChildIndex(), 82
ForeignKey::getChildTable(), 81
ForeignKey::getName(), 80
ForeignKey::getObjectId(), 84
ForeignKey::getObjectStatus(), 84
ForeignKey::getObjectVersion(), 85
ForeignKey::getOnDeleteAction(), 83
ForeignKey::getOnUpdateAction(), 82
ForeignKey::getParentColumnCount(), 81
ForeignKey::getParentColumnNo(), 82
ForeignKey::getParentIndex(), 81
ForeignKey::getParentTable(), 80
ForeignKey::setChild(), 83
ForeignKey::setName(), 83
ForeignKey::setOnDeleteAction(), 84
ForeignKey::setOnUpdateAction(), 84
ForeignKey::setParent(), 83
found (ClusterJ), 591
fractional seconds

ndbmemcache, 654
fragment
defined, 3
FragmentType (Object data type), 250

G

GCP (Global Checkpoint)
defined, 2
ge() (method of NdbScanFilter), 220
get (ClusterJ), 577, 615
getArrayType() (method of Column), 30
getAutoGrowSpecification() (method of LogfileGroup), 99
getAutoGrowSpecification() (method of Tablespace), 288
getBlobEventName() (method of NdbBlob), 134
getBlobHandle() (method of NdbEventOperation), 153
getBlobHandle() (method of NdbOperation), 197
getBlobTableName() (method of NdbBlob), 135
getBooleanProperty (ClusterJ), 558
getCharset() (method of Column), 30
getChildColumnCount() (method of ForeignKey), 81
getChildColumnNo() (method of ForeignKey), 82
getChildIndex() (method of ForeignKey), 82
getChildTable() (method of ForeignKey), 81
getClassification (ClusterJ), 547
getCode (ClusterJ), 548
getColumn() (method of Index), 93
getColumn() (method of NdbBlob), 135
getColumn() (method of NdbRecAttr), 210
getColumn() (method of Table), 269
getColumnNo() (method of Column), 31
getConnectionPoolSessionCounts (ClusterJ), 596
getDatabaseName() (method of Ndb), 108
getDatabaseSchemaName() (method of Ndb), 108
getDatafile() (method of Dictionary), 58
getDefaultHashMap() (method of Dictionary), 58
getDefaultLogfileGroup() (method of Tablespace), 288
getDefaultLogfileGroupId() (method of Tablespace), 288
getDefaultNoPartitionsFlag() (method of Table), 268
getDefaultValue() (method of Column), 31
getDescending() (method of NdbIndexScanOperation), 164
getDictionary() (method of Ndb), 108
getDurability() (method of Event), 72
getEmptyBitmask() (method of NdbDictionary), 142
getEpoch() (method of NdbEventOperation), 153
getEvent() (method of Dictionary), 59
getEventColumn() (method of Event), 73
getEventType() (method of NdbEventOperation), 154
getEventType2() (method of NdbEventOperation), 154
getExtentSize() (method of Tablespace), 288
getFileNo() (method of Datafile), 44
getFileNo() (method of Undofile), 293
getFirstAttrId() (method of NdbDictionary), 143
getForeignKey() (method of Dictionary), 59
getFragmentCount() (method of Table), 268
getFragmentData() (method of Table), 268
getFragmentDataLen() (method of Table), 269
getFragmentNodes() (method of Table), 270
getFragmentType() (method of Table), 270
getFree() (method of Datafile), 44
getFrmData() (method of Table), 269
getFrmLength() (method of Table), 269
getGCI() (method of NdbEventOperation), 154
getGCI() (method of NdbTransaction), 238
getGCIEventOperations() (method of Ndb), 109
getHashMap() (method of Dictionary), 60
getHashMap() (method of Table), 271
getHighestQueuedEpoch() (method of Ndb), 110
getIndex() (method of Dictionary), 60
getIndex() (method of NdbIndexOperation), 161
getInlineSize() (method of Column), 32
getKValue() (method of Table), 271
getLatestGCI() (method of Ndb), 110
getLatestGCI() (method of NdbEventOperation), 154
getLength() (method of Column), 32
getLength() (method of NdbBlob), 136
getLinearFlag() (method of Table), 271
getLockHandle() (method of NdbOperation), 197
getLockMode() (method of NdbOperation), 198
getLogfileGroup() (method of Dictionary), 60
getLogfileGroup() (method of Undofile), 293
getLogfileGroupId() (method of Undofile), 293
getLogging() (method of Index), 93
getLogging() (method of Table), 271
getMapLen() (method of HashMap), 89
getMapping() (method of Session), 629
getMapValues() (method of HashMap), 89
getMaxLoadFactor() (method of Table), 272
getMaxPendingBlobReadBytes() (method of NdbTransaction), 238
getMaxPendingBlobWriteBytes() (method of NdbTransaction), 238
getMaxRows() (method of Table), 272
getMinLoadFactor() (method of Table), 272
getMysqlCode (ClusterJ), 548
getName() (method of Column), 32
getName() (method of Event), 73
getName() (method of ForeignKey), 80
getName() (method of HashMap), 88
getName() (method of Index), 94
getName() (method of LogfileGroup), 99
getName() (method of Tablespace), 289
getNdbError() (method of Dictionary), 61
getNdbError() (method of Ndb), 111
getNdbError() (method of NdbBlob), 136
getNdbError() (method of NdbEventOperation), 155
getNdbError() (method of NdbInterpretedCode), 187
getNdbError() (method of NdbOperation), 198
getNdbError() (method of NdbScanFilter), 221
getNdbError() (method of NdbTransaction), 239
getNdbErrorDetail() (method of Ndb), 111
getNdbErrorLine() (method of NdbOperation), 199
getNdbErrorLine() (method of NdbTransaction), 239

getNdbErrorOperation() (method of NdbTransaction), 239
getNdbIndexOperation() (method of NdbTransaction), 240
getNdbIndexScanOperation() (method of NdbTransaction), 240
getNdbObjectName() (method of Ndb), 112
getNdbOperation() (method of NdbBlob), 136
getNdbOperation() (method of NdbScanFilter), 221
getNdbOperation() (method of NdbTransaction), 240
getNdbScanOperation() (method of NdbTransaction), 241
getNdbTransaction() (method of NdbOperation), 199
getNdbTransaction() (method of NdbScanOperation), 226
getNextAttrId() (method of NdbDictionary), 143
getNextCompletedOperation() (method of NdbTransaction), 241
getNode() (method of Datafile), 44
getNode() (method of Undofile), 294
getNoOfColumns() (method of Index), 94
getNoOfCOlumns() (method of Table), 273
getNoOfEventColumns() (method of Event), 73
getNoOfPrimaryKeys() (method of Table), 273
getNull() (method of NdbBlob), 136
getNullable() (method of Column), 33
getNullBitOffset() (method of NdbDictionary), 143
getObjectId() (method of Datafile), 45
getObjectId() (method of Event), 74
getObjectId() (method of ForeignKey), 84
getObjectId() (method of HashMap), 90
getObjectId() (method of Index), 95
getObjectId() (method of LogfileGroup), 100
getObjectId() (method of Object), 252
getObjectId() (method of Table), 273
getObjectId() (method of Tablespace), 289
getObjectId() (method of Undofile), 294
getObjectStatus() (method of Datafile), 45
getObjectStatus() (method of Event), 73
getObjectStatus() (method of ForeignKey), 84
getObjectStatus() (method of HashMap), 89
getObjectStatus() (method of Index), 94
getObjectStatus() (method of LogfileGroup), 100
getObjectStatus() (method of Object), 253
getObjectStatus() (method of Table), 273
getObjectStatus() (method of Tablespace), 289
getObjectStatus() (method of Undofile), 294
getObjectType() (method of Table), 274
getObjectVersion() (method of Datafile), 45
getObjectVersion() (method of Event), 73
getObjectVersion() (method of ForeignKey), 85
getObjectVersion() (method of HashMap), 90
getObjectVersion() (method of Index), 94
getObjectVersion() (method of LogfileGroup), 100
getObjectVersion() (method of Object), 253
getObjectVersion() (method of Table), 274
getObjectVersion() (method of Tablespace), 289
getObjectVersion() (method of Undofile), 294
getOffset() (method of NdbDictionary), 144
getOnDeleteAction() (method of ForeignKey), 83
getOnUpdateAction() (method of ForeignKey), 82
getOpenSessionFactories() (method of Mynode), 628
getOpenSessions() (method of SessionFactory), 630
getParentColumnCount() (method of ForeignKey), 81
getParentColumnNo() (method of ForeignKey), 82
getParentIndex() (method of ForeignKey), 81
getParentTable() (method of ForeignKey), 80
getPartitionBalance() (method of Table), 274
getPartitionBalanceString() (method of Table), 274
getPartitionId() (method of Table), 275
getPartitionKey() (method of Column), 33
getPartSize() (method of Column), 34
getPath() (method of Datafile), 45
getPath() (method of Undofile), 295
getPos() (method of NdbBlob), 137
getPreBlobHandle() (method of NdbEventOperation), 155
getPrecision() (method of Column), 34
getPreValue() (method of NdbEventOperation), 155
getPrimaryKey() (method of Column), 34
getPrimaryKey() (method of Table), 275
getPruned() (method of NdbScanOperation), 226
getQueryBuilder (ClusterJ), 591
getRangeListData() (method of Table), 275
getRangeListDataLen() (method of Table), 275
getRecordIndexName() (method of NdbDictionary), 144
getRecordRowLength() (method of NdbDictionary), 144
getRecordTableName() (method of NdbDictionary), 144
getRecordType() (method of NdbDictionary), 145
getReport() (method of Event), 74
getResultList (ClusterJ), 584
getRollbackOnly (ClusterJ), 598
getRollbackOnly() (method of Transaction), 633
getRowChecksumIndicator() (method of Table), 276
getRowGCIIndicator() (method of Table), 276
getServiceInstance (ClusterJ), 558, 559, 559, 559
getServiceInstances (ClusterJ), 559
getSession (ClusterJ), 596, 596
getSession() (method of Batch), 623
getSessionFactory (ClusterJ), 560, 560, 597
getSize() (method of Datafile), 46
getSize() (method of Undofile), 295
getSizeInBytesForRecord() (method of Column), 35
getSorted() (method of NdbIndexScanOperation), 165
getState() (method of NdbBlob), 137
getState() (method of NdbEventOperation), 156
getStatus (ClusterJ), 548
getStorageType() (method of Column), 36
getStringProperty (ClusterJ), 560
getStripeSize() (method of Column), 36
getTable() (method of Dictionary), 61
getTable() (method of Event), 74
getTable() (method of Index), 95
getTable() (method of NdbInterpretedCode), 187
getTable() (method of NdbOperation), 199
getTableEvent() (method of Event), 74

getTableId() (method of Table), 276
getTableMetadata() (method of Session), 629
getTableName() (method of Event), 75
getTableName() (method of NdbOperation), 199
getTablespace() (method of Datafile), 46
getTablespace() (method of Dictionary), 61
getTablespace() (method of Table), 276
getTablespaceData() (method of Table), 277
getTablespaceDataLen() (method of Table), 277
getTablespaceId() (method of Datafile), 46
getTablespaceNames() (method of Table), 278
getTablespaceNamesLen() (method of Table), 278
getTransactionId() (method of NdbTransaction), 241
getType (ClusterJ), 615
getType() (method of Column), 36
getType() (method of Index), 95
getType() (method of NdbOperation), 200
getType() (method of NdbRecAttr), 210
getUndoBufferSize() (method of LogfileGroup), 101
getUndofile() (method of Dictionary), 62
getUndoFreeWords() (method of LogfileGroup), 101
getValue() (method of NdbBlob), 137
getValue() (method of NdbEventOperation), 156
getValue() (method of NdbOperation), 200
getValuePtr() (method of NdbDictionary), 145
GetValueSpec
  NDB API structure, 85
getVersion() (method of NdbBlob), 138
getWordsUsed() (method of NdbInterpretedCode), 187
get_auto_reconnect() (method of Ndb_cluster_connection), 124
get_eventbuffer_free_percent() (method of Ndb), 109
get_eventbuf_max_alloc()() (method of Ndb), 109
get_event_buffer_memory_usage() (method of Ndb), 110
get_latest_error() (method of Ndb_cluster_connection), 124
get_latest_error_msg() (method of Ndb_cluster_connection), 125
get_max_adaptive_send_time() (method of Ndb_cluster_connection), 125
get_next_ndb_object() (method of Ndb_cluster_connection), 125
get_num_recv_threads() (method of Ndb_cluster_connection), 126
get_range_no() (method of NdbIndexScanOperation), 164
get_size_in_bytes() (method of NdbRecAttr), 209
grant tables
  and NDB API applications, 2
greaterEqual (ClusterJ), 612
greaterThan (ClusterJ), 612
Group (NdbScanFilter data type), 221
gt() (method of NdbScanFilter), 222

H

hasDefaultValues() (method of Table), 278
hasError() (method of NdbEventOperation), 157
HashMap
  NDB API class, 86
HashMap constructor, 87
HashMap::equal(), 89
HashMap::getMapLen(), 88
HashMap::getMapValues(), 89
HashMap::getName(), 88
HashMap::getObjectId(), 90
HashMap::getObjectStatus(), 89
HashMap::getObjectVersion(), 90
HashMap::setMap(), 88
HashMap::setName(), 88
hasSchemaTrans() (method of Dictionary), 62

I

in (ClusterJ), 612
Index
  NDB API class, 90
Index (ClusterJ), 602
Index::addColumn(), 92
Index::addColumnName(), 92
Index::addColumnNames(), 92
Index::getColumn(), 93
Index::getLogging(), 93
Index::getName(), 94
Index::getNoOfColumns(), 94
Index::getObjectId(), 95
Index::getObjectStatus(), 94
Index::getObjectVersion(), 94
Index::getTable(), 95
Index::getType(), 95
Index::setName(), 96
Index::setTable(), 96
Index::setType(), 96
Index::Type, 96
IndexBound
  NDB API structure, 97
INDEX_USED (ClusterJ), 582
Indices (ClusterJ), 602
init() (method of Ndb), 113
initDefaultHashMap() (method of Dictionary), 62
initial node restart
  defined, 3
insertTuple() (method of NdbOperation), 201
insertTuple() (method of NdbTransaction), 242
int32_value() (method of NdbRecAttr), 210
int64_value() (method of NdbRecAttr), 211
int8_value() (method of NdbRecAttr), 210
integer comparison methods (of NdbScanFilter class), 215
Interfaces (ClusterJ)
  ColumnMetadata, 562
  Constants, 567
  Dbug, 575
  DynamicObjectDelegate, 579
  Predicate, 609
  PredicateOperand, 610
  Query, 581

QueryBuilder, 613
QueryDefinition, 614
QueryDomainType, 615
Results, 586
Session, 587
SessionFactory, 595
SessionFactoryService, 596
Transaction, 597
interpret_exit_last_row() (method of NdbInterpretedCode), 187
interpret_exit_nok() (method of NdbInterpretedCode), 188
interpret_exit_ok() (method of NdbInterpretedCode), 188
invalidateIndex() (method of Dictionary), 63
invalidateTable() (method of Dictionary), 63
isActive (ClusterJ), 598
isActive() (method of Transaction), 633
isBatch() (method of Context), 625
isClosed (ClusterJ), 591
isClosed() (method of Session), 629
isConsistent() (method of Ndb), 113
isConsistent() (method of NdbEventOperation), 157
isConsistentGCI() (method of Ndb), 113
isEmptyEpoch() (method of NdbEventOperation), 158
isErrorEpoch() (method of NdbEventOperation), 158
isExpectingHigherQueuedEpochs() (method of Ndb), 114
isfalse() (method of NdbScanFilter), 219
isNotNull (ClusterJ), 612
isnotnull() (method of NdbScanFilter), 220
isNull (ClusterJ), 612
isNull() (method of NdbDictionary), 145
isNULL() (method of NdbRecAttr), 211
isnull() (method of NdbScanFilter), 220
isOverrun() (method of NdbEventOperation), 158
isPartitionKey (ClusterJ), 563
isPrimaryKey (ClusterJ), 564
istrue() (method of NdbScanFilter), 220
iteration
Ndb objects, 125
iterator (ClusterJ), 587

J

Java, 529
JavaScript, 621
API documentation, 622
javaType (ClusterJ), 564
JDBC
known issues, 619

K

key (ClusterJ), 601
Key_part_ptr
NDB API structure, 103

L

LCP (Local Checkpoint)
defined, 2
le() (method of NdbScanFilter), 222
lessEqual (ClusterJ), 612
lessThan (ClusterJ), 613
like (ClusterJ), 613
List
NDB API class, 102
listBatches() (method of Session), 629
listIndexes() (method of Dictionary), 64
listObjects() (method of Dictionary), 64
listTables() (method of Session), 629
load (ClusterJ), 591
load() (method of Context), 623
load_const_null() (method of NdbInterpretedCode), 188
load_const_u16() (method of NdbInterpretedCode), 188
load_const_u32() (method of NdbInterpretedCode), 189
load_const_u64() (method of NdbInterpretedCode), 189
Lob (ClusterJ), 603
lock handles
NDB API, 197, 247
lock handling
and scan operations, 10
lockCurrentTuple() (method of NdbScanOperation), 227
locking and transactions
NDB API, 197
LockMode (ClusterJ), 580
LockMode (NdbOperation data type), 202
lock_ndb_objects() (method of ndb_cluster_connection), 126
LogfileGroup
NDB API class, 97
LogfileGroup::getAutoGrowSpecification(), 99
LogfileGroup::getName(), 99
LogfileGroup::getObjectId(), 100
LogfileGroup::getObjectStatus(), 100
LogfileGroup::getObjectVersion(), 100
LogfileGroup::getUndoBufferSize(), 101
LogfileGroup::getUndoFreeWords(), 101
LogfileGroup::setAutoGrowSpecification(), 101
LogfileGroup::setName(), 101
LogfileGroup::setUndoBufferSize(), 102
lookup (ClusterJ), 550
lt() (method of NdbScanFilter), 222

M

makePersistent (ClusterJ), 592
makePersistentAll (ClusterJ), 592
management (MGM) node
defined, 3
mapField() (method of TableMapping), 630
markModified (ClusterJ), 592

maximumLength (ClusterJ), 564
medium_value() (method of NdbRecAttr), 211
memcache commands
  ndbmemcache, 650
mergeEvents() (method of Event), 75
mergeEvents() (method of NdbEventOperation), 158
Methods (ClusterJ)
  and, 610
  append, 577
  begin, 597
  between, 611
  charsetName, 563
  close, 589, 596
  columnType, 563
  commit, 598
  createQuery, 589
  createQueryDefinition, 613
  currentTransaction, 589
  debug, 577, 577
  deletePersistent, 590, 590
  deletePersistentAll, 583, 590, 590
  equal, 611
  execute, 583, 584, 584
  explain, 584
  find, 590
  flush, 577, 591
  found, 591
  get, 577, 615
  getBooleanProperty, 558
  getClassification, 547
  getCode, 548
  getConnectionPoolSessionCounts, 596
  getMysqlCode, 548
  getQueryBuilder, 591
  getResultList, 584
  getRollbackOnly, 598
  getServiceInstance, 558, 559, 559, 559
  getServiceInstances, 559
  getSession, 596, 596
  getSessionFactory, 560, 560, 597
  getStatus, 548
  getStringProperty, 560
  getType, 615
  greaterEqual, 612
  greaterThan, 612
  in, 612
  isActive, 598
  isClosed, 591
  isNotNull, 612
  isNull, 612
  isPartitionKey, 563
  isPrimaryKey, 564
  iterator, 587
  javaType, 564
  lessEqual, 612
  lessThan, 613
  like, 613
  load, 591
  lookup, 550
  makePersistent, 592
  makePersistentAll, 592
  markModified, 592
  maximumLength, 564
  name, 564
  newDbug, 561
  newInstance, 592, 593
  not, 610, 614
  nullable, 564
  number, 564
  or, 610
  output, 577
  param, 614
  persist, 593
  pop, 578
  precision, 565
  print, 578
  push, 578, 578
  release, 593
  remove, 593
  rollback, 598
  savePersistent, 593
  savePersistentAll, 594
  scale, 565
  set, 578, 578
  setLimits, 585
  setLockMode, 594
  setOrdering, 585
  setParameter, 585
  setPartitionKey, 594
  setRollbackOnly, 598
  trace, 578
  unloadSchema, 594
  updatePersistent, 595
  updatePersistentAll, 595
  where, 614
MGM API
  coding examples, 523
  errors, 521
multiple clusters, 121
multiple clusters, connecting to
  example, 364, 525
Mynode (Connector for JavaScript), 626
Mynode.connect(), 628
Mynode.ConnectionProperties(), 626
Mynode.getOpenSessionFactories(), 628
Mynode.openSession(), 628
MySQL Cluster Connector for Java, 529
  and foreign keys, 619
  and joins, 618
  and TIMESTAMP, 619
  and views, 618
  known issues, 618, 618
MySQL privileges
  and NDB API applications, 2

N

name (ClusterJ), 564, 599, 602, 609
NDB
  defined, 3
  record structure, 12
Ndb
  NDB API class, 103
NDB API
  and MySQL privileges, 2
  coding examples, 359
  defined, 4
  errors, 296
  lock handles, 197, 247
NDB API classes
  overview, 4
NDB Cluster
  benchmarks, 6
  configuration (ndbmemcache), 644
  memcache, 643
  ndbmemcache, 643
  Node.js, 621
  performance, 6
NDB Cluster replication
  and Column, 25
  and Dictionary::create*() methods, 49
Ndb::closeTransaction(), 106
Ndb::computeHash(), 106
Ndb::dropEventOperation(), 107
Ndb::EventBufferMemoryUsage
  NDB API structure, 77
Ndb::getDatabaseName(), 108
Ndb::getDatabaseSchemaName(), 108
Ndb::getDictionary(), 108
Ndb::getGCIEventOperations(), 109
Ndb::getHighestQueuedEpoch(), 110
Ndb::getLatestGCI(), 110
Ndb::getNdbError(), 111
Ndb::getNdbErrorDetail(), 111
Ndb::getNdbObjectName(), 112
Ndb::get_eventbuffer_free_percent(), 109
Ndb::get_eventbuf_max_alloc(), 109
Ndb::get_event_buffer_memory_usage(), 110
Ndb::init(), 113
Ndb::isConsistent(), 113
Ndb::isConsistentGCI(), 113
Ndb::isExpectingHigherQueuedEpochs(), 114
Ndb::Key_part_ptr
  NDB API structure, 103
Ndb::nextEvent(), 114
Ndb::nextEvent2(), 115
Ndb::PartitionSpec
  NDB API structure, 256
Ndb::pollEvents(), 115
Ndb::pollEvents2(), 116
Ndb::setDatabaseName(), 117
Ndb::setDatabaseSchemaName(), 117
Ndb::setNdbObjectName(), 118
Ndb::set_eventbuffer_free_percent(), 118
Ndb::set_eventbuf_max_alloc(), 118
Ndb::startTransaction(), 119
NdbBlob
  NDB API class, 131
NdbBlob::ActiveHook type, 133
NdbBlob::blobsFirstBlob(), 134
NdbBlob::blobsNextBlob(), 134
NdbBlob::getBlobEventName(), 134
NdbBlob::getBlobTableName(), 135
NdbBlob::getColumn(), 135
NdbBlob::getLength(), 135
NdbBlob::getNdbError(), 136
NdbBlob::getNdbOperation(), 136
NdbBlob::getNull(), 136
NdbBlob::getPos(), 137
NdbBlob::getState(), 137
NdbBlob::getValue(), 137
NdbBlob::getVersion(), 138
NdbBlob::readData(), 138
NdbBlob::setActiveHook(), 138
NdbBlob::setNull(), 139
NdbBlob::setPos(), 139
NdbBlob::setValue(), 139
NdbBlob::State type, 140
NdbBlob::truncate(), 140
NdbBlob::writeData(), 140
NdbDictionary
  NDB API class, 141
NdbDictionary::AutoGrowSpecification
  NDB API structure, 23
NdbDictionary::Column
  NDB API class, 24
NdbDictionary::Dictionary
  NDB API class, 48
NdbDictionary::Event
  NDB API class, 67
NdbDictionary::getEmptyBitmask(), 142
NdbDictionary::getFirstAttrId(), 143
NdbDictionary::getNextAttrId(), 143
NdbDictionary::getNullBitOffset(), 143
NdbDictionary::getOffset(), 144
NdbDictionary::getRecordIndexName(), 144
NdbDictionary::getRecordRowLength(), 144
NdbDictionary::getRecordTableName(), 144
NdbDictionary::getRecordType(), 145
NdbDictionary::getValuePtr(), 145
NdbDictionary::Index
  NDB API class, 90
NdbDictionary::isNull(), 145
NdbDictionary::LogfileGroup
  NDB API class, 97
NdbDictionary::Object
  NDB API class, 249
NdbDictionary::RecordSpecification
  NDB API structure, 257
NdbDictionary::setNull(), 146
NdbDictionary::Table

NDB API class, 262
NdbDictionary::Tablespace
NDB API class, 286
NdbDictionary::Undofile
NDB API class, 291
NdbError
NDB API structure, 146
NdbError::Classification type, 148
NdbError::Status type, 149
NdbEventOperation
NDB API class, 150
NdbEventOperation::clearError(), 152
NdbEventOperation::execute(), 153
NdbEventOperation::getBlobHandle(), 153
NdbEventOperation::getEpoch(), 153
NdbEventOperation::getEventType(), 154
NdbEventOperation::getEventType2(), 154
NdbEventOperation::getGCI(), 154
NdbEventOperation::getLatestGCI(), 154
NdbEventOperation::getNdbError(), 155
NdbEventOperation::getPreBlobHandle(), 155
NdbEventOperation::getPreValue(), 155
NdbEventOperation::getState(), 156
NdbEventOperation::getValue(), 156
NdbEventOperation::hasError(), 157
NdbEventOperation::isConsistent(), 157
NdbEventOperation::isEmptyEpoch(), 158
NdbEventOperation::isErrorEpoch(), 158
NdbEventOperation::isOverrun(), 158
NdbEventOperation::mergeEvents(), 158
NdbEventOperation::State, 159
NdbEventOperation::tableFragmentationChanged(), 159
NdbEventOperation::tableFrmChanged(), 159
NdbEventOperation::tableNameChanged(), 160
NdbIndexOperation
NDB API class, 160
NdbIndexOperation class
example, 7
NdbIndexOperation::deleteTuple(), 161
NdbIndexOperation::getIndex(), 161
NdbIndexOperation::readTuple(), 162
NdbIndexOperation::updateTuple(), 162
NdbIndexScanOperation
NDB API class, 162
NdbIndexScanOperation::BoundType, 163
NdbIndexScanOperation::end_of_bound(), 164
NdbIndexScanOperation::getDescending(), 164
NdbIndexScanOperation::getSorted(), 164
NdbIndexScanOperation::get_range_no(), 164
NdbIndexScanOperation::IndexBound
NDB API structure, 97
NdbIndexScanOperation::readTuples(), 165
NdbIndexScanOperation::reset_bounds(), 165
NdbInterpretedCode
NDB API class, 167
NdbInterpretedCode register-loading methods, 171
NdbInterpretedCode() (constructor), 175
NdbInterpretedCode::add_reg(), 176
NdbInterpretedCode::add_val(), 176
NdbInterpretedCode::branch_col_and_mask_eq_mask(), 177
NdbInterpretedCode::branch_col_and_mask_eq_zero(), 177
NdbInterpretedCode::branch_col_and_mask_ne_mask(), 177
NdbInterpretedCode::branch_col_and_mask_ne_zero(), 178
NdbInterpretedCode::branch_col_eq(), 178
NdbInterpretedCode::branch_col_eq_null(), 179
NdbInterpretedCode::branch_col_ge(), 179
NdbInterpretedCode::branch_col_gt(), 179
NdbInterpretedCode::branch_col_le(), 180
NdbInterpretedCode::branch_col_like(), 180
NdbInterpretedCode::branch_col_lt(), 181
NdbInterpretedCode::branch_col_ne(), 181
NdbInterpretedCode::branch_col_ne_null(), 182
NdbInterpretedCode::branch_col_notlike(), 182
NdbInterpretedCode::branch_eq(), 182
NdbInterpretedCode::branch_eq_null(), 183
NdbInterpretedCode::branch_ge(), 183
NdbInterpretedCode::branch_gt(), 183
NdbInterpretedCode::branch_label(), 184
NdbInterpretedCode::branch_le(), 184
NdbInterpretedCode::branch_lt(), 184
NdbInterpretedCode::branch_ne(), 185
NdbInterpretedCode::branch_ne_null(), 185
NdbInterpretedCode::call_sub(), 185
NdbInterpretedCode::copy(), 185
NdbInterpretedCode::def_label(), 186
NdbInterpretedCode::def_sub(), 186
NdbInterpretedCode::finalise(), 186
NdbInterpretedCode::getNdbError(), 187
NdbInterpretedCode::getTable(), 187
NdbInterpretedCode::getWordsUsed(), 187
NdbInterpretedCode::interpret_exit_last_row(), 187
NdbInterpretedCode::interpret_exit_nok(), 188
NdbInterpretedCode::interpret_exit_ok(), 188
NdbInterpretedCode::load_const_null(), 188
NdbInterpretedCode::load_const_u16(), 188
NdbInterpretedCode::load_const_u32(), 189
NdbInterpretedCode::load_const_u64(), 189
NdbInterpretedCode::read_attr(), 189
NdbInterpretedCode::ret_sub(), 190
NdbInterpretedCode::sub_reg(), 190
NdbInterpretedCode::sub_val(), 191
NdbInterpretedCode::write_attr(), 191
NdbLockHandle
defined, 198
using, 197
ndbmemcache, 643
AUTO_INCREMENT columns, 654
availability, 643
cache policies, 646
compiling, 643
configuration, 644, 645, 648, 649, 650

configuration tables, 646, 647
configuration versioning, 649
enabling support for, 643
fractional seconds, 654
known issues, 654
limitations, 654
log file, 652
memcache commands, 650
memcache protocol, 650
memcached options, 643
metadata, 646
online configuration, 650
performance, 649, 650
schema changes, 654
server_roles table, 646
starting, 645
upgrades, 649
NdbMgmHandle, 483, 490, 490, 491
NdbOperation
NDB API class, 192
NdbOperation class
example, 6
NdbOperation::AbortOption, 195
NdbOperation::deleteTuple(), 195
NdbOperation::equal(), 195
NdbOperation::getBlobHandle(), 197
NdbOperation::getLockHandle(), 197
NdbOperation::getLockMode(), 198
NdbOperation::getNdbError(), 198
NdbOperation::getNdbErrorLine(), 199
NdbOperation::getNdbTransaction(), 199
NdbOperation::getTable(), 199
NdbOperation::getTableName(), 199
NdbOperation::getType(), 200
NdbOperation::getValue(), 200
NdbOperation::GetValueSpec
NDB API structure, 85
NdbOperation::insertTuple(), 201
NdbOperation::LockMode, 202
NdbOperation::NdbIndexOperation
NDB API class, 160
NdbOperation::NdbScanOperation
NDB API class, 223
NdbOperation::OperationOptions
NDB API structure, 253
NdbOperation::readTuple(), 202
NdbOperation::setValue(), 202
NdbOperation::SetValueSpec
NDB API structure, 261
NdbOperation::Type, 204
NdbOperation::updateTuple(), 205
NdbOperation::writeTuple(), 205
NdbRecAttr
NDB API class, 205
NdbRecAttr class, 205
NdbRecAttr::aRef(), 208
NdbRecAttr::char_value(), 208
NdbRecAttr::clone(), 208
NdbRecAttr::double_value(), 209
NdbRecAttr::float_value(), 209
NdbRecAttr::getColumn(), 209
NdbRecAttr::getType(), 210
NdbRecAttr::get_size_in_bytes(), 209
NdbRecAttr::int32_value(), 210
NdbRecAttr::int64_value(), 211
NdbRecAttr::int8_value(), 210
NdbRecAttr::isNULL(), 211
NdbRecAttr::medium_value(), 211
NdbRecAttr::short_value(), 212
NdbRecAttr::u_32_value(), 212
NdbRecAttr::u_64_value(), 212
NdbRecAttr::u_8_value(), 212
NdbRecAttr::u_char_value(), 213
NdbRecAttr::u_medium_value(), 213
NdbRecAttr::u_short_value(), 213
NdbRecord
example, 390, 452
NDB API interface, 213
obtaining column size for, 35
NdbScanFilter
NDB API class, 214
NdbScanFilter class
integer comparison methods, 215
NdbScanFilter::begin(), 216
NdbScanFilter::BinaryCondition, 217
NdbScanFilter::cmp(), 217
NdbScanFilter::end(), 219
NdbScanFilter::eq(), 219
NdbScanFilter::ge(), 220
NdbScanFilter::getNdbError(), 221
NdbScanFilter::getNdbOperation(), 221
NdbScanFilter::Group, 221
NdbScanFilter::gt(), 222
NdbScanFilter::isfalse(), 219
NdbScanFilter::isnotnull(), 220
NdbScanFilter::isnull(), 220
NdbScanFilter::istrue(), 220
NdbScanFilter::le(), 222
NdbScanFilter::lt(), 222
NdbScanFilter::ne(), 223
NdbScanOperation
NDB API class, 223
NdbScanOperation::close(), 225
NdbScanOperation::deleteCurrentTuple(), 225
NdbScanOperation::getNdbTransaction(), 226
NdbScanOperation::getPruned(), 226
NdbScanOperation::lockCurrentTuple(), 227
NdbScanOperation::NdbIndexScanOperation
NDB API class, 162
NdbScanOperation::nextResult(), 228
NdbScanOperation::readTuples(), 229
NdbScanOperation::restart(), 230
NdbScanOperation::ScanFlag, 230
NdbScanOperation::ScanOptions
NDB API structure, 259
NdbScanOperation::updateCurrentTuple(), 231

NdbTransaction
  NDB API class, 232
NdbTransaction class methods
  using, 5
NdbTransaction::AbortOption (OBSOLETE), 195
NdbTransaction::close(), 234
NdbTransaction::commitStatus(), 235
NdbTransaction::CommitStatusType, 235
NdbTransaction::deleteTuple(), 235
NdbTransaction::ExecType, 236
NdbTransaction::execute(), 236
NdbTransaction::executePendingBlobOps(), 237
NdbTransaction::getGCI(), 238
NdbTransaction::getMaxPendingBlobReadBytes(), 238
NdbTransaction::getMaxPendingBlobWriteBytes(), 238
NdbTransaction::getNdbError(), 239
NdbTransaction::getNdbErrorLine(), 239
NdbTransaction::getNdbErrorOperation(), 239
NdbTransaction::getNdbIndexOperation(), 240
NdbTransaction::getNdbIndexScanOperation(), 240
NdbTransaction::getNdbOperation(), 240
NdbTransaction::getNdbScanOperation(), 241
NdbTransaction::getNextCompletedOperation(), 241
NdbTransaction::getTransactionId(), 241
NdbTransaction::insertTuple(), 242
NdbTransaction::readTuple(), 243
NdbTransaction::refresh(), 244
NdbTransaction::releaseLockHandle(), 244
NdbTransaction::scanIndex(), 244
NdbTransaction::scanTable(), 245
NdbTransaction::setMaxPendingBlobReadBytes(), 246
NdbTransaction::setMaxPendingBlobWriteBytes(), 246
NdbTransaction::unlock(), 247
NdbTransaction::updateTuple(), 247
NdbTransaction::writeTuple(), 248
Ndb_cluster_connection
  get_max_adaptive_send_time() method, 125
  get_next_ndb_object() method, 125
  get_num_recv_threads(), 126
  get_recv_thread_activation_threshold(), 126
  NDB API class, 121
  set_max_adaptive_send_time(), 127
  set_num_recv_threads(), 128
  set_recv_thread_activation_threshold(), 128
  set_recv_thread_cpu(), 129
  unset_recv_thread_cpu(), 130
ndb_cluster_connection
  lock_ndb_objects() method, 126
  unlock_ndb_objects() method, 130
Ndb_cluster_connection::connect(), 124
Ndb_cluster_connection::get_auto_reconnect(), 124
Ndb_cluster_connection::get_latest_error(), 124
Ndb_cluster_connection::get_latest_error_msg(), 125
Ndb_cluster_connection::set_auto_reconnect(), 127
Ndb_cluster_connection::set_data_node_neighbour(), 127
Ndb_cluster_connection::set_name(), 128
Ndb_cluster_connection::set_optimized_node_selection(), 128
Ndb_cluster_connection::set_timeout(), 129
Ndb_cluster_connection::wait_until_ready(), 130
ndb_end()
  NDB API function, 19
ndb_init()
  NDB API function, 19
ndb_logevent structure (MGM API), 514
ndb_logevent_get_fd() function (MGM API), 486
ndb_logevent_get_latest_error() function (MGM API), 488
ndb_logevent_get_latest_error_msg() function (MGM API), 488
ndb_logevent_get_next() function (MGM API), 486
ndb_logevent_get_next2() function (MGM API), 487
ndb_logevent_handle_error type (MGM API), 513
Ndb_logevent_type type (MGM API), 508
ndb_memcache_metadata.sql, 646
ndb_mgm_abort_backup() function (MGM API), 506
ndb_mgm_check_connection() function (MGM API), 493
ndb_mgm_cluster_state structure (MGM API), 521
ndb_mgm_connect() function (MGM API), 496
ndb_mgm_create_handle() function (MGM API), 490
ndb_mgm_create_logevent_handle() function (MGM API), 486, 486
ndb_mgm_destroy_handle() function (MGM API), 491
ndb_mgm_destroy_logevent_handle() function (MGM API), 486
ndb_mgm_disconnect() function (MGM API), 496
ndb_mgm_dump_state() function (MGM API), 497
ndb_mgm_enter_single_user() function (MGM API), 507
ndb_mgm_error type (MGM API), 508
ndb_mgm_event_category type (MGM API), 514
ndb_mgm_event_severity type (MGM API), 513
ndb_mgm_exit_single_user() function (MGM API), 507
ndb_mgm_get_clusterlog_loglevel() function (MGM API), 505
ndb_mgm_get_clusterlog_severity_filter() function (MGM API), 503
ndb_mgm_get_configuration_nodeid() function (MGM API), 492
ndb_mgm_get_connected_host() function (MGM API), 492
ndb_mgm_get_connected_port() function (MGM API), 492
ndb_mgm_get_connectstring() function (MGM API), 491
ndb_mgm_get_latest_error() function (MGM API), 488
ndb_mgm_get_latest_error_desc() function (MGM API), 489
ndb_mgm_get_latest_error_msg() function (MGM API), 489
ndb_mgm_get_loglevel_clusterlog() function (MGM API) - DEPRECATED, 505
ndb_mgm_get_status() function (MGM API), 496

ndb_mgm_get_status2() function (MGM API), 497
ndb_mgm_get_version() function (MGM API), 492
ndb_mgm_is_connected() function (MGM API), 493
ndb_mgm_listen_event() function (MGM API), 485
ndb_mgm_node_state structure (MGM API), 520
ndb_mgm_node_status type (MGM API), 508
ndb_mgm_node_type type (MGM API), 508
ndb_mgm_number_of_mgmd_in_connect_string() function (MGM API), 493
ndb_mgm_reply structure (MGM API), 521
ndb_mgm_restart() function (MGM API), 501
ndb_mgm_restart2() function (MGM API), 501
ndb_mgm_restart3() function (MGM API), 502
ndb_mgm_restart4() function (MGM API), 502
ndb_mgm_set_bindaddress() function (MGM API), 494
ndb_mgm_set_clusterlog_loglevel() function (MGM API), 505
ndb_mgm_set_clusterlog_severity_filter() function (MGM API), 504
ndb_mgm_set_configuration_nodeid() function (MGM API), 495
ndb_mgm_set_connectstring() function (MGM API), 494
ndb_mgm_set_error_stream() function (MGM API), 489
ndb_mgm_set_ignore_sigpipe() function (MGM API), 490
ndb_mgm_set_name() function (MGM API), 490
ndb_mgm_set_timeout() function (MGM API), 495
ndb_mgm_start() function (MGM API), 498
ndb_mgm_start_backup() function (MGM API), 506
ndb_mgm_stop() function (MGM API), 499
ndb_mgm_stop2() function (MGM API), 499
ndb_mgm_stop3() function (MGM API), 499
ndb_mgm_stop4() function (MGM API), 500
ne() (method of NdbScanFilter), 223
newDbug (ClusterJ), 561
newInstance (ClusterJ), 592, 593
nextEvent() (method of Ndb), 114
nextEvent2() (method of Ndb), 115
nextResult() (method of NdbScanOperation), 228
NoCommit
  defined, 5
node
  defined, 3
node failure
  defined, 3
node restart
  defined, 3
Node.js, 621
not (ClusterJ), 610, 614
NotPersistent (ClusterJ), 604
nullable (ClusterJ), 564
NullValue (ClusterJ), 604
nullValue (ClusterJ), 608
number (ClusterJ), 564

O

Object
  NDB API class, 249
Object::Datafile
  NDB API class, 42
Object::ForeignKey
  NDB API class, 78
Object::FragmentType, 250
Object::getObjectId(), 252
Object::getObjectStatus(), 253
Object::getObjectVersion(), 253
Object::HashMap
  NDB API class, 86
Object::PartitionBalance, 250
Object::State, 251
Object::Status, 251
Object::Store, 252
Object::Type, 252
obtaining ndbmemcache, 643
openSession() (method of Mynode), 628
openSession() (method of SessionFactory), 630
OperationOptions
  NDB API structure, 253
operations
  defined, 5
  scanning, 8
  single-row, 6
  transactions and, 5
or (ClusterJ), 610
output (ClusterJ), 577

P

param (ClusterJ), 614
PartitionBalance (Object data type), 250
PartitionKey (ClusterJ), 605
PartitionSpec
  NDB API structure, 256
persist (ClusterJ), 593
persist() (method of Context), 624
PersistenceCapable (ClusterJ), 606
PersistenceModifier (ClusterJ), 606
Persistent (ClusterJ), 607
pollEvents() (method of Ndb), 115
pollEvents2() (method of Ndb), 116
pop (ClusterJ), 578
precision (ClusterJ), 565
Predicate (ClusterJ), 609
PredicateOperand (ClusterJ), 610
prepareHashMap() (method of Dictionary), 65
primaryKey (ClusterJ), 608
PrimaryKey (ClusterJ), 608
print (ClusterJ), 578
PROPERTY_CLUSTER_BYTE_BUFFER_POOL_SIZES (ClusterJ), 572
PROPERTY_CLUSTER_CONNECTION_SERVICE (ClusterJ), 573
PROPERTY_CLUSTER_CONNECTSTRING (ClusterJ), 573
PROPERTY_CLUSTER_CONNECT_AUTO_INCREMENT_BAT (ClusterJ), 572

PROPERTY_CLUSTER_CONNECT_AUTO_INCREMENT_START (ClusterJ), 572
PROPERTY_CLUSTER_CONNECT_AUTO_INCREMENT_STEP (ClusterJ), 572
PROPERTY_CLUSTER_CONNECT_DELAY (ClusterJ), 572
PROPERTY_CLUSTER_CONNECT_RETRIES (ClusterJ), 573
PROPERTY_CLUSTER_CONNECT_TIMEOUT_AFTER (ClusterJ), 573
PROPERTY_CLUSTER_CONNECT_TIMEOUT_BEFORE (ClusterJ), 573
PROPERTY_CLUSTER_CONNECT_TIMEOUT_MGM (ClusterJ), 573
PROPERTY_CLUSTER_CONNECT_VERBOSE (ClusterJ), 573
PROPERTY_CLUSTER_DATABASE (ClusterJ), 573
PROPERTY_CLUSTER_MAX_TRANSACTIONS (ClusterJ), 574
PROPERTY_CONNECTION_POOL_NODEIDS (ClusterJ), 574
PROPERTY_CONNECTION_POOL_SIZE (ClusterJ), 574
PROPERTY_DEFER_CHANGES (ClusterJ), 574
PROPERTY_JDBC_DRIVER_NAME (ClusterJ), 574
PROPERTY_JDBC_PASSWORD (ClusterJ), 574
PROPERTY_JDBC_URL (ClusterJ), 574
PROPERTY_JDBC_USERNAME (ClusterJ), 575
push (ClusterJ), 578, 578

Q

Query (ClusterJ), 581
Query.Ordering (ClusterJ), 586
QueryBuilder (ClusterJ), 613
QueryDefinition (ClusterJ), 614
QueryDomainType (ClusterJ), 615

R

readData() (method of NdbBlob), 138
readTuple() (method of NdbIndexOperation), 162
readTuple() (method of NdbOperation), 202
readTuple() (method of NdbTransaction), 243
readTuples() (method of NdbIndexScanOperation), 165
readTuples() (method of NdbScanOperation), 229
read_attr() (method of NdbInterpretedCode), 189
record structure
  NDB, 13
RecordSpecification
  NDB API structure, 257
refresh() (method of NdbTransaction), 244
Register-loading methods (NdbInterpretedCode), 171
release (ClusterJ), 593
releaseLockHandle() (method of NdbTransaction), 244
releaseRecord() (method of Dictionary), 65
remove (ClusterJ), 593
remove() (method of Context), 624
removeCachedIndex() (method of Dictionary), 66
removeCachedTable() (method of Dictionary), 65
replica
  defined, 3
reset_bounds() (method of NdbIndexScanOperation), 165
restart() (method of NdbScanOperation), 230
restore
  defined, 2
Results (ClusterJ), 586
ret_sub() (method of NdbInterpretedCode), 190
rollback (ClusterJ), 598
rollback() (method of Transaction), 633

S

save() (method of Context), 625
savePersistent (ClusterJ), 593
savePersistentAll (ClusterJ), 594
scale (ClusterJ), 565
scan operations, 8
  characteristics, 8
  used for updates or deletes, 9
  with lock handling, 10
ScanFlag (NdbScanOperation data type), 230
scanIndex() (method of NdbTransaction), 244
ScanOptions
  NDB API structure, 259
scans
  performing with NdbScanFilter and NdbScanOperation, 373
  types supported, 1
  using secondary indexes
    example, 386
    example (using NdbRecord), 390
scanTable() (method of NdbTransaction), 245
SCAN_TYPE (ClusterJ), 583
SCAN_TYPE_INDEX_SCAN (ClusterJ), 583
SCAN_TYPE_PRIMARY_KEY (ClusterJ), 583
SCAN_TYPE_TABLE_SCAN (ClusterJ), 583
SCAN_TYPE_UNIQUE_KEY (ClusterJ), 583
schema changes
  ndbmemcache, 654
Schema transactions, 51
schema transactions
  and Dictionary::prepareHashMap() method, 65
SchemaTransFlag, 58
server_roles table (ndbmemcache), 646
Session (ClusterJ), 587
Session class (Connector for JavaScript), 629
Session.close(), 629
Session.createBatch(), 629
Session.currentTransaction(), 629
Session.getMapping(), 629
Session.getTableMetadata(), 629
Session.isClosed(), 629
Session.listBatches(), 629
Session.listTables(), 629
Session.setLockMode(), 629
SessionFactory (ClusterJ), 595

SessionFactory class (Connector for JavaScript), 630
SessionFactory.close(), 630
SessionFactory.getOpenSessions(), 630
SessionFactory.openSession(), 630
SessionFactoryService (ClusterJ), 596
SESSION_FACTORY_SERVICE_CLASS_NAME (ClusterJ), 575
SESSION_FACTORY_SERVICE_FILE_NAME (ClusterJ), 575
set (ClusterJ), 578, 578
setActiveHook() (method of NdbBlob), 138
setArrayType() (method of Column), 37
setAutoGrowSpecification() (method of LogfileGroup), 101
setAutoGrowSpecification() (method of Tablespace), 290
setCharset() (method of Column), 37
setChild() (method of ForeignKey), 83
setDatabaseName() (method of Ndb), 117
setDatabaseSchemaName() (method of Ndb), 117
setDefaultLogfileGroup() (method of Tablespace), 290
setDefaultNoPartitionsFlag() (method of Table), 278
setDefaultValue() (method of Column), 37
setDurability() (method of Event), 75
setEventBufferQueueEmptyEpoch() (method of Ndb), 117
setExtentSize() (method of Tablespace), 290
setFragmentCount() (method of Table), 279
setFragmentData() (method of Table), 279
setFragmentType() (method of Table), 279
setFrm() (method of Table), 280
setHashMap() (method of Table), 280
setKValue() (method of Table), 280
setLength() (method of Column), 38
setLimits (ClusterJ), 585
setLinearFlag() (method of Table), 280
setLockMode (ClusterJ), 594
setLockMode() (method of Session), 629
setLogfileGroup() (method of Undofile), 295
setLogging() (method of Table), 281
setMap() (method of HashMap), 88
setMaxLoadFactor() (method of Table), 281
setMaxPendingBlobReadBytes() (method of NdbTransaction), 246
setMaxPendingBlobWriteBytes() (method of NdbTransaction), 246
setMaxRows() (method of Table), 281
setMinLoadFactor() (method of Table), 282
setName() (method of Column), 38
setName() (method of Event), 76
setName() (method of ForeignKey), 83
setName() (method of HashMap), 88
setName() (method of Index), 96
setName() (method of LogfileGroup), 101
setName() (method of Table), 282
setName() (method of Tablespace), 291
setNdbObjectName() (method of Ndb), 118
setNode() (method of Datafile), 47
setNode() (method of Undofile), 296
setNull() (method of NdbBlob), 139
setNull() (method of NdbDictionary), 146
setNullable() (method of Column), 39
setObjectType() (method of Table), 282
setOnDeleteAction() (method of ForeignKey), 84
setOnUpdateAction() (method of ForeignKey), 84
setOrdering (ClusterJ), 585
setParameter (ClusterJ), 585
setParent() (method of ForeignKey), 83
setPartitionBalance() (method of Table), 282
setPartitionKey (ClusterJ), 594
setPartitionKey() (method of Column), 39
setPartSize() (method of Column), 39
setPath() (method of Datafile), 47
setPath() (method of Undofile), 296
setPos() (method of NdbBlob), 139
setPrecision() (method of Column), 40
setPrimaryKey() (method of Column), 40
setRangeListData() (method of Table), 283
setReport() (method of Event), 76
setRollbackOnly (ClusterJ), 598
setRollbackOnly() (method of Transaction), 633
setRowChecksumIndicator() (method of Table), 283
setRowGCIIndicator() (method of Table), 283
setScale() (method of Column), 41
setSchemaObjectOwnerChecks(), 247
setSchemaObjectOwnerChecks() (method of NdbTransaction), 247
setSize() (method of Datafile), 47
setSize() (method of Undofile), 296
setStatusInvalid() (method of Table), 284
setStorageType() (method of Column), 41
setStripeSize() (method of Column), 41
setTable() (method of Event), 76
setTable() (method of Index), 96
setTablespace() (method of Datafile), 47
setTablespace() (method of Table), 284
setTablespaceData() (method of Table), 284
setTablespaceNames() (method of Table), 285
setType() (method of Column), 42
setType() (method of Index), 96
setUndoBufferSize() (method of LogfileGroup), 102
setValue() (method of NdbBlob), 139
setValue() (method of NdbOperation), 202
SetValueSpec
  NDB API structure, 261
set_auto_reconnect() (method of Ndb_cluster_connection), 127
set_data_node_neighbour() (method of Ndb_cluster_connection), 127
set_eventbuffer_free_percent() (method of Ndb), 118
set_eventbuf_max_alloc() (method of Ndb), 118
set_max_adaptive_send_time() (method of Ndb_cluster_connection), 127
set_name() (method of Ndb_cluster_connection), 128
set_num_recv_threads() (method of Ndb_cluster_connection), 128

set_optimized_node_selection() (method of Ndb_cluster_connection), 128
set_recv_thread_activation_threshold() (method of Ndb_cluster_connection), 126, 128
set_recv_thread_cpu() (method of Ndb_cluster_connection), 129
set_timeout() (method of Ndb_cluster_connection), 129
short_value() (method of NdbRecAttr), 212
SingleUserMode (Table data type), 285
SQL node
  defined, 3
startTransaction() (method of Ndb), 119
State (NdbBlob data type), 140
State (NdbEventOperation data type), 159
State (Object data type), 251
Status (NdbError data type), 149
Status (Object data type), 251
StorageType (Column data type), 28
Store (Object data type), 252
sub_reg() (method of NdbInterpretedCode), 190
sub_val() (method of NdbInterpretedCode), 191
system crash
  defined, 3
system restart
  defined, 3

T

Table
  NDB API class, 262
Table::addColumn(), 267
Table::aggregate(), 267
Table::equal(), 268
Table::getColumn(), 269
Table::getDefaultNoPartitionsFlag(), 268
Table::getFragmentCount(), 268
Table::getFragmentData(), 268
Table::getFragmentDataLen(), 269
Table::getFragmentNodes(), 270
Table::getFragmentType(), 270
Table::getFrmData(), 269
Table::getFrmLength(), 269
Table::getHashMap(), 270
Table::getKValue(), 271
Table::getLinearFlag(), 271
Table::getLogging(), 271
Table::getMaxLoadFactor(), 272
Table::getMaxRows(), 272
Table::getMinLoadFactor(), 272
Table::getNoOfCOlumns(), 273
Table::getNoOfPrimaryKeys(), 273
Table::getObjectId(), 273
Table::getObjectStatus(), 273
Table::getObjectType(), 274
Table::getObjectVersion(), 274
Table::getPartitionBalance(), 274
Table::getPartitionBalanceString(), 274
Table::getPartitionId(), 275
Table::getPrimaryKey(), 275
Table::getRangeListData(), 275
Table::getRangeListDataLen(), 275
Table::getRowChecksumIndicator(), 276
Table::getRowGCIIndicator(), 276
Table::getTableId(), 276
Table::getTablespace(), 276
Table::getTablespaceData(), 277
Table::getTablespaceDataLen(), 277
Table::getTablespaceNames(), 278
Table::getTablespaceNamesLen(), 278
Table::hasDefaultValues(), 278
Table::setDefaultNoPartitionsFlag(), 278
Table::setFragmentCount(), 279
Table::setFragmentData(), 279
Table::setFragmentType(), 279
Table::setFrm(), 280
Table::setHashMap(), 280
Table::setKValue(), 280
Table::setLinearFlag(), 280
Table::setLogging(), 281
Table::setMaxLoadFactor(), 281
Table::setMaxRows(), 281
Table::setMinLoadFactor(), 282
Table::setName(), 282
Table::setObjectType(), 282
Table::setPartitionBalance(), 282
Table::setRangeListData(), 283
Table::setRowChecksumIndicator(), 283
Table::setRowGCIIndicator(), 283
Table::setStatusInvalid(), 284
Table::setTablespace(), 284
Table::setTablespaceData(), 284
Table::setTablespaceNames(), 285
Table::SingleUserMode, 285
Table::validate(), 285
TableEvent (Event data type), 77
tableFragmentationChanged() (method of NdbEventOperation), 159
tableFrmChanged() (method of NdbEventOperation), 159
TableMapping class (Connector for JavaScript), 630
TableMapping.applyToClass(), 631
TableMapping.mapField(), 630
TableMetadata class (Connector for JavaScript), 631
tableNameChanged() (method of NdbEventOperation), 160
Tablespace
  NDB API class, 286
Tablespace::getAutoGrowSpecification(), 288
Tablespace::getDefaultLogfileGroup(), 288
Tablespace::getDefaultLogfileGroupId(), 288
Tablespace::getExtentSize(), 288
Tablespace::getName(), 289
Tablespace::getObjectId(), 289
Tablespace::getObjectStatus(), 289
Tablespace::getObjectVersion(), 289
Tablespace::setAutoGrowSpecification(), 290
Tablespace::setDefaultLogfileGroup(), 290

Tablespace::setExtentSize(), 290
Tablespace::setName(), 291
TC
  and NDB Kernel, 12
  defined, 4
  selecting, 12
threading, 14
trace (ClusterJ), 578
Transaction (ClusterJ), 597
Transaction class (Connector for JavaScript), 632
Transaction Coordinator
  defined, 4
transaction coordinator, 119
Transaction.begin(), 632
Transaction.commit(), 632
Transaction.getRollbackOnly(), 633
Transaction.isActive(), 633
Transaction.rollback(), 633
Transaction.setRollbackOnly(), 633
transactions
  concurrency, 13
  example, 364
  handling and transmission, 13
  performance, 13
  synchronous, 5
    example of use, 360
  using, 5
transactions and locking
  NDB API, 197
transporter
  defined, 3
truncate() (method of NdbBlob), 140
TUP
  and NDB Kernel, 12
  defined, 3
Tuple Manager
  defined, 3
Type (Column data type), 28
Type (Index data type), 97
Type (NdbOperation data type), 204
Type (Object data type), 252

U

Undofile
  NDB API class, 291
Undofile::getFileNo(), 293
Undofile::getLogfileGroup(), 293
Undofile::getLogfileGroupId(), 293
Undofile::getNode(), 294
Undofile::getObjectId(), 294
Undofile::getObjectStatus(), 294
Undofile::getObjectVersion(), 294
Undofile::getPath(), 295
Undofile::getSize(), 295
Undofile::setLogfileGroup(), 295
Undofile::setNode(), 296
Undofile::setPath(), 296
Undofile::setSize(), 296
unique (ClusterJ), 602
unloadSchema (ClusterJ), 594
unlock() (method of NdbTransaction), 247
unlock_ndb_objects() (method of ndb_cluster_connection), 130
unset_recv_thread_cpu() (method of Ndb_cluster_connection), 130
update() (method of Context), 624
updateCurrentTuple() (method of NdbScanOperation), 231
updatePersistent (ClusterJ), 595
updatePersistentAll (ClusterJ), 595
updateTuple() (method of NdbIndexOperation), 162
updateTuple() (method of NdbOperation), 205
updateTuple() (method of NdbTransaction), 247
u_32_value() (method of NdbRecAttr), 212
u_64_value() (method of NdbRecAttr), 212
u_8_value() (method of NdbRecAttr), 212
u_char_value() (method of NdbRecAttr), 213
u_medium_value() (method of NdbRecAttr), 213
u_short_value() (method of NdbRecAttr), 213

V

validate() (method of Table), 285
value (ClusterJ), 600, 601, 601, 603
vendorName (ClusterJ), 601
version information
  in MGM API, 492
visibility of database objects
  and MySQL Server, 25, 49

W

wait_until_ready() (method of Ndb_cluster_connection), 130
where (ClusterJ), 614
writeData() (method of NdbBlob), 140
writeTuple() (method of NdbOperation), 205
writeTuple() (method of NdbTransaction), 248
write_attr() (method of NdbInterpretedCode), 191