

MySQL Partitioning

Abstract

This is the MySQL Partitioning extract from the MySQL 5.6 Reference Manual.

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](#).

Licensing information—MySQL 5.6. This product may include third-party software, used under license. If you are using a *Commercial* release of MySQL 5.6, 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 MySQL 5.6, see [this document](#) for licensing information, including licensing information relating to third-party software that may be included in this Community release.

Licensing information—MySQL Cluster. This product may include third-party software, used under license. If you are using a *Commercial* release of MySQL Cluster NDB 7.3 or NDB 7.4, 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 MySQL Cluster NDB 7.3 or NDB 7.4, see [this document](#) for licensing information, including licensing information relating to third-party software that may be included in this Community release.

Document generated on: 2016-08-15 (revision: 48549)

Table of Contents

Preface and Legal Notices	v
1 Partitioning	1
2 Overview of Partitioning in MySQL	3
3 Partitioning Types	7
3.1 RANGE Partitioning	9
3.2 LIST Partitioning	13
3.3 COLUMNS Partitioning	15
3.3.1 RANGE COLUMNS partitioning	15
3.3.2 LIST COLUMNS partitioning	21
3.4 HASH Partitioning	22
3.4.1 LINEAR HASH Partitioning	24
3.5 KEY Partitioning	25
3.6 Subpartitioning	27
3.7 How MySQL Partitioning Handles NULL	30
4 Partition Management	35
4.1 Management of RANGE and LIST Partitions	36
4.2 Management of HASH and KEY Partitions	41
4.3 Exchanging Partitions and Subpartitions with Tables	42
4.4 Maintenance of Partitions	47
4.5 Obtaining Information About Partitions	49
5 Partition Pruning	53
6 Restrictions and Limitations on Partitioning	57
6.1 Partitioning Keys, Primary Keys, and Unique Keys	63
6.2 Partitioning Limitations Relating to Storage Engines	66
6.3 Partitioning Limitations Relating to Functions	67
6.4 Partitioning and Locking	68

Preface and Legal Notices

This is the MySQL Partitioning extract from the MySQL 5.6 Reference Manual.

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 Partitioning

This chapter discusses MySQL's implementation of *user-defined partitioning*. You can determine whether your MySQL Server supports partitioning by checking the output of the `SHOW PLUGINS` statement, like this:

Note

Previous versions of MySQL had the `have_partitioning` variable, which was deprecated and removed in MySQL 5.6.1.

```
mysql> SHOW PLUGINS;
```

Name	Status	Type	Library	License
binlog	ACTIVE	STORAGE ENGINE	NULL	GPL
partition	ACTIVE	STORAGE ENGINE	NULL	GPL
ARCHIVE	ACTIVE	STORAGE ENGINE	NULL	GPL
BLACKHOLE	ACTIVE	STORAGE ENGINE	NULL	GPL
CSV	ACTIVE	STORAGE ENGINE	NULL	GPL
FEDERATED	DISABLED	STORAGE ENGINE	NULL	GPL
MEMORY	ACTIVE	STORAGE ENGINE	NULL	GPL
InnoDB	ACTIVE	STORAGE ENGINE	NULL	GPL
MRG_MYISAM	ACTIVE	STORAGE ENGINE	NULL	GPL
MyISAM	ACTIVE	STORAGE ENGINE	NULL	GPL
ndbcluster	DISABLED	STORAGE ENGINE	NULL	GPL

```
11 rows in set (0.00 sec)
```

You can also check the `INFORMATION_SCHEMA.PLUGINS` table with a query similar to this one:

```
mysql> SELECT
->     PLUGIN_NAME as Name,
->     PLUGIN_VERSION as Version,
->     PLUGIN_STATUS as Status
-> FROM INFORMATION_SCHEMA.PLUGINS
-> WHERE PLUGIN_TYPE='STORAGE ENGINE';
```

Name	Version	Status
binlog	1.0	ACTIVE
CSV	1.0	ACTIVE
MEMORY	1.0	ACTIVE
MRG_MYISAM	1.0	ACTIVE
MyISAM	1.0	ACTIVE
PERFORMANCE_SCHEMA	0.1	ACTIVE
BLACKHOLE	1.0	ACTIVE
ARCHIVE	3.0	ACTIVE
InnoDB	5.6	ACTIVE
partition	1.0	ACTIVE

```
10 rows in set (0.00 sec)
```

In either case, if you do not see the `partition` plugin listed with the value `ACTIVE` for the `Status` column in the output (shown in bold text in each of the examples just given), then your version of MySQL was not built with partitioning support.

MySQL 5.6 Community binaries provided by Oracle include partitioning support. For information about partitioning support offered in MySQL Enterprise Edition binaries, see [MySQL Enterprise Edition](#).

To enable partitioning if you are compiling MySQL 5.6 from source, the build must be configured with the `-DWITH_PARTITION_STORAGE_ENGINE` option. For more information, see [Installing MySQL from Source](#).

If your MySQL binary is built with partitioning support, nothing further needs to be done to enable it (for example, no special entries are required in your `my.cnf` file).

If you want to disable partitioning support, you can start the MySQL Server with the `--skip-partition` option, in which case the value of `have_partitioning` is `DISABLED`. When partitioning support is disabled, you can see any existing partitioned tables and drop them (although doing this is not advised), but you cannot otherwise manipulate them or access their data.

See [Chapter 2, Overview of Partitioning in MySQL](#), for an introduction to partitioning and partitioning concepts.

MySQL supports several types of partitioning as well as subpartitioning; see [Chapter 3, Partitioning Types](#), and [Section 3.6, “Subpartitioning”](#).

[Chapter 4, Partition Management](#), covers methods of adding, removing, and altering partitions in existing partitioned tables.

[Section 4.4, “Maintenance of Partitions”](#), discusses table maintenance commands for use with partitioned tables.

The `PARTITIONS` table in the `INFORMATION_SCHEMA` database provides information about partitions and partitioned tables. See [The INFORMATION_SCHEMA PARTITIONS Table](#), for more information; for some examples of queries against this table, see [Section 3.7, “How MySQL Partitioning Handles NULL”](#).

For known issues with partitioning in MySQL 5.6, see [Chapter 6, Restrictions and Limitations on Partitioning](#).

You may also find the following resources to be useful when working with partitioned tables.

Additional Resources. Other sources of information about user-defined partitioning in MySQL include the following:

- [MySQL Partitioning Forum](#)

This is the official discussion forum for those interested in or experimenting with MySQL Partitioning technology. It features announcements and updates from MySQL developers and others. It is monitored by members of the Partitioning Development and Documentation Teams.

- [Mikael Ronström's Blog](#)

MySQL Partitioning Architect and Lead Developer Mikael Ronström frequently posts articles here concerning his work with MySQL Partitioning and MySQL Cluster.

- [PlanetMySQL](#)

A MySQL news site featuring MySQL-related blogs, which should be of interest to anyone using my MySQL. We encourage you to check here for links to blogs kept by those working with MySQL Partitioning, or to have your own blog added to those covered.

MySQL 5.6 binaries are available from <http://dev.mysql.com/downloads/mysql/5.6.html>.

However, for the latest partitioning bugfixes and feature additions, you can obtain the source from our GitHub repository. To enable partitioning, the build must be configured with the `-DWITH_PARTITION_STORAGE_ENGINE` option. For more information about building MySQL, see [Installing MySQL from Source](#). If you have problems compiling a partitioning-enabled MySQL 5.6 build, check the [MySQL Partitioning Forum](#) and ask for assistance there if you do not find a solution to your problem already posted.

Chapter 2 Overview of Partitioning in MySQL

This section provides a conceptual overview of partitioning in MySQL 5.6.

For information on partitioning restrictions and feature limitations, see [Chapter 6, *Restrictions and Limitations on Partitioning*](#).

The SQL standard does not provide much in the way of guidance regarding the physical aspects of data storage. The SQL language itself is intended to work independently of any data structures or media underlying the schemas, tables, rows, or columns with which it works. Nonetheless, most advanced database management systems have evolved some means of determining the physical location to be used for storing specific pieces of data in terms of the file system, hardware or even both. In MySQL, the [InnoDB](#) storage engine has long supported the notion of a tablespace, and the MySQL Server, even prior to the introduction of partitioning, could be configured to employ different physical directories for storing different databases (see [Using Symbolic Links](#), for an explanation of how this is done).

Partitioning takes this notion a step further, by enabling you to distribute portions of individual tables across a file system according to rules which you can set largely as needed. In effect, different portions of a table are stored as separate tables in different locations. The user-selected rule by which the division of data is accomplished is known as a *partitioning function*, which in MySQL can be the modulus, simple matching against a set of ranges or value lists, an internal hashing function, or a linear hashing function. The function is selected according to the partitioning type specified by the user, and takes as its parameter the value of a user-supplied expression. This expression can be a column value, a function acting on one or more column values, or a set of one or more column values, depending on the type of partitioning that is used.

In the case of [RANGE](#), [LIST](#), and [\[LINEAR\] HASH](#) partitioning, the value of the partitioning column is passed to the partitioning function, which returns an integer value representing the number of the partition in which that particular record should be stored. This function must be nonconstant and nonrandom. It may not contain any queries, but may use an SQL expression that is valid in MySQL, as long as that expression returns either [NULL](#) or an integer *intval* such that

```
-MAXVALUE <= intval <= MAXVALUE
```

([MAXVALUE](#) is used to represent the least upper bound for the type of integer in question. [-MAXVALUE](#) represents the greatest lower bound.)

For [\[LINEAR\] KEY](#), [RANGE COLUMNS](#), and [LIST COLUMNS](#) partitioning, the partitioning expression consists of a list of one or more columns.

For [\[LINEAR\] KEY](#) partitioning, the partitioning function is supplied by MySQL.

For more information about permitted partitioning column types and partitioning functions, see [Chapter 3, *Partitioning Types*](#), as well as [CREATE TABLE Syntax](#), which provides partitioning syntax descriptions and additional examples. For information about restrictions on partitioning functions, see [Section 6.3, “Partitioning Limitations Relating to Functions”](#).

This is known as *horizontal partitioning*—that is, different rows of a table may be assigned to different physical partitions. MySQL 5.6 does not support *vertical partitioning*, in which different columns of a table are assigned to different physical partitions. There are no plans at this time to introduce vertical partitioning into MySQL.

For information about determining whether your MySQL Server binary supports user-defined partitioning, see [Chapter 1, *Partitioning*](#).

For creating partitioned tables, you can use most storage engines that are supported by your MySQL server; the MySQL partitioning engine runs in a separate layer and can interact with any of these.

In MySQL 5.6, all partitions of the same partitioned table must use the same storage engine; for example, you cannot use [MyISAM](#) for one partition and [InnoDB](#) for another. However, there is nothing preventing you from using different storage engines for different partitioned tables on the same MySQL server or even in the same database.

MySQL partitioning cannot be used with the [MERGE](#), [CSV](#), or [FEDERATED](#) storage engines.

Partitioning by [KEY](#) or [LINEAR KEY](#) is possible with [NDB](#), but other types of user-defined partitioning are not supported for tables using this storage engine. In addition, an [NDB](#) table that employs user-defined partitioning must have an explicit primary key, and any columns referenced in the table's partitioning expression must be part of the primary key. However, if no columns are listed in the [PARTITION BY KEY](#) or [PARTITION BY LINEAR KEY](#) clause of the [CREATE TABLE](#) or [ALTER TABLE](#) statement used to create or modify a user-partitioned [NDB](#) table, then the table is not required to have an explicit primary key. For more information, see [Noncompliance with SQL Syntax in MySQL Cluster](#).

To employ a particular storage engine for a partitioned table, it is necessary only to use the [\[STORAGE\] ENGINE](#) option just as you would for a nonpartitioned table. However, you should keep in mind that [\[STORAGE\] ENGINE](#) (and other table options) need to be listed *before* any partitioning options are used in a [CREATE TABLE](#) statement. This example shows how to create a table that is partitioned by hash into 6 partitions and which uses the [InnoDB](#) storage engine:

```
CREATE TABLE ti (id INT, amount DECIMAL(7,2), tr_date DATE)
ENGINE=INNODB
PARTITION BY HASH( MONTH(tr_date) )
PARTITIONS 6;
```

Each [PARTITION](#) clause can include a [\[STORAGE\] ENGINE](#) option, but in MySQL 5.6 this has no effect.

Important

Partitioning applies to all data and indexes of a table; you cannot partition only the data and not the indexes, or vice versa, nor can you partition only a portion of the table.

Data and indexes for each partition can be assigned to a specific directory using the [DATA DIRECTORY](#) and [INDEX DIRECTORY](#) options for the [PARTITION](#) clause of the [CREATE TABLE](#) statement used to create the partitioned table.

[DATA DIRECTORY](#) and [INDEX DIRECTORY](#) are not supported for individual partitions or subpartitions of [MyISAM](#) tables on Windows. They are supported for individual partitions and subpartitions of [InnoDB](#) tables (as on all platforms).

All columns used in the table's partitioning expression must be part of every unique key that the table may have, including any primary key. This means that a table such as this one, created by the following SQL statement, cannot be partitioned:

```
CREATE TABLE tnp (
  id INT NOT NULL AUTO_INCREMENT,
  ref BIGINT NOT NULL,
  name VARCHAR(255),
  PRIMARY KEY pk (id),
  UNIQUE KEY uk (name)
);
```

Because the keys [pk](#) and [uk](#) have no columns in common, there are no columns available for use in a partitioning expression. Possible workarounds in this situation include adding the [name](#) column to the table's primary key, adding the [id](#) column to [uk](#), or simply removing the unique key altogether. See [Section 6.1, "Partitioning Keys, Primary Keys, and Unique Keys"](#), for more information.

In addition, `MAX_ROWS` and `MIN_ROWS` can be used to determine the maximum and minimum numbers of rows, respectively, that can be stored in each partition. The `MAX_ROWS` option can be useful for causing MySQL Cluster tables to be created with extra partitions, thus allowing for greater storage of hash indexes. See the documentation for the `DataMemory` data node configuration parameter, as well as [MySQL Cluster Nodes, Node Groups, Replicas, and Partitions](#), for more information.

Some advantages of partitioning are listed here:

- Partitioning makes it possible to store more data in one table than can be held on a single disk or file system partition.
- Data that loses its usefulness can often be easily removed from a partitioned table by dropping the partition (or partitions) containing only that data. Conversely, the process of adding new data can in some cases be greatly facilitated by adding one or more new partitions for storing specifically that data.
- Some queries can be greatly optimized in virtue of the fact that data satisfying a given `WHERE` clause can be stored only on one or more partitions, which automatically excludes any remaining partitions from the search. Because partitions can be altered after a partitioned table has been created, you can reorganize your data to enhance frequent queries that may not have been often used when the partitioning scheme was first set up. This ability to exclude non-matching partitions (and thus any rows they contain) is often referred to as *partition pruning*. For more information, see [Chapter 5, Partition Pruning](#).

In addition, MySQL 5.6 supports explicit partition selection for queries. For example, `SELECT * FROM t PARTITION (p0,p1) WHERE c < 5` selects only those rows in partitions `p0` and `p1` that match the `WHERE` condition. In this case, MySQL does not check any other partitions of table `t`; this can greatly speed up queries when you already know which partition or partitions you wish to examine. Partition selection is also supported for the data modification statements `DELETE`, `INSERT`, `REPLACE`, `UPDATE`, and `LOAD DATA`, `LOAD XML`. See the descriptions of these statements for more information and examples.

Other benefits usually associated with partitioning include those in the following list. These features are not currently implemented in MySQL Partitioning, but are high on our list of priorities.

- Queries involving aggregate functions such as `SUM()` and `COUNT()` can easily be parallelized. A simple example of such a query might be `SELECT salesperson_id, COUNT(orders) as order_total FROM sales GROUP BY salesperson_id;`. By “parallelized,” we mean that the query can be run simultaneously on each partition, and the final result obtained merely by summing the results obtained for all partitions.
- Achieving greater query throughput in virtue of spreading data seeks over multiple disks.

Be sure to check this section and chapter frequently for updates as MySQL Partitioning development continues.

Chapter 3 Partitioning Types

Table of Contents

3.1 RANGE Partitioning	9
3.2 LIST Partitioning	13
3.3 COLUMNS Partitioning	15
3.3.1 RANGE COLUMNS partitioning	15
3.3.2 LIST COLUMNS partitioning	21
3.4 HASH Partitioning	22
3.4.1 LINEAR HASH Partitioning	24
3.5 KEY Partitioning	25
3.6 Subpartitioning	27
3.7 How MySQL Partitioning Handles NULL	30

This section discusses the types of partitioning which are available in MySQL 5.6. These include the types listed here:

- **RANGE partitioning.** This type of partitioning assigns rows to partitions based on column values falling within a given range. See [Section 3.1, “RANGE Partitioning”](#). For information about an extension to this type, [RANGE COLUMNS](#), see [Section 3.3.1, “RANGE COLUMNS partitioning”](#).
- **LIST partitioning.** Similar to partitioning by [RANGE](#), except that the partition is selected based on columns matching one of a set of discrete values. See [Section 3.2, “LIST Partitioning”](#). For information about an extension to this type, [LIST COLUMNS](#), see [Section 3.3.2, “LIST COLUMNS partitioning”](#).
- **HASH partitioning.** With this type of partitioning, a partition is selected based on the value returned by a user-defined expression that operates on column values in rows to be inserted into the table. The function may consist of any expression valid in MySQL that yields a nonnegative integer value. An extension to this type, [LINEAR HASH](#), is also available. See [Section 3.4, “HASH Partitioning”](#).
- **KEY partitioning.** This type of partitioning is similar to partitioning by [HASH](#), except that only one or more columns to be evaluated are supplied, and the MySQL server provides its own hashing function. These columns can contain other than integer values, since the hashing function supplied by MySQL guarantees an integer result regardless of the column data type. An extension to this type, [LINEAR KEY](#), is also available. See [Section 3.5, “KEY Partitioning”](#).

A very common use of database partitioning is to segregate data by date. Some database systems support explicit date partitioning, which MySQL does not implement in 5.6. However, it is not difficult in MySQL to create partitioning schemes based on [DATE](#), [TIME](#), or [DATETIME](#) columns, or based on expressions making use of such columns.

When partitioning by [KEY](#) or [LINEAR KEY](#), you can use a [DATE](#), [TIME](#), or [DATETIME](#) column as the partitioning column without performing any modification of the column value. For example, this table creation statement is perfectly valid in MySQL:

```
CREATE TABLE members (  
    firstname VARCHAR(25) NOT NULL,  
    lastname VARCHAR(25) NOT NULL,  
    username VARCHAR(16) NOT NULL,  
    email VARCHAR(35),  
    joined DATE NOT NULL  
)  
PARTITION BY KEY(joined)
```

```
PARTITIONS 6;
```

In MySQL 5.6, it is also possible to use a [DATE](#) or [DATETIME](#) column as the partitioning column using [RANGE COLUMNS](#) and [LIST COLUMNS](#) partitioning.

MySQL's other partitioning types, however, require a partitioning expression that yields an integer value or [NULL](#). If you wish to use date-based partitioning by [RANGE](#), [LIST](#), [HASH](#), or [LINEAR HASH](#), you can simply employ a function that operates on a [DATE](#), [TIME](#), or [DATETIME](#) column and returns such a value, as shown here:

```
CREATE TABLE members (
  firstname VARCHAR(25) NOT NULL,
  lastname VARCHAR(25) NOT NULL,
  username VARCHAR(16) NOT NULL,
  email VARCHAR(35),
  joined DATE NOT NULL
)
PARTITION BY RANGE( YEAR(joined) ) (
  PARTITION p0 VALUES LESS THAN (1960),
  PARTITION p1 VALUES LESS THAN (1970),
  PARTITION p2 VALUES LESS THAN (1980),
  PARTITION p3 VALUES LESS THAN (1990),
  PARTITION p4 VALUES LESS THAN MAXVALUE
);
```

Additional examples of partitioning using dates may be found in the following sections of this chapter:

- [Section 3.1, “RANGE Partitioning”](#)
- [Section 3.4, “HASH Partitioning”](#)
- [Section 3.4.1, “LINEAR HASH Partitioning”](#)

For more complex examples of date-based partitioning, see the following sections:

- [Chapter 5, *Partition Pruning*](#)
- [Section 3.6, “Subpartitioning”](#)

MySQL partitioning is optimized for use with the [TO_DAYS\(\)](#), [YEAR\(\)](#), and [TO_SECONDS\(\)](#) functions. However, you can use other date and time functions that return an integer or [NULL](#), such as [WEEKDAY\(\)](#), [DAYOFYEAR\(\)](#), or [MONTH\(\)](#). See [Date and Time Functions](#), for more information about such functions.

It is important to remember—regardless of the type of partitioning that you use—that partitions are always numbered automatically and in sequence when created, starting with [0](#). When a new row is inserted into a partitioned table, it is these partition numbers that are used in identifying the correct partition. For example, if your table uses 4 partitions, these partitions are numbered [0](#), [1](#), [2](#), and [3](#). For the [RANGE](#) and [LIST](#) partitioning types, it is necessary to ensure that there is a partition defined for each partition number. For [HASH](#) partitioning, the user function employed must return an integer value greater than [0](#). For [KEY](#) partitioning, this issue is taken care of automatically by the hashing function which the MySQL server employs internally.

Names of partitions generally follow the rules governing other MySQL identifiers, such as those for tables and databases. However, partition names are not case-sensitive. For example, the following [CREATE TABLE](#) statement fails as shown:

```
mysql> CREATE TABLE t2 (val INT)
-> PARTITION BY LIST(val){
->   PARTITION mypart VALUES IN (1,3,5),
->   PARTITION MyPart VALUES IN (2,4,6)
-> };
ERROR 1488 (HY000): Duplicate partition name mypart
```

Failure occurs because MySQL sees no difference between the partition names `mypart` and `MyPart`.

When you specify the number of partitions for the table, this must be expressed as a positive, nonzero integer literal with no leading zeros, and may not be an expression such as `0.8E+01` or `6-2`, even if it evaluates to an integer value. Decimal fractions are not permitted.

In the sections that follow, we do not necessarily provide all possible forms for the syntax that can be used for creating each partition type; this information may be found in [CREATE TABLE Syntax](#).

3.1 RANGE Partitioning

A table that is partitioned by range is partitioned in such a way that each partition contains rows for which the partitioning expression value lies within a given range. Ranges should be contiguous but not overlapping, and are defined using the `VALUES LESS THAN` operator. For the next few examples, suppose that you are creating a table such as the following to hold personnel records for a chain of 20 video stores, numbered 1 through 20:

```
CREATE TABLE employees (
  id INT NOT NULL,
  fname VARCHAR(30),
  lname VARCHAR(30),
  hired DATE NOT NULL DEFAULT '1970-01-01',
  separated DATE NOT NULL DEFAULT '9999-12-31',
  job_code INT NOT NULL,
  store_id INT NOT NULL
);
```

Note

The `employees` table used here has no primary or unique keys. While the examples work as shown for purposes of the present discussion, you should keep in mind that tables are extremely likely in practice to have primary keys, unique keys, or both, and that allowable choices for partitioning columns depend on the columns used for these keys, if any are present. For a discussion of these issues, see [Section 6.1, “Partitioning Keys, Primary Keys, and Unique Keys”](#).

This table can be partitioned by range in a number of ways, depending on your needs. One way would be to use the `store_id` column. For instance, you might decide to partition the table 4 ways by adding a `PARTITION BY RANGE` clause as shown here:

```
CREATE TABLE employees (
  id INT NOT NULL,
  fname VARCHAR(30),
  lname VARCHAR(30),
  hired DATE NOT NULL DEFAULT '1970-01-01',
  separated DATE NOT NULL DEFAULT '9999-12-31',
  job_code INT NOT NULL,
  store_id INT NOT NULL
)
PARTITION BY RANGE (store_id) (
  PARTITION p0 VALUES LESS THAN (6),
  PARTITION p1 VALUES LESS THAN (11),
  PARTITION p2 VALUES LESS THAN (16),
  PARTITION p3 VALUES LESS THAN (21)
);
```

In this partitioning scheme, all rows corresponding to employees working at stores 1 through 5 are stored in partition `p0`, to those employed at stores 6 through 10 are stored in partition `p1`, and so on. Note that each partition is defined in order, from lowest to highest. This is a requirement of the `PARTITION BY RANGE` syntax; you can think of it as being analogous to a series of `if ... elseif ...` statements in C or Java in this regard.

It is easy to determine that a new row containing the data (72, 'Mitchell', 'Wilson', '1998-06-25', NULL, 13) is inserted into partition `p2`, but what happens when your chain adds a 21st store? Under this scheme, there is no rule that covers a row whose `store_id` is greater than 20, so an error results because the server does not know where to place it. You can keep this from occurring by using a “catchall” `VALUES LESS THAN` clause in the `CREATE TABLE` statement that provides for all values greater than the highest value explicitly named:

```
CREATE TABLE employees (
  id INT NOT NULL,
  fname VARCHAR(30),
  lname VARCHAR(30),
  hired DATE NOT NULL DEFAULT '1970-01-01',
  separated DATE NOT NULL DEFAULT '9999-12-31',
  job_code INT NOT NULL,
  store_id INT NOT NULL
)
PARTITION BY RANGE (store_id) (
  PARTITION p0 VALUES LESS THAN (6),
  PARTITION p1 VALUES LESS THAN (11),
  PARTITION p2 VALUES LESS THAN (16),
  PARTITION p3 VALUES LESS THAN MAXVALUE
);
```

Note

Another way to avoid an error when no matching value is found is to use the `IGNORE` keyword as part of the `INSERT` statement. For an example, see [Section 3.2, “LIST Partitioning”](#). Also see [INSERT Syntax](#), for general information about `IGNORE`.

`MAXVALUE` represents an integer value that is always greater than the largest possible integer value (in mathematical language, it serves as a *least upper bound*). Now, any rows whose `store_id` column value is greater than or equal to 16 (the highest value defined) are stored in partition `p3`. At some point in the future—when the number of stores has increased to 25, 30, or more—you can use an `ALTER TABLE` statement to add new partitions for stores 21-25, 26-30, and so on (see [Chapter 4, Partition Management](#), for details of how to do this).

In much the same fashion, you could partition the table based on employee job codes—that is, based on ranges of `job_code` column values. For example—assuming that two-digit job codes are used for regular (in-store) workers, three-digit codes are used for office and support personnel, and four-digit codes are used for management positions—you could create the partitioned table using the following statement:

```
CREATE TABLE employees (
  id INT NOT NULL,
  fname VARCHAR(30),
  lname VARCHAR(30),
  hired DATE NOT NULL DEFAULT '1970-01-01',
  separated DATE NOT NULL DEFAULT '9999-12-31',
  job_code INT NOT NULL,
  store_id INT NOT NULL
)
PARTITION BY RANGE (job_code) (
  PARTITION p0 VALUES LESS THAN (100),
  PARTITION p1 VALUES LESS THAN (1000),
  PARTITION p2 VALUES LESS THAN (10000)
);
```

In this instance, all rows relating to in-store workers would be stored in partition `p0`, those relating to office and support staff in `p1`, and those relating to managers in partition `p2`.

It is also possible to use an expression in `VALUES LESS THAN` clauses. However, MySQL must be able to evaluate the expression's return value as part of a `LESS THAN (<)` comparison.

Rather than splitting up the table data according to store number, you can use an expression based on one of the two [DATE](#) columns instead. For example, let us suppose that you wish to partition based on the year that each employee left the company; that is, the value of [YEAR\(separated\)](#). An example of a [CREATE TABLE](#) statement that implements such a partitioning scheme is shown here:

```
CREATE TABLE employees (
  id INT NOT NULL,
  fname VARCHAR(30),
  lname VARCHAR(30),
  hired DATE NOT NULL DEFAULT '1970-01-01',
  separated DATE NOT NULL DEFAULT '9999-12-31',
  job_code INT,
  store_id INT
)
PARTITION BY RANGE ( YEAR(separated) ) (
  PARTITION p0 VALUES LESS THAN (1991),
  PARTITION p1 VALUES LESS THAN (1996),
  PARTITION p2 VALUES LESS THAN (2001),
  PARTITION p3 VALUES LESS THAN MAXVALUE
);
```

In this scheme, for all employees who left before 1991, the rows are stored in partition [p0](#); for those who left in the years 1991 through 1995, in [p1](#); for those who left in the years 1996 through 2000, in [p2](#); and for any workers who left after the year 2000, in [p3](#).

It is also possible to partition a table by [RANGE](#), based on the value of a [TIMESTAMP](#) column, using the [UNIX_TIMESTAMP\(\)](#) function, as shown in this example:

```
CREATE TABLE quarterly_report_status (
  report_id INT NOT NULL,
  report_status VARCHAR(20) NOT NULL,
  report_updated TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP
)
PARTITION BY RANGE ( UNIX_TIMESTAMP(report_updated) ) (
  PARTITION p0 VALUES LESS THAN ( UNIX_TIMESTAMP('2008-01-01 00:00:00') ),
  PARTITION p1 VALUES LESS THAN ( UNIX_TIMESTAMP('2008-04-01 00:00:00') ),
  PARTITION p2 VALUES LESS THAN ( UNIX_TIMESTAMP('2008-07-01 00:00:00') ),
  PARTITION p3 VALUES LESS THAN ( UNIX_TIMESTAMP('2008-10-01 00:00:00') ),
  PARTITION p4 VALUES LESS THAN ( UNIX_TIMESTAMP('2009-01-01 00:00:00') ),
  PARTITION p5 VALUES LESS THAN ( UNIX_TIMESTAMP('2009-04-01 00:00:00') ),
  PARTITION p6 VALUES LESS THAN ( UNIX_TIMESTAMP('2009-07-01 00:00:00') ),
  PARTITION p7 VALUES LESS THAN ( UNIX_TIMESTAMP('2009-10-01 00:00:00') ),
  PARTITION p8 VALUES LESS THAN ( UNIX_TIMESTAMP('2010-01-01 00:00:00') ),
  PARTITION p9 VALUES LESS THAN (MAXVALUE)
);
```

Any other expressions involving [TIMESTAMP](#) values are not permitted. (See Bug #42849.)

Range partitioning is particularly useful when one or more of the following conditions is true:

- You want or need to delete “old” data. If you are using the partitioning scheme shown previously for the [employees](#) table, you can simply use [ALTER TABLE employees DROP PARTITION p0](#); to delete all rows relating to employees who stopped working for the firm prior to 1991. (See [ALTER TABLE Syntax](#), and [Chapter 4, Partition Management](#), for more information.) For a table with a great many rows, this can be much more efficient than running a [DELETE](#) query such as [DELETE FROM employees WHERE YEAR\(separated\) <= 1990](#);
- You want to use a column containing date or time values, or containing values arising from some other series.
- You frequently run queries that depend directly on the column used for partitioning the table. For example, when executing a query such as [EXPLAIN PARTITIONS SELECT COUNT\(*\) FROM employees WHERE separated BETWEEN '2000-01-01' AND '2000-12-31' GROUP BY store_id](#);, MySQL can quickly determine that only partition [p2](#) needs to be scanned because

the remaining partitions cannot contain any records satisfying the [WHERE](#) clause. See [Chapter 5, Partition Pruning](#), for more information about how this is accomplished.

A variant on this type of partitioning is [RANGE COLUMNS](#) partitioning. Partitioning by [RANGE COLUMNS](#) makes it possible to employ multiple columns for defining partitioning ranges that apply both to placement of rows in partitions and for determining the inclusion or exclusion of specific partitions when performing partition pruning. See [Section 3.3.1, “RANGE COLUMNS partitioning”](#), for more information.

Partitioning schemes based on time intervals. If you wish to implement a partitioning scheme based on ranges or intervals of time in MySQL 5.6, you have two options:

1. Partition the table by [RANGE](#), and for the partitioning expression, employ a function operating on a [DATE](#), [TIME](#), or [DATETIME](#) column and returning an integer value, as shown here:

```
CREATE TABLE members (
  firstname VARCHAR(25) NOT NULL,
  lastname VARCHAR(25) NOT NULL,
  username VARCHAR(16) NOT NULL,
  email VARCHAR(35),
  joined DATE NOT NULL
)
PARTITION BY RANGE( YEAR(joined) ) (
  PARTITION p0 VALUES LESS THAN (1960),
  PARTITION p1 VALUES LESS THAN (1970),
  PARTITION p2 VALUES LESS THAN (1980),
  PARTITION p3 VALUES LESS THAN (1990),
  PARTITION p4 VALUES LESS THAN MAXVALUE
);
```

In MySQL 5.6, it is also possible to partition a table by [RANGE](#) based on the value of a [TIMESTAMP](#) column, using the [UNIX_TIMESTAMP\(\)](#) function, as shown in this example:

```
CREATE TABLE quarterly_report_status (
  report_id INT NOT NULL,
  report_status VARCHAR(20) NOT NULL,
  report_updated TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP
)
PARTITION BY RANGE ( UNIX_TIMESTAMP(report_updated) ) (
  PARTITION p0 VALUES LESS THAN ( UNIX_TIMESTAMP('2008-01-01 00:00:00') ),
  PARTITION p1 VALUES LESS THAN ( UNIX_TIMESTAMP('2008-04-01 00:00:00') ),
  PARTITION p2 VALUES LESS THAN ( UNIX_TIMESTAMP('2008-07-01 00:00:00') ),
  PARTITION p3 VALUES LESS THAN ( UNIX_TIMESTAMP('2008-10-01 00:00:00') ),
  PARTITION p4 VALUES LESS THAN ( UNIX_TIMESTAMP('2009-01-01 00:00:00') ),
  PARTITION p5 VALUES LESS THAN ( UNIX_TIMESTAMP('2009-04-01 00:00:00') ),
  PARTITION p6 VALUES LESS THAN ( UNIX_TIMESTAMP('2009-07-01 00:00:00') ),
  PARTITION p7 VALUES LESS THAN ( UNIX_TIMESTAMP('2009-10-01 00:00:00') ),
  PARTITION p8 VALUES LESS THAN ( UNIX_TIMESTAMP('2010-01-01 00:00:00') ),
  PARTITION p9 VALUES LESS THAN (MAXVALUE)
);
```

In MySQL 5.6, any other expressions involving [TIMESTAMP](#) values are not permitted. (See Bug #42849.)

Note

It is also possible in MySQL 5.6 to use [UNIX_TIMESTAMP\(timestamp_column\)](#) as a partitioning expression for tables that are partitioned by [LIST](#). However, it is usually not practical to do so.

2. Partition the table by [RANGE COLUMNS](#), using a [DATE](#) or [DATETIME](#) column as the partitioning column. For example, the [members](#) table could be defined using the [joined](#) column directly, as shown here:

```
CREATE TABLE members (
  firstname VARCHAR(25) NOT NULL,
  lastname VARCHAR(25) NOT NULL,
  username VARCHAR(16) NOT NULL,
  email VARCHAR(35),
  joined DATE NOT NULL
)
PARTITION BY RANGE COLUMNS(joined) (
  PARTITION p0 VALUES LESS THAN ('1960-01-01'),
  PARTITION p1 VALUES LESS THAN ('1970-01-01'),
  PARTITION p2 VALUES LESS THAN ('1980-01-01'),
  PARTITION p3 VALUES LESS THAN ('1990-01-01'),
  PARTITION p4 VALUES LESS THAN MAXVALUE
);
```

Note

The use of partitioning columns employing date or time types other than [DATE](#) or [DATETIME](#) is not supported with [RANGE COLUMNS](#).

3.2 LIST Partitioning

List partitioning in MySQL is similar to range partitioning in many ways. As in partitioning by [RANGE](#), each partition must be explicitly defined. The chief difference between the two types of partitioning is that, in list partitioning, each partition is defined and selected based on the membership of a column value in one of a set of value lists, rather than in one of a set of contiguous ranges of values. This is done by using [PARTITION BY LIST\(*expr*\)](#) where *expr* is a column value or an expression based on a column value and returning an integer value, and then defining each partition by means of a [VALUES IN \(*value_list*\)](#), where *value_list* is a comma-separated list of integers.

Note

In MySQL 5.6, it is possible to match against only a list of integers (and possibly [NULL](#)—see [Section 3.7, “How MySQL Partitioning Handles NULL”](#)) when partitioning by [LIST](#).

However, other column types may be used in value lists when employing [LIST COLUMN](#) partitioning, which is described later in this section.

Unlike the case with partitions defined by range, list partitions do not need to be declared in any particular order. For more detailed syntactical information, see [CREATE TABLE Syntax](#).

For the examples that follow, we assume that the basic definition of the table to be partitioned is provided by the [CREATE TABLE](#) statement shown here:

```
CREATE TABLE employees (
  id INT NOT NULL,
  fname VARCHAR(30),
  lname VARCHAR(30),
  hired DATE NOT NULL DEFAULT '1970-01-01',
  separated DATE NOT NULL DEFAULT '9999-12-31',
  job_code INT,
  store_id INT
);
```

(This is the same table used as a basis for the examples in [Section 3.1, “RANGE Partitioning”](#).)

Suppose that there are 20 video stores distributed among 4 franchises as shown in the following table.

Region	Store ID Numbers
North	3, 5, 6, 9, 17
East	1, 2, 10, 11, 19, 20

Region	Store ID Numbers
West	4, 12, 13, 14, 18
Central	7, 8, 15, 16

To partition this table in such a way that rows for stores belonging to the same region are stored in the same partition, you could use the [CREATE TABLE](#) statement shown here:

```
CREATE TABLE employees (
  id INT NOT NULL,
  fname VARCHAR(30),
  lname VARCHAR(30),
  hired DATE NOT NULL DEFAULT '1970-01-01',
  separated DATE NOT NULL DEFAULT '9999-12-31',
  job_code INT,
  store_id INT
)
PARTITION BY LIST(store_id) (
  PARTITION pNorth VALUES IN (3,5,6,9,17),
  PARTITION pEast VALUES IN (1,2,10,11,19,20),
  PARTITION pWest VALUES IN (4,12,13,14,18),
  PARTITION pCentral VALUES IN (7,8,15,16)
);
```

This makes it easy to add or drop employee records relating to specific regions to or from the table. For instance, suppose that all stores in the West region are sold to another company. In MySQL 5.6, all rows relating to employees working at stores in that region can be deleted with the query [ALTER TABLE employees TRUNCATE PARTITION pWest](#), which can be executed much more efficiently than the equivalent [DELETE FROM employees WHERE store_id IN \(4,12,13,14,18\)](#); (Using [ALTER TABLE employees DROP PARTITION pWest](#) would also delete all of these rows, but would also remove the partition `pWest` from the definition of the table; you would need to use an [ALTER TABLE ... ADD PARTITION](#) statement to restore the table's original partitioning scheme.)

As with [RANGE](#) partitioning, it is possible to combine [LIST](#) partitioning with partitioning by hash or key to produce a composite partitioning (subpartitioning). See [Section 3.6, “Subpartitioning”](#).

Unlike the case with [RANGE](#) partitioning, there is no “catch-all” such as [MAXVALUE](#); all expected values for the partitioning expression should be covered in [PARTITION ... VALUES IN \(...\)](#) clauses. An [INSERT](#) statement containing an unmatched partitioning column value fails with an error, as shown in this example:

```
mysql> CREATE TABLE h2 (
->   c1 INT,
->   c2 INT
-> )
-> PARTITION BY LIST(c1) (
->   PARTITION p0 VALUES IN (1, 4, 7),
->   PARTITION p1 VALUES IN (2, 5, 8)
-> );
Query OK, 0 rows affected (0.11 sec)
mysql> INSERT INTO h2 VALUES (3, 5);
ERROR 1525 (HY000): Table has no partition for value 3
```

When inserting multiple rows using a single [INSERT](#) statement the behavior depends on whether the table uses a transactional storage engine. For an [InnoDB](#) table, the statement is considered a single transaction, so the presence of any unmatched values causes the statement to fail completely, and no rows are inserted. For a table using a nontransactional storage engine such as [MyISAM](#), any rows coming before the row containing the unmatched value are inserted, but any coming after it are not.

You can cause this type of error to be ignored by using the [IGNORE](#) keyword. If you do so, rows containing unmatched partitioning column values are not inserted, but any rows with matching values are inserted, and no errors are reported:

```
mysql> TRUNCATE h2;
Query OK, 1 row affected (0.00 sec)
mysql> SELECT * FROM h2;
Empty set (0.00 sec)
mysql> INSERT IGNORE INTO h2 VALUES (2, 5), (6, 10), (7, 5), (3, 1), (1, 9);
Query OK, 3 rows affected (0.00 sec)
Records: 5  Duplicates: 2  Warnings: 0
mysql> SELECT * FROM h2;
+-----+-----+
| c1    | c2    |
+-----+-----+
| 7     | 5     |
| 1     | 9     |
| 2     | 5     |
+-----+-----+
3 rows in set (0.00 sec)
```

MySQL 5.6 provides support for **LIST COLUMNS** partitioning. This is a variant of **LIST** partitioning that enables you to use columns of types other than integer types for partitioning columns, as well as to use multiple columns as partitioning keys. For more information, see [Section 3.3.2, “LIST COLUMNS partitioning”](#).

3.3 COLUMNS Partitioning

The next two sections discuss **COLUMNS partitioning**, which are variants on **RANGE** and **LIST** partitioning. **COLUMNS** partitioning enables the use of multiple columns in partitioning keys. All of these columns are taken into account both for the purpose of placing rows in partitions and for the determination of which partitions are to be checked for matching rows in partition pruning.

In addition, both **RANGE COLUMNS** partitioning and **LIST COLUMNS** partitioning support the use of non-integer columns for defining value ranges or list members. The permitted data types are shown in the following list:

- All integer types: **TINYINT**, **SMALLINT**, **MEDIUMINT**, **INT** (**INTEGER**), and **BIGINT**. (This is the same as with partitioning by **RANGE** and **LIST**.)

Other numeric data types (such as **DECIMAL** or **FLOAT**) are not supported as partitioning columns.

- **DATE** and **DATETIME**.

Columns using other data types relating to dates or times are not supported as partitioning columns.

- The following string types: **CHAR**, **VARCHAR**, **BINARY**, and **VARBINARY**.

TEXT and **BLOB** columns are not supported as partitioning columns.

The discussions of **RANGE COLUMNS** and **LIST COLUMNS** partitioning in the next two sections assume that you are already familiar with partitioning based on ranges and lists as supported in MySQL 5.1 and later; for more information about these, see [Section 3.1, “RANGE Partitioning”](#), and [Section 3.2, “LIST Partitioning”](#), respectively.

3.3.1 RANGE COLUMNS partitioning

Range columns partitioning is similar to range partitioning, but enables you to define partitions using ranges based on multiple column values. In addition, you can define the ranges using columns of types other than integer types.

RANGE COLUMNS partitioning differs significantly from **RANGE** partitioning in the following ways:

- **RANGE COLUMNS** does not accept expressions, only names of columns.
- **RANGE COLUMNS** accepts a list of one or more columns.

RANGE COLUMNS partitions are based on comparisons between *tuples* (lists of column values) rather than comparisons between scalar values. Placement of rows in **RANGE COLUMNS** partitions is also based on comparisons between tuples; this is discussed further later in this section.

- **RANGE COLUMNS** partitioning columns are not restricted to integer columns; string, **DATE** and **DATETIME** columns can also be used as partitioning columns. (See [Section 3.3, “COLUMNS Partitioning”](#), for details.)

The basic syntax for creating a table partitioned by **RANGE COLUMNS** is shown here:

```
CREATE TABLE table_name
PARTITIONED BY RANGE COLUMNS(column_list) (
    PARTITION partition_name VALUES LESS THAN (value_list)[,
    PARTITION partition_name VALUES LESS THAN (value_list)][,
    ...]
)
column_list:
    column_name[, column_name][, ...]
value_list:
    value[, value][, ...]
```

Note

Not all **CREATE TABLE** options that can be used when creating partitioned tables are shown here. For complete information, see [CREATE TABLE Syntax](#).

In the syntax just shown, *column_list* is a list of one or more columns (sometimes called a *partitioning column list*), and *value_list* is a list of values (that is, it is a *partition definition value list*). A *value_list* must be supplied for each partition definition, and each *value_list* must have the same number of values as the *column_list* has columns. Generally speaking, if you use *N* columns in the **COLUMNS** clause, then each **VALUES LESS THAN** clause must also be supplied with a list of *N* values.

The elements in the partitioning column list and in the value list defining each partition must occur in the same order. In addition, each element in the value list must be of the same data type as the corresponding element in the column list. However, the order of the column names in the partitioning column list and the value lists does not have to be the same as the order of the table column definitions in the main part of the **CREATE TABLE** statement. As with table partitioned by **RANGE**, you can use **MAXVALUE** to represent a value such that any legal value inserted into a given column is always less than this value. Here is an example of a **CREATE TABLE** statement that helps to illustrate all of these points:

```
mysql> CREATE TABLE rcx (
->     a INT,
->     b INT,
->     c CHAR(3),
->     d INT
-> )
-> PARTITION BY RANGE COLUMNS(a,d,c) (
->     PARTITION p0 VALUES LESS THAN (5,10,'ggg'),
->     PARTITION p1 VALUES LESS THAN (10,20,'mmm'),
->     PARTITION p2 VALUES LESS THAN (15,30,'sss'),
->     PARTITION p3 VALUES LESS THAN (MAXVALUE,MAXVALUE,MAXVALUE)
-> );
Query OK, 0 rows affected (0.15 sec)
```

Table **rcx** contains the columns **a**, **b**, **c**, **d**. The partitioning column list supplied to the **COLUMNS** clause uses 3 of these columns, in the order **a**, **d**, **c**. Each value list used to define a partition contains 3 values in the same order; that is, each value list tuple has the form (**INT**, **INT**, **CHAR(3)**), which corresponds to the data types used by columns **a**, **d**, and **c** (in that order).

Placement of rows into partitions is determined by comparing the tuple from a row to be inserted that matches the column list in the **COLUMNS** clause with the tuples used in the **VALUES LESS THAN**

clauses to define partitions of the table. Because we are comparing tuples (that is, lists or sets of values) rather than scalar values, the semantics of **VALUES LESS THAN** as used with **RANGE COLUMNS** partitions differs somewhat from the case with simple **RANGE** partitions. In **RANGE** partitioning, a row generating an expression value that is equal to a limiting value in a **VALUES LESS THAN** is never placed in the corresponding partition; however, when using **RANGE COLUMNS** partitioning, it is sometimes possible for a row whose partitioning column list's first element is equal in value to the that of the first element in a **VALUES LESS THAN** value list to be placed in the corresponding partition.

Consider the **RANGE** partitioned table created by this statement:

```
CREATE TABLE r1 (
  a INT,
  b INT
)
PARTITION BY RANGE (a) (
  PARTITION p0 VALUES LESS THAN (5),
  PARTITION p1 VALUES LESS THAN (MAXVALUE)
);
```

If we insert 3 rows into this table such that the column value for **a** is 5 for each row, all 3 rows are stored in partition **p1** because the **a** column value is in each case not less than 5, as we can see by executing the proper query against the **INFORMATION_SCHEMA.PARTITIONS** table:

```
mysql> INSERT INTO r1 VALUES (5,10), (5,11), (5,12);
Query OK, 3 rows affected (0.00 sec)
Records: 3 Duplicates: 0 Warnings: 0
mysql> SELECT PARTITION_NAME, TABLE_ROWS
-> FROM INFORMATION_SCHEMA.PARTITIONS
-> WHERE TABLE_NAME = 'r1';
+-----+-----+
| PARTITION_NAME | TABLE_ROWS |
+-----+-----+
| p0             |            0 |
| p1             |            3 |
+-----+-----+
2 rows in set (0.00 sec)
```

Now consider a similar table **rc1** that uses **RANGE COLUMNS** partitioning with both columns **a** and **b** referenced in the **COLUMNS** clause, created as shown here:

```
CREATE TABLE rc1 (
  a INT,
  b INT
)
PARTITION BY RANGE COLUMNS(a, b) (
  PARTITION p0 VALUES LESS THAN (5, 12),
  PARTITION p3 VALUES LESS THAN (MAXVALUE, MAXVALUE)
);
```

If we insert exactly the same rows into **rc1** as we just inserted into **r1**, the distribution of the rows is quite different:

```
mysql> INSERT INTO rc1 VALUES (5,10), (5,11), (5,12);
Query OK, 3 rows affected (0.00 sec)
Records: 3 Duplicates: 0 Warnings: 0
mysql> SELECT PARTITION_NAME, TABLE_ROWS
-> FROM INFORMATION_SCHEMA.PARTITIONS
-> WHERE TABLE_NAME = 'rc1';
+-----+-----+-----+
| TABLE_SCHEMA | PARTITION_NAME | TABLE_ROWS |
+-----+-----+-----+
| p             | p0             |            2 |
| p             | p1             |            1 |
+-----+-----+-----+
2 rows in set (0.00 sec)
```

This is because we are comparing rows rather than scalar values. We can compare the row values inserted with the limiting row value from the `VALUES THAN LESS THAN` clause used to define partition `p0` in table `rc1`, like this:

```
mysql> SELECT (5,10) < (5,12), (5,11) < (5,12), (5,12) < (5,12);
+-----+-----+-----+
| (5,10) < (5,12) | (5,11) < (5,12) | (5,12) < (5,12) |
+-----+-----+-----+
| 1 | 1 | 0 |
+-----+-----+-----+
1 row in set (0.00 sec)
```

The 2 tuples `(5,10)` and `(5,11)` evaluate as less than `(5,12)`, so they are stored in partition `p0`. Since 5 is not less than 5 and 12 is not less than 12, `(5,12)` is considered not less than `(5,12)`, and is stored in partition `p1`.

The `SELECT` statement in the preceding example could also have been written using explicit row constructors, like this:

```
SELECT ROW(5,10) < ROW(5,12), ROW(5,11) < ROW(5,12), ROW(5,12) < ROW(5,12);
```

For more information about the use of row constructors in MySQL, see [Row Subqueries](#).

For a table partitioned by `RANGE COLUMNS` using only a single partitioning column, the storing of rows in partitions is the same as that of an equivalent table that is partitioned by `RANGE`. The following `CREATE TABLE` statement creates a table partitioned by `RANGE COLUMNS` using 1 partitioning column:

```
CREATE TABLE rx (
  a INT,
  b INT
)
PARTITION BY RANGE COLUMNS (a) (
  PARTITION p0 VALUES LESS THAN (5),
  PARTITION p1 VALUES LESS THAN (MAXVALUE)
);
```

If we insert the rows `(5,10)`, `(5,11)`, and `(5,12)` into this table, we can see that their placement is the same as it is for the table `r` we created and populated earlier:

```
mysql> INSERT INTO rx VALUES (5,10), (5,11), (5,12);
Query OK, 3 rows affected (0.00 sec)
Records: 3 Duplicates: 0 Warnings: 0
mysql> SELECT PARTITION_NAME, TABLE_ROWS
-> FROM INFORMATION_SCHEMA.PARTITIONS
-> WHERE TABLE_NAME = 'rx';
+-----+-----+-----+
| TABLE_SCHEMA | PARTITION_NAME | TABLE_ROWS |
+-----+-----+-----+
| p | p0 | 0 |
| p | p1 | 3 |
+-----+-----+-----+
2 rows in set (0.00 sec)
```

It is also possible to create tables partitioned by `RANGE COLUMNS` where limiting values for one or more columns are repeated in successive partition definitions. You can do this as long as the tuples of column values used to define the partitions are strictly increasing. For example, each of the following `CREATE TABLE` statements is valid:

```
CREATE TABLE rc2 (
  a INT,
  b INT
)
PARTITION BY RANGE COLUMNS(a,b) (
  PARTITION p0 VALUES LESS THAN (0,10),
```

```

PARTITION p1 VALUES LESS THAN (10,20),
PARTITION p2 VALUES LESS THAN (10,30),
PARTITION p3 VALUES LESS THAN (MAXVALUE,MAXVALUE)
);

CREATE TABLE rc3 (
  a INT,
  b INT
)
PARTITION BY RANGE COLUMNS(a,b) (
  PARTITION p0 VALUES LESS THAN (0,10),
  PARTITION p1 VALUES LESS THAN (10,20),
  PARTITION p2 VALUES LESS THAN (10,30),
  PARTITION p3 VALUES LESS THAN (10,35),
  PARTITION p4 VALUES LESS THAN (20,40),
  PARTITION p5 VALUES LESS THAN (MAXVALUE,MAXVALUE)
);

```

The following statement also succeeds, even though it might appear at first glance that it would not, since the limiting value of column **b** is 25 for partition **p0** and 20 for partition **p1**, and the limiting value of column **c** is 100 for partition **p1** and 50 for partition **p2**:

```

CREATE TABLE rc4 (
  a INT,
  b INT,
  c INT
)
PARTITION BY RANGE COLUMNS(a,b,c) (
  PARTITION p0 VALUES LESS THAN (0,25,50),
  PARTITION p1 VALUES LESS THAN (10,20,100),
  PARTITION p2 VALUES LESS THAN (10,30,50)
  PARTITION p3 VALUES LESS THAN (MAXVALUE,MAXVALUE,MAXVALUE)
);

```

When designing tables partitioned by **RANGE COLUMNS**, you can always test successive partition definitions by comparing the desired tuples using the **mysql** client, like this:

```

mysql> SELECT (0,25,50) < (10,20,100), (10,20,100) < (10,30,50);
+-----+-----+
| (0,25,50) < (10,20,100) | (10,20,100) < (10,30,50) |
+-----+-----+
| 1 | 1 |
+-----+-----+
1 row in set (0.00 sec)

```

If a **CREATE TABLE** statement contains partition definitions that are not in strictly increasing order, it fails with an error, as shown in this example:

```

mysql> CREATE TABLE rcf (
->   a INT,
->   b INT,
->   c INT
-> )
-> PARTITION BY RANGE COLUMNS(a,b,c) (
->   PARTITION p0 VALUES LESS THAN (0,25,50),
->   PARTITION p1 VALUES LESS THAN (20,20,100),
->   PARTITION p2 VALUES LESS THAN (10,30,50),
->   PARTITION p3 VALUES LESS THAN (MAXVALUE,MAXVALUE,MAXVALUE)
-> );
ERROR 1493 (HY000): VALUES LESS THAN value must be strictly increasing for each partition

```

When you get such an error, you can deduce which partition definitions are invalid by making “less than” comparisons between their column lists. In this case, the problem is with the definition of partition **p2** because the tuple used to define it is not less than the tuple used to define partition **p3**, as shown here:

```
mysql> SELECT (0,25,50) < (20,20,100), (20,20,100) < (10,30,50);
+-----+-----+
| (0,25,50) < (20,20,100) | (20,20,100) < (10,30,50) |
+-----+-----+
| 1 | 0 |
+-----+-----+
1 row in set (0.00 sec)
```

It is also possible for **MAXVALUE** to appear for the same column in more than one **VALUES LESS THAN** clause when using **RANGE COLUMNS**. However, the limiting values for individual columns in successive partition definitions should otherwise be increasing, there should be no more than one partition defined where **MAXVALUE** is used as the upper limit for all column values, and this partition definition should appear last in the list of **PARTITION ... VALUES LESS THAN** clauses. In addition, you cannot use **MAXVALUE** as the limiting value for the first column in more than one partition definition.

As stated previously, it is also possible with **RANGE COLUMNS** partitioning to use non-integer columns as partitioning columns. (See [Section 3.3, "COLUMNS Partitioning"](#), for a complete listing of these.) Consider a table named **employees** (which is not partitioned), created using the following statement:

```
CREATE TABLE employees (
  id INT NOT NULL,
  fname VARCHAR(30),
  lname VARCHAR(30),
  hired DATE NOT NULL DEFAULT '1970-01-01',
  separated DATE NOT NULL DEFAULT '9999-12-31',
  job_code INT NOT NULL,
  store_id INT NOT NULL
);
```

Using **RANGE COLUMNS** partitioning, you can create a version of this table that stores each row in one of four partitions based on the employee's last name, like this:

```
CREATE TABLE employees_by_lname (
  id INT NOT NULL,
  fname VARCHAR(30),
  lname VARCHAR(30),
  hired DATE NOT NULL DEFAULT '1970-01-01',
  separated DATE NOT NULL DEFAULT '9999-12-31',
  job_code INT NOT NULL,
  store_id INT NOT NULL
)
PARTITION BY RANGE COLUMNS (lname) (
  PARTITION p0 VALUES LESS THAN ('g'),
  PARTITION p1 VALUES LESS THAN ('m'),
  PARTITION p2 VALUES LESS THAN ('t'),
  PARTITION p3 VALUES LESS THAN (MAXVALUE)
);
```

Alternatively, you could cause the **employees** table as created previously to be partitioned using this scheme by executing the following **ALTER TABLE** statement:

```
ALTER TABLE employees PARTITION BY RANGE COLUMNS (lname) (
  PARTITION p0 VALUES LESS THAN ('g'),
  PARTITION p1 VALUES LESS THAN ('m'),
  PARTITION p2 VALUES LESS THAN ('t'),
  PARTITION p3 VALUES LESS THAN (MAXVALUE)
);
```

Note

Because different character sets and collations have different sort orders, the character sets and collations in use may effect which partition of a table partitioned by **RANGE COLUMNS** a given row is stored in when using string columns as partitioning columns. In addition, changing the character set or collation for a given database, table, or column after such a table is created may

cause changes in how rows are distributed. For example, when using a case-sensitive collation, 'and' sorts before 'Andersen', but when using a collation that is case insensitive, the reverse is true.

For information about how MySQL handles character sets and collations, see [Character Set Support](#).

Similarly, you can cause the `employees` table to be partitioned in such a way that each row is stored in one of several partitions based on the decade in which the corresponding employee was hired using the `ALTER TABLE` statement shown here:

```
ALTER TABLE employees PARTITION BY RANGE COLUMNS (hired) (
  PARTITION p0 VALUES LESS THAN ('1970-01-01'),
  PARTITION p1 VALUES LESS THAN ('1980-01-01'),
  PARTITION p2 VALUES LESS THAN ('1990-01-01'),
  PARTITION p3 VALUES LESS THAN ('2000-01-01'),
  PARTITION p4 VALUES LESS THAN ('2010-01-01'),
  PARTITION p5 VALUES LESS THAN (MAXVALUE)
);
```

See [CREATE TABLE Syntax](#), for additional information about `PARTITION BY RANGE COLUMNS` syntax.

3.3.2 LIST COLUMNS partitioning

MySQL 5.6 provides support for `LIST COLUMNS` partitioning. This is a variant of `LIST` partitioning that enables the use of multiple columns as partition keys, and for columns of data types other than integer types to be used as partitioning columns; you can use string types, `DATE`, and `DATETIME` columns. (For more information about permitted data types for `COLUMNS` partitioning columns, see [Section 3.3, “COLUMNS Partitioning”](#).)

Suppose that you have a business that has customers in 12 cities which, for sales and marketing purposes, you organize into 4 regions of 3 cities each as shown in the following table:

Region	Cities
1	Oskarshamn, Högsby, Mönsterås
2	Vimmerby, Hultsfred, Västervik
3	Nässjö, Eksjö, Vetlanda
4	Uppvidinge, Alvesta, Växjö

With `LIST COLUMNS` partitioning, you can create a table for customer data that assigns a row to any of 4 partitions corresponding to these regions based on the name of the city where a customer resides, as shown here:

```
CREATE TABLE customers_1 (
  first_name VARCHAR(25),
  last_name VARCHAR(25),
  street_1 VARCHAR(30),
  street_2 VARCHAR(30),
  city VARCHAR(15),
  renewal DATE
)
PARTITION BY LIST COLUMNS(city) (
  PARTITION pRegion_1 VALUES IN('Oskarshamn', 'Högsby', 'Mönsterås'),
  PARTITION pRegion_2 VALUES IN('Vimmerby', 'Hultsfred', 'Västervik'),
  PARTITION pRegion_3 VALUES IN('Nässjö', 'Eksjö', 'Vetlanda'),
  PARTITION pRegion_4 VALUES IN('Uppvidinge', 'Alvesta', 'Växjö')
);
```

As with partitioning by `RANGE COLUMNS`, you do not need to use expressions in the `COLUMNS()` clause to convert column values into integers. (In fact, the use of expressions other than column names is not permitted with `COLUMNS()`.)

It is also possible to use [DATE](#) and [DATETIME](#) columns, as shown in the following example that uses the same name and columns as the [customers_1](#) table shown previously, but employs [LIST COLUMNS](#) partitioning based on the [renewal](#) column to store rows in one of 4 partitions depending on the week in February 2010 the customer's account is scheduled to renew:

```
CREATE TABLE customers_2 (
  first_name VARCHAR(25),
  last_name VARCHAR(25),
  street_1 VARCHAR(30),
  street_2 VARCHAR(30),
  city VARCHAR(15),
  renewal DATE
)
PARTITION BY LIST COLUMNS(renewal) (
  PARTITION pWeek_1 VALUES IN('2010-02-01', '2010-02-02', '2010-02-03',
    '2010-02-04', '2010-02-05', '2010-02-06', '2010-02-07'),
  PARTITION pWeek_2 VALUES IN('2010-02-08', '2010-02-09', '2010-02-10',
    '2010-02-11', '2010-02-12', '2010-02-13', '2010-02-14'),
  PARTITION pWeek_3 VALUES IN('2010-02-15', '2010-02-16', '2010-02-17',
    '2010-02-18', '2010-02-19', '2010-02-20', '2010-02-21'),
  PARTITION pWeek_4 VALUES IN('2010-02-22', '2010-02-23', '2010-02-24',
    '2010-02-25', '2010-02-26', '2010-02-27', '2010-02-28')
);
```

This works, but becomes cumbersome to define and maintain if the number of dates involved grows very large; in such cases, it is usually more practical to employ [RANGE](#) or [RANGE COLUMNS](#) partitioning instead. In this case, since the column we wish to use as the partitioning key is a [DATE](#) column, we use [RANGE COLUMNS](#) partitioning, as shown here:

```
CREATE TABLE customers_3 (
  first_name VARCHAR(25),
  last_name VARCHAR(25),
  street_1 VARCHAR(30),
  street_2 VARCHAR(30),
  city VARCHAR(15),
  renewal DATE
)
PARTITION BY RANGE COLUMNS(renewal) (
  PARTITION pWeek_1 VALUES LESS THAN('2010-02-09'),
  PARTITION pWeek_2 VALUES LESS THAN('2010-02-15'),
  PARTITION pWeek_3 VALUES LESS THAN('2010-02-22'),
  PARTITION pWeek_4 VALUES LESS THAN('2010-03-01')
);
```

See [Section 3.3.1, “RANGE COLUMNS partitioning”](#), for more information.

In addition (as with [RANGE COLUMNS](#) partitioning), you can use multiple columns in the [COLUMNS\(\)](#) clause.

See [CREATE TABLE Syntax](#), for additional information about [PARTITION BY LIST COLUMNS\(\)](#) syntax.

3.4 HASH Partitioning

Partitioning by [HASH](#) is used primarily to ensure an even distribution of data among a predetermined number of partitions. With range or list partitioning, you must specify explicitly into which partition a given column value or set of column values is to be stored; with hash partitioning, MySQL takes care of this for you, and you need only specify a column value or expression based on a column value to be hashed and the number of partitions into which the partitioned table is to be divided.

To partition a table using [HASH](#) partitioning, it is necessary to append to the [CREATE TABLE](#) statement a [PARTITION BY HASH \(expr\)](#) clause, where *expr* is an expression that returns an integer. This can simply be the name of a column whose type is one of MySQL's integer types. In addition, you

most likely want to follow this with `PARTITIONS num`, where *num* is a positive integer representing the number of partitions into which the table is to be divided.

Note

For simplicity, the tables in the examples that follow do not use any keys. You should be aware that, if a table has any unique keys, every column used in the partitioning expression for this table must be part of every unique key, including the primary key. See [Section 6.1, “Partitioning Keys, Primary Keys, and Unique Keys”](#), for more information.

The following statement creates a table that uses hashing on the `store_id` column and is divided into 4 partitions:

```
CREATE TABLE employees (
  id INT NOT NULL,
  fname VARCHAR(30),
  lname VARCHAR(30),
  hired DATE NOT NULL DEFAULT '1970-01-01',
  separated DATE NOT NULL DEFAULT '9999-12-31',
  job_code INT,
  store_id INT
)
PARTITION BY HASH(store_id)
PARTITIONS 4;
```

If you do not include a `PARTITIONS` clause, the number of partitions defaults to 1.

Using the `PARTITIONS` keyword without a number following it results in a syntax error.

You can also use an SQL expression that returns an integer for *expr*. For instance, you might want to partition based on the year in which an employee was hired. This can be done as shown here:

```
CREATE TABLE employees (
  id INT NOT NULL,
  fname VARCHAR(30),
  lname VARCHAR(30),
  hired DATE NOT NULL DEFAULT '1970-01-01',
  separated DATE NOT NULL DEFAULT '9999-12-31',
  job_code INT,
  store_id INT
)
PARTITION BY HASH( YEAR(hired) )
PARTITIONS 4;
```

expr must return a nonconstant, nonrandom integer value (in other words, it should be varying but deterministic), and must not contain any prohibited constructs as described in [Chapter 6, Restrictions and Limitations on Partitioning](#). You should also keep in mind that this expression is evaluated each time a row is inserted or updated (or possibly deleted); this means that very complex expressions may give rise to performance issues, particularly when performing operations (such as batch inserts) that affect a great many rows at one time.

The most efficient hashing function is one which operates upon a single table column and whose value increases or decreases consistently with the column value, as this allows for “pruning” on ranges of partitions. That is, the more closely that the expression varies with the value of the column on which it is based, the more efficiently MySQL can use the expression for hash partitioning.

For example, where `date_col` is a column of type `DATE`, then the expression `TO_DAYS(date_col)` is said to vary directly with the value of `date_col`, because for every change in the value of `date_col`, the value of the expression changes in a consistent manner. The variance of the expression `YEAR(date_col)` with respect to `date_col` is not quite as direct as that of `TO_DAYS(date_col)`, because not every possible change in `date_col` produces an equivalent change in `YEAR(date_col)`. Even so, `YEAR(date_col)` is a good candidate for a hashing function,

because it varies directly with a portion of `date_col` and there is no possible change in `date_col` that produces a disproportionate change in `YEAR(date_col)`.

By way of contrast, suppose that you have a column named `int_col` whose type is `INT`. Now consider the expression `POW(5-int_col,3) + 6`. This would be a poor choice for a hashing function because a change in the value of `int_col` is not guaranteed to produce a proportional change in the value of the expression. Changing the value of `int_col` by a given amount can produce by widely different changes in the value of the expression. For example, changing `int_col` from 5 to 6 produces a change of -1 in the value of the expression, but changing the value of `int_col` from 6 to 7 produces a change of -7 in the expression value.

In other words, the more closely the graph of the column value versus the value of the expression follows a straight line as traced by the equation $y=cx$ where c is some nonzero constant, the better the expression is suited to hashing. This has to do with the fact that the more nonlinear an expression is, the more uneven the distribution of data among the partitions it tends to produce.

In theory, pruning is also possible for expressions involving more than one column value, but determining which of such expressions are suitable can be quite difficult and time-consuming. For this reason, the use of hashing expressions involving multiple columns is not particularly recommended.

When `PARTITION BY HASH` is used, MySQL determines which partition of `num` partitions to use based on the modulus of the result of the user function. In other words, for an expression `expr`, the partition in which the record is stored is partition number N , where $N = \text{MOD}(\text{expr}, \text{num})$. Suppose that table `t1` is defined as follows, so that it has 4 partitions:

```
CREATE TABLE t1 (col1 INT, col2 CHAR(5), col3 DATE)
PARTITION BY HASH( YEAR(col3) )
PARTITIONS 4;
```

If you insert a record into `t1` whose `col3` value is `'2005-09-15'`, then the partition in which it is stored is determined as follows:

```
MOD(YEAR( '2005-09-01' ), 4)
= MOD(2005, 4)
= 1
```

MySQL 5.6 also supports a variant of `HASH` partitioning known as *linear hashing* which employs a more complex algorithm for determining the placement of new rows inserted into the partitioned table. See [Section 3.4.1, “LINEAR HASH Partitioning”](#), for a description of this algorithm.

The user function is evaluated each time a record is inserted or updated. It may also—depending on the circumstances—be evaluated when records are deleted.

Note

If a table to be partitioned has a `UNIQUE` key, then any columns supplied as arguments to the `HASH` user function or to the `KEY`'s `column_list` must be part of that key.

3.4.1 LINEAR HASH Partitioning

MySQL also supports linear hashing, which differs from regular hashing in that linear hashing utilizes a linear powers-of-two algorithm whereas regular hashing employs the modulus of the hashing function's value.

Syntactically, the only difference between linear-hash partitioning and regular hashing is the addition of the `LINEAR` keyword in the `PARTITION BY` clause, as shown here:

```
CREATE TABLE employees (
  id INT NOT NULL,
```

```

    fname VARCHAR(30),
    lname VARCHAR(30),
    hired DATE NOT NULL DEFAULT '1970-01-01',
    separated DATE NOT NULL DEFAULT '9999-12-31',
    job_code INT,
    store_id INT
)
PARTITION BY LINEAR HASH( YEAR(hired) )
PARTITIONS 4;

```

Given an expression *expr*, the partition in which the record is stored when linear hashing is used is partition number *N* from among *num* partitions, where *N* is derived according to the following algorithm:

1. Find the next power of 2 greater than *num*. We call this value *V*; it can be calculated as:

```
V = POWER(2, CEILING(LOG(2, num)))
```

(Suppose that *num* is 13. Then $\text{LOG}(2, 13)$ is 3.7004397181411. $\text{CEILING}(3.7004397181411)$ is 4, and $V = \text{POWER}(2, 4)$, which is 16.)

2. Set $N = F(\text{column_list}) \& (V - 1)$.
3. While $N \geq \text{num}$:
 - Set $V = \text{CEIL}(V / 2)$
 - Set $N = N \& (V - 1)$

Suppose that the table *t1*, using linear hash partitioning and having 6 partitions, is created using this statement:

```

CREATE TABLE t1 (col1 INT, col2 CHAR(5), col3 DATE)
PARTITION BY LINEAR HASH( YEAR(col3) )
PARTITIONS 6;

```

Now assume that you want to insert two records into *t1* having the *col3* column values '2003-04-14' and '1998-10-19'. The partition number for the first of these is determined as follows:

```

V = POWER(2, CEILING( LOG(2,6) )) = 8
N = YEAR('2003-04-14') & (8 - 1)
  = 2003 & 7
  = 3
(3 >= 6 is FALSE: record stored in partition #3)

```

The number of the partition where the second record is stored is calculated as shown here:

```

V = 8
N = YEAR('1998-10-19') & (8-1)
  = 1998 & 7
  = 6
(6 >= 6 is TRUE: additional step required)
N = 6 & CEILING(8 / 2)
  = 6 & 3
  = 2
(2 >= 6 is FALSE: record stored in partition #2)

```

The advantage in partitioning by linear hash is that the adding, dropping, merging, and splitting of partitions is made much faster, which can be beneficial when dealing with tables containing extremely large amounts (terabytes) of data. The disadvantage is that data is less likely to be evenly distributed between partitions as compared with the distribution obtained using regular hash partitioning.

3.5 KEY Partitioning

Partitioning by key is similar to partitioning by hash, except that where hash partitioning employs a user-defined expression, the hashing function for key partitioning is supplied by the MySQL server. MySQL Cluster uses `MD5()` for this purpose; for tables using other storage engines, the server employs its own internal hashing function which is based on the same algorithm as `PASSWORD()`.

The syntax rules for `CREATE TABLE ... PARTITION BY KEY` are similar to those for creating a table that is partitioned by hash. The major differences are listed here:

- `KEY` is used rather than `HASH`.
- `KEY` takes only a list of zero or more column names. Any columns used as the partitioning key must comprise part or all of the table's primary key, if the table has one. Where no column name is specified as the partitioning key, the table's primary key is used, if there is one. For example, the following `CREATE TABLE` statement is valid in MySQL 5.6:

```
CREATE TABLE k1 (  
    id INT NOT NULL PRIMARY KEY,  
    name VARCHAR(20)  
)  
PARTITION BY KEY()  
PARTITIONS 2;
```

If there is no primary key but there is a unique key, then the unique key is used for the partitioning key:

```
CREATE TABLE k1 (  
    id INT NOT NULL,  
    name VARCHAR(20),  
    UNIQUE KEY (id)  
)  
PARTITION BY KEY()  
PARTITIONS 2;
```

However, if the unique key column were not defined as `NOT NULL`, then the previous statement would fail.

In both of these cases, the partitioning key is the `id` column, even though it is not shown in the output of `SHOW CREATE TABLE` or in the `PARTITION_EXPRESSION` column of the `INFORMATION_SCHEMA.PARTITIONS` table.

Unlike the case with other partitioning types, columns used for partitioning by `KEY` are not restricted to integer or `NULL` values. For example, the following `CREATE TABLE` statement is valid:

```
CREATE TABLE tm1 (  
    s1 CHAR(32) PRIMARY KEY  
)  
PARTITION BY KEY(s1)  
PARTITIONS 10;
```

The preceding statement would *not* be valid, were a different partitioning type to be specified. (In this case, simply using `PARTITION BY KEY()` would also be valid and have the same effect as `PARTITION BY KEY(s1)`, since `s1` is the table's primary key.)

For additional information about this issue, see [Chapter 6, Restrictions and Limitations on Partitioning](#).

Note

Tables using the `NDB` storage engine are implicitly partitioned by `KEY`, again using the table's primary key as the partitioning key. In the event that the MySQL Cluster table has no explicit primary key, the “hidden” primary key

generated by the [NDB](#) storage engine for each MySQL Cluster table is used as the partitioning key.

If you define an explicit partitioning scheme for an [NDB](#) table, the table must have an explicit primary key, and any columns used in the partitioning expression must be part of this key. However, if the table uses an “empty” partitioning expression—that is, [PARTITION BY KEY\(\)](#) with no column references—then no explicit primary key is required.

You can observe this partitioning using the [ndb_desc](#) utility (with the [-p](#) option).

Important

For a key-partitioned table, you cannot execute an [ALTER TABLE DROP PRIMARY KEY](#), as doing so generates the error [ERROR 1466 \(HY000\): Field in list of fields for partition function not found in table](#). This is not an issue for MySQL Cluster tables which are partitioned by [KEY](#); in such cases, the table is reorganized using the “hidden” primary key as the table’s new partitioning key. See [MySQL Cluster NDB 7.3](#) and [MySQL Cluster NDB 7.4](#).

It is also possible to partition a table by linear key. Here is a simple example:

```
CREATE TABLE tk (
  col1 INT NOT NULL,
  col2 CHAR(5),
  col3 DATE
)
PARTITION BY LINEAR KEY (col1)
PARTITIONS 3;
```

Using [LINEAR](#) has the same effect on [KEY](#) partitioning as it does on [HASH](#) partitioning, with the partition number being derived using a powers-of-two algorithm rather than modulo arithmetic. See [Section 3.4.1, “LINEAR HASH Partitioning”](#), for a description of this algorithm and its implications.

3.6 Subpartitioning

Subpartitioning—also known as *composite partitioning*—is the further division of each partition in a partitioned table. Consider the following [CREATE TABLE](#) statement:

```
CREATE TABLE ts (id INT, purchased DATE)
PARTITION BY RANGE( YEAR(purchased) )
SUBPARTITION BY HASH( TO_DAYS(purchased) )
SUBPARTITIONS 2 (
  PARTITION p0 VALUES LESS THAN (1990),
  PARTITION p1 VALUES LESS THAN (2000),
  PARTITION p2 VALUES LESS THAN MAXVALUE
);
```

Table [ts](#) has 3 [RANGE](#) partitions. Each of these partitions—[p0](#), [p1](#), and [p2](#)—is further divided into 2 subpartitions. In effect, the entire table is divided into $3 * 2 = 6$ partitions. However, due to the action of the [PARTITION BY RANGE](#) clause, the first 2 of these store only those records with a value less than 1990 in the [purchased](#) column.

In MySQL 5.6, it is possible to subpartition tables that are partitioned by [RANGE](#) or [LIST](#). Subpartitions may use either [HASH](#) or [KEY](#) partitioning. This is also known as *composite partitioning*.

Note

[SUBPARTITION BY HASH](#) and [SUBPARTITION BY KEY](#) generally follow the same syntax rules as [PARTITION BY HASH](#) and [PARTITION BY KEY](#),

respectively. An exception to this is that **SUBPARTITION BY KEY** (unlike **PARTITION BY KEY**) does not currently support a default column, so the column used for this purpose must be specified, even if the table has an explicit primary key. This is a known issue which we are working to address; see [Issues with subpartitions](#), for more information and an example.

It is also possible to define subpartitions explicitly using **SUBPARTITION** clauses to specify options for individual subpartitions. For example, a more verbose fashion of creating the same table **ts** as shown in the previous example would be:

```
CREATE TABLE ts (id INT, purchased DATE)
PARTITION BY RANGE( YEAR(purchased) )
SUBPARTITION BY HASH( TO_DAYS(purchased) ) (
  PARTITION p0 VALUES LESS THAN (1990) (
    SUBPARTITION s0,
    SUBPARTITION s1
  ),
  PARTITION p1 VALUES LESS THAN (2000) (
    SUBPARTITION s2,
    SUBPARTITION s3
  ),
  PARTITION p2 VALUES LESS THAN MAXVALUE (
    SUBPARTITION s4,
    SUBPARTITION s5
  )
);
```

Some syntactical items of note are listed here:

- Each partition must have the same number of subpartitions.
- If you explicitly define any subpartitions using **SUBPARTITION** on any partition of a partitioned table, you must define them all. In other words, the following statement will fail:

```
CREATE TABLE ts (id INT, purchased DATE)
PARTITION BY RANGE( YEAR(purchased) )
SUBPARTITION BY HASH( TO_DAYS(purchased) ) (
  PARTITION p0 VALUES LESS THAN (1990) (
    SUBPARTITION s0,
    SUBPARTITION s1
  ),
  PARTITION p1 VALUES LESS THAN (2000),
  PARTITION p2 VALUES LESS THAN MAXVALUE (
    SUBPARTITION s2,
    SUBPARTITION s3
  )
);
```

This statement would still fail even if it included a **SUBPARTITIONS 2** clause.

- Each **SUBPARTITION** clause must include (at a minimum) a name for the subpartition. Otherwise, you may set any desired option for the subpartition or allow it to assume its default setting for that option.
- Subpartition names must be unique across the entire table. For example, the following **CREATE TABLE** statement is valid in MySQL 5.6:

```
CREATE TABLE ts (id INT, purchased DATE)
PARTITION BY RANGE( YEAR(purchased) )
SUBPARTITION BY HASH( TO_DAYS(purchased) ) (
  PARTITION p0 VALUES LESS THAN (1990) (
    SUBPARTITION s0,
    SUBPARTITION s1
  ),
```

```

PARTITION p1 VALUES LESS THAN (2000) (
    SUBPARTITION s2,
    SUBPARTITION s3
),
PARTITION p2 VALUES LESS THAN MAXVALUE (
    SUBPARTITION s4,
    SUBPARTITION s5
)
);

```

Subpartitions can be used with especially large tables to distribute data and indexes across many disks. Suppose that you have 6 disks mounted as `/disk0`, `/disk1`, `/disk2`, and so on. Now consider the following example:

```

CREATE TABLE ts (id INT, purchased DATE)
PARTITION BY RANGE( YEAR(purchased) )
SUBPARTITION BY HASH( TO_DAYS(purchased) ) (
    PARTITION p0 VALUES LESS THAN (1990) (
        SUBPARTITION s0
            DATA DIRECTORY = '/disk0/data'
            INDEX DIRECTORY = '/disk0/idx',
        SUBPARTITION s1
            DATA DIRECTORY = '/disk1/data'
            INDEX DIRECTORY = '/disk1/idx'
    ),
    PARTITION p1 VALUES LESS THAN (2000) (
        SUBPARTITION s2
            DATA DIRECTORY = '/disk2/data'
            INDEX DIRECTORY = '/disk2/idx',
        SUBPARTITION s3
            DATA DIRECTORY = '/disk3/data'
            INDEX DIRECTORY = '/disk3/idx'
    ),
    PARTITION p2 VALUES LESS THAN MAXVALUE (
        SUBPARTITION s4
            DATA DIRECTORY = '/disk4/data'
            INDEX DIRECTORY = '/disk4/idx',
        SUBPARTITION s5
            DATA DIRECTORY = '/disk5/data'
            INDEX DIRECTORY = '/disk5/idx'
    )
);

```

In this case, a separate disk is used for the data and for the indexes of each `RANGE`. Many other variations are possible; another example might be:

```

CREATE TABLE ts (id INT, purchased DATE)
PARTITION BY RANGE(YEAR(purchased))
SUBPARTITION BY HASH( TO_DAYS(purchased) ) (
    PARTITION p0 VALUES LESS THAN (1990) (
        SUBPARTITION s0a
            DATA DIRECTORY = '/disk0'
            INDEX DIRECTORY = '/disk1',
        SUBPARTITION s0b
            DATA DIRECTORY = '/disk2'
            INDEX DIRECTORY = '/disk3'
    ),
    PARTITION p1 VALUES LESS THAN (2000) (
        SUBPARTITION s1a
            DATA DIRECTORY = '/disk4/data'
            INDEX DIRECTORY = '/disk4/idx',
        SUBPARTITION s1b
            DATA DIRECTORY = '/disk5/data'
            INDEX DIRECTORY = '/disk5/idx'
    ),
    PARTITION p2 VALUES LESS THAN MAXVALUE (
        SUBPARTITION s2a,
        SUBPARTITION s2b
    )
);

```

```
);
```

Here, the storage is as follows:

- Rows with `purchased` dates from before 1990 take up a vast amount of space, so are split up 4 ways, with a separate disk dedicated to the data and to the indexes for each of the two subpartitions (`s0a` and `s0b`) making up partition `p0`. In other words:
 - The data for subpartition `s0a` is stored on `/disk0`.
 - The indexes for subpartition `s0a` are stored on `/disk1`.
 - The data for subpartition `s0b` is stored on `/disk2`.
 - The indexes for subpartition `s0b` are stored on `/disk3`.
- Rows containing dates ranging from 1990 to 1999 (partition `p1`) do not require as much room as those from before 1990. These are split between 2 disks (`/disk4` and `/disk5`) rather than 4 disks as with the legacy records stored in `p0`:
 - Data and indexes belonging to `p1`'s first subpartition (`s1a`) are stored on `/disk4`—the data in `/disk4/data`, and the indexes in `/disk4/idx`.
 - Data and indexes belonging to `p1`'s second subpartition (`s1b`) are stored on `/disk5`—the data in `/disk5/data`, and the indexes in `/disk5/idx`.
- Rows reflecting dates from the year 2000 to the present (partition `p2`) do not take up as much space as required by either of the two previous ranges. Currently, it is sufficient to store all of these in the default location.

In future, when the number of purchases for the decade beginning with the year 2000 grows to a point where the default location no longer provides sufficient space, the corresponding rows can be moved using an `ALTER TABLE ... REORGANIZE PARTITION` statement. See [Chapter 4, Partition Management](#), for an explanation of how this can be done.

The `DATA DIRECTORY` and `INDEX DIRECTORY` options are not permitted in partition definitions when the `NO_DIR_IN_CREATE` server SQL mode is in effect. In MySQL 5.6, these options are also not permitted when defining subpartitions (Bug #42954).

3.7 How MySQL Partitioning Handles NULL

Partitioning in MySQL does nothing to disallow `NULL` as the value of a partitioning expression, whether it is a column value or the value of a user-supplied expression. Even though it is permitted to use `NULL` as the value of an expression that must otherwise yield an integer, it is important to keep in mind that `NULL` is not a number. MySQL's partitioning implementation treats `NULL` as being less than any non-`NULL` value, just as `ORDER BY` does.

This means that treatment of `NULL` varies between partitioning of different types, and may produce behavior which you do not expect if you are not prepared for it. This being the case, we discuss in this section how each MySQL partitioning type handles `NULL` values when determining the partition in which a row should be stored, and provide examples for each.

Handling of `NULL` with `RANGE` partitioning. If you insert a row into a table partitioned by `RANGE` such that the column value used to determine the partition is `NULL`, the row is inserted into the lowest partition. Consider these two tables in a database named `p`, created as follows:

```
mysql> CREATE TABLE t1 (
->     c1 INT,
->     c2 VARCHAR(20)
-> )
-> PARTITION BY RANGE(c1) (
```

```

-> PARTITION p0 VALUES LESS THAN (0),
-> PARTITION p1 VALUES LESS THAN (10),
-> PARTITION p2 VALUES LESS THAN MAXVALUE
-> );
Query OK, 0 rows affected (0.09 sec)
mysql> CREATE TABLE t2 (
->   c1 INT,
->   c2 VARCHAR(20)
-> )
-> PARTITION BY RANGE(c1) (
->   PARTITION p0 VALUES LESS THAN (-5),
->   PARTITION p1 VALUES LESS THAN (0),
->   PARTITION p2 VALUES LESS THAN (10),
->   PARTITION p3 VALUES LESS THAN MAXVALUE
-> );
Query OK, 0 rows affected (0.09 sec)

```

You can see the partitions created by these two `CREATE TABLE` statements using the following query against the `PARTITIONS` table in the `INFORMATION_SCHEMA` database:

```

mysql> SELECT TABLE_NAME, PARTITION_NAME, TABLE_ROWS, AVG_ROW_LENGTH, DATA_LENGTH
> FROM INFORMATION_SCHEMA.PARTITIONS
> WHERE TABLE_SCHEMA = 'p' AND TABLE_NAME LIKE 't_';
+-----+-----+-----+-----+-----+
| TABLE_NAME | PARTITION_NAME | TABLE_ROWS | AVG_ROW_LENGTH | DATA_LENGTH |
+-----+-----+-----+-----+-----+
| t1          | p0              | 0           | 0              | 0            |
| t1          | p1              | 0           | 0              | 0            |
| t1          | p2              | 0           | 0              | 0            |
| t2          | p0              | 0           | 0              | 0            |
| t2          | p1              | 0           | 0              | 0            |
| t2          | p2              | 0           | 0              | 0            |
| t2          | p3              | 0           | 0              | 0            |
+-----+-----+-----+-----+-----+
7 rows in set (0.00 sec)

```

(For more information about this table, see [The INFORMATION_SCHEMA PARTITIONS Table](#).) Now let us populate each of these tables with a single row containing a `NULL` in the column used as the partitioning key, and verify that the rows were inserted using a pair of `SELECT` statements:

```

mysql> INSERT INTO t1 VALUES (NULL, 'mothra');
Query OK, 1 row affected (0.00 sec)
mysql> INSERT INTO t2 VALUES (NULL, 'mothra');
Query OK, 1 row affected (0.00 sec)
mysql> SELECT * FROM t1;
+-----+-----+
| id  | name  |
+-----+-----+
| NULL | mothra |
+-----+-----+
1 row in set (0.00 sec)
mysql> SELECT * FROM t2;
+-----+-----+
| id  | name  |
+-----+-----+
| NULL | mothra |
+-----+-----+
1 row in set (0.00 sec)

```

You can see which partitions are used to store the inserted rows by rerunning the previous query against `INFORMATION_SCHEMA.PARTITIONS` and inspecting the output:

```

mysql> SELECT TABLE_NAME, PARTITION_NAME, TABLE_ROWS, AVG_ROW_LENGTH, DATA_LENGTH
> FROM INFORMATION_SCHEMA.PARTITIONS
> WHERE TABLE_SCHEMA = 'p' AND TABLE_NAME LIKE 't_';
+-----+-----+-----+-----+-----+
| TABLE_NAME | PARTITION_NAME | TABLE_ROWS | AVG_ROW_LENGTH | DATA_LENGTH |
+-----+-----+-----+-----+-----+

```

t1	p0	1	20	20
t1	p1	0	0	0
t1	p2	0	0	0
t2	p0	1	20	20
t2	p1	0	0	0
t2	p2	0	0	0
t2	p3	0	0	0

7 rows in set (0.01 sec)

You can also demonstrate that these rows were stored in the lowest partition of each table by dropping these partitions, and then re-running the [SELECT](#) statements:

```
mysql> ALTER TABLE t1 DROP PARTITION p0;
Query OK, 0 rows affected (0.16 sec)
mysql> ALTER TABLE t2 DROP PARTITION p0;
Query OK, 0 rows affected (0.16 sec)
mysql> SELECT * FROM t1;
Empty set (0.00 sec)
mysql> SELECT * FROM t2;
Empty set (0.00 sec)
```

(For more information on [ALTER TABLE ... DROP PARTITION](#), see [ALTER TABLE Syntax](#).)

NULL is also treated in this way for partitioning expressions that use SQL functions. Suppose that we define a table using a [CREATE TABLE](#) statement such as this one:

```
CREATE TABLE tndate (
  id INT,
  dt DATE
)
PARTITION BY RANGE( YEAR(dt) ) (
  PARTITION p0 VALUES LESS THAN (1990),
  PARTITION p1 VALUES LESS THAN (2000),
  PARTITION p2 VALUES LESS THAN MAXVALUE
);
```

As with other MySQL functions, [YEAR\(NULL\)](#) returns **NULL**. A row with a **dt** column value of **NULL** is treated as though the partitioning expression evaluated to a value less than any other value, and so is inserted into partition **p0**.

Handling of NULL with LIST partitioning. A table that is partitioned by [LIST](#) admits **NULL** values if and only if one of its partitions is defined using that value-list that contains **NULL**. The converse of this is that a table partitioned by [LIST](#) which does not explicitly use **NULL** in a value list rejects rows resulting in a **NULL** value for the partitioning expression, as shown in this example:

```
mysql> CREATE TABLE ts1 (
->   c1 INT,
->   c2 VARCHAR(20)
-> )
-> PARTITION BY LIST(c1) (
->   PARTITION p0 VALUES IN (0, 3, 6),
->   PARTITION p1 VALUES IN (1, 4, 7),
->   PARTITION p2 VALUES IN (2, 5, 8)
-> );
Query OK, 0 rows affected (0.01 sec)
mysql> INSERT INTO ts1 VALUES (9, 'mothra');
ERROR 1504 (HY000): Table has no partition for value 9
mysql> INSERT INTO ts1 VALUES (NULL, 'mothra');
ERROR 1504 (HY000): Table has no partition for value NULL
```

Only rows having a **c1** value between 0 and 8 inclusive can be inserted into **ts1**. **NULL** falls outside this range, just like the number 9. We can create tables **ts2** and **ts3** having value lists containing **NULL**, as shown here:

```
mysql> CREATE TABLE ts2 (
->   c1 INT,
->   c2 VARCHAR(20)
-> )
-> PARTITION BY LIST(c1) (
->   PARTITION p0 VALUES IN (0, 3, 6),
->   PARTITION p1 VALUES IN (1, 4, 7),
->   PARTITION p2 VALUES IN (2, 5, 8),
->   PARTITION p3 VALUES IN (NULL)
-> );
Query OK, 0 rows affected (0.01 sec)
mysql> CREATE TABLE ts3 (
->   c1 INT,
->   c2 VARCHAR(20)
-> )
-> PARTITION BY LIST(c1) (
->   PARTITION p0 VALUES IN (0, 3, 6),
->   PARTITION p1 VALUES IN (1, 4, 7, NULL),
->   PARTITION p2 VALUES IN (2, 5, 8)
-> );
Query OK, 0 rows affected (0.01 sec)
```

When defining value lists for partitioning, you can (and should) treat **NULL** just as you would any other value. For example, both **VALUES IN (NULL)** and **VALUES IN (1, 4, 7, NULL)** are valid, as are **VALUES IN (1, NULL, 4, 7)**, **VALUES IN (NULL, 1, 4, 7)**, and so on. You can insert a row having **NULL** for column **c1** into each of the tables **ts2** and **ts3**:

```
mysql> INSERT INTO ts2 VALUES (NULL, 'mothra');
Query OK, 1 row affected (0.00 sec)
mysql> INSERT INTO ts3 VALUES (NULL, 'mothra');
Query OK, 1 row affected (0.00 sec)
```

By issuing the appropriate query against **INFORMATION_SCHEMA.PARTITIONS**, you can determine which partitions were used to store the rows just inserted (we assume, as in the previous examples, that the partitioned tables were created in the **p** database):

```
mysql> SELECT TABLE_NAME, PARTITION_NAME, TABLE_ROWS, AVG_ROW_LENGTH, DATA_LENGTH
> FROM INFORMATION_SCHEMA.PARTITIONS
> WHERE TABLE_SCHEMA = 'p' AND TABLE_NAME LIKE 'ts_';
+-----+-----+-----+-----+-----+
| TABLE_NAME | PARTITION_NAME | TABLE_ROWS | AVG_ROW_LENGTH | DATA_LENGTH |
+-----+-----+-----+-----+-----+
| ts2         | p0              | 0           | 0              | 0            |
| ts2         | p1              | 0           | 0              | 0            |
| ts2         | p2              | 0           | 0              | 0            |
| ts2         | p3              | 1           | 20             | 20           |
| ts3         | p0              | 0           | 0              | 0            |
| ts3         | p1              | 1           | 20             | 20           |
| ts3         | p2              | 0           | 0              | 0            |
+-----+-----+-----+-----+-----+
7 rows in set (0.01 sec)
```

As shown earlier in this section, you can also verify which partitions were used for storing the rows by deleting these partitions and then performing a **SELECT**.

Handling of NULL with HASH and KEY partitioning. **NULL** is handled somewhat differently for tables partitioned by **HASH** or **KEY**. In these cases, any partition expression that yields a **NULL** value is treated as though its return value were zero. We can verify this behavior by examining the effects on the file system of creating a table partitioned by **HASH** and populating it with a record containing appropriate values. Suppose that you have a table **th** (also in the **p** database) created using the following statement:

```
mysql> CREATE TABLE th (
->   c1 INT,
->   c2 VARCHAR(20)
-> )
```

```
-> PARTITION BY HASH(c1)
-> PARTITIONS 2;
Query OK, 0 rows affected (0.00 sec)
```

The partitions belonging to this table can be viewed using the query shown here:

```
mysql> SELECT TABLE_NAME, PARTITION_NAME, TABLE_ROWS, AVG_ROW_LENGTH, DATA_LENGTH
> FROM INFORMATION_SCHEMA.PARTITIONS
> WHERE TABLE_SCHEMA = 'p' AND TABLE_NAME = 'th';
```

TABLE_NAME	PARTITION_NAME	TABLE_ROWS	AVG_ROW_LENGTH	DATA_LENGTH
th	p0	0	0	0
th	p1	0	0	0

```
2 rows in set (0.00 sec)
```

Notice that `TABLE_ROWS` for each partition is 0. Now insert two rows into `th` whose `c1` column values are `NULL` and 0, and verify that these rows were inserted, as shown here:

```
mysql> INSERT INTO th VALUES (NULL, 'mothra'), (0, 'gigan');
Query OK, 1 row affected (0.00 sec)
mysql> SELECT * FROM th;
```

c1	c2
NULL	mothra
0	gigan

```
2 rows in set (0.01 sec)
```

Recall that for any integer `N`, the value of `NULL MOD N` is always `NULL`. For tables that are partitioned by `HASH` or `KEY`, this result is treated for determining the correct partition as `0`. Checking the `INFORMATION_SCHEMA.PARTITIONS` table once again, we can see that both rows were inserted into partition `p0`:

```
mysql> SELECT TABLE_NAME, PARTITION_NAME, TABLE_ROWS, AVG_ROW_LENGTH, DATA_LENGTH
> FROM INFORMATION_SCHEMA.PARTITIONS
> WHERE TABLE_SCHEMA = 'p' AND TABLE_NAME = 'th';
```

TABLE_NAME	PARTITION_NAME	TABLE_ROWS	AVG_ROW_LENGTH	DATA_LENGTH
th	p0	2	20	20
th	p1	0	0	0

```
2 rows in set (0.00 sec)
```

If you repeat this example using `PARTITION BY KEY` in place of `PARTITION BY HASH` in the definition of the table, you can verify easily that `NULL` is also treated like 0 for this type of partitioning.

Chapter 4 Partition Management

Table of Contents

4.1 Management of RANGE and LIST Partitions	36
4.2 Management of HASH and KEY Partitions	41
4.3 Exchanging Partitions and Subpartitions with Tables	42
4.4 Maintenance of Partitions	47
4.5 Obtaining Information About Partitions	49

MySQL 5.6 provides a number of ways to modify partitioned tables. It is possible to add, drop, redefine, merge, or split existing partitions. All of these actions can be carried out using the partitioning extensions to the `ALTER TABLE` statement. There are also ways to obtain information about partitioned tables and partitions. We discuss these topics in the sections that follow.

- For information about partition management in tables partitioned by `RANGE` or `LIST`, see [Section 4.1, “Management of RANGE and LIST Partitions”](#).
- For a discussion of managing `HASH` and `KEY` partitions, see [Section 4.2, “Management of HASH and KEY Partitions”](#).
- See [Section 4.5, “Obtaining Information About Partitions”](#), for a discussion of mechanisms provided in MySQL 5.6 for obtaining information about partitioned tables and partitions.
- For a discussion of performing maintenance operations on partitions, see [Section 4.4, “Maintenance of Partitions”](#).

Note

In MySQL 5.6, all partitions of a partitioned table must have the same number of subpartitions, and it is not possible to change the subpartitioning once the table has been created.

To change a table's partitioning scheme, it is necessary only to use the `ALTER TABLE` statement with a `partition_options` clause. This clause has the same syntax as that as used with `CREATE TABLE` for creating a partitioned table, and always begins with the keywords `PARTITION BY`. Suppose that you have a table partitioned by range using the following `CREATE TABLE` statement:

```
CREATE TABLE trb3 (id INT, name VARCHAR(50), purchased DATE)
  PARTITION BY RANGE( YEAR(purchased) ) (
    PARTITION p0 VALUES LESS THAN (1990),
    PARTITION p1 VALUES LESS THAN (1995),
    PARTITION p2 VALUES LESS THAN (2000),
    PARTITION p3 VALUES LESS THAN (2005)
  );
```

To repartition this table so that it is partitioned by key into two partitions using the `id` column value as the basis for the key, you can use this statement:

```
ALTER TABLE trb3 PARTITION BY KEY(id) PARTITIONS 2;
```

This has the same effect on the structure of the table as dropping the table and re-creating it using `CREATE TABLE trb3 PARTITION BY KEY(id) PARTITIONS 2;`

`ALTER TABLE ... ENGINE = ...` changes only the storage engine used by the table, and leaves the table's partitioning scheme intact. Use `ALTER TABLE ... REMOVE PARTITIONING` to remove a table's partitioning. See [ALTER TABLE Syntax](#).

Important

Only a single `PARTITION BY`, `ADD PARTITION`, `DROP PARTITION`, `REORGANIZE PARTITION`, or `COALESCE PARTITION` clause can be used in a given `ALTER TABLE` statement. If you (for example) wish to drop a partition and reorganize a table's remaining partitions, you must do so in two separate `ALTER TABLE` statements (one using `DROP PARTITION` and then a second one using `REORGANIZE PARTITIONS`).

In MySQL 5.6, it is possible to delete all rows from one or more selected partitions using `ALTER TABLE ... TRUNCATE PARTITION`.

4.1 Management of RANGE and LIST Partitions

Range and list partitions are very similar with regard to how the adding and dropping of partitions are handled. For this reason we discuss the management of both sorts of partitioning in this section. For information about working with tables that are partitioned by hash or key, see [Section 4.2, “Management of HASH and KEY Partitions”](#). Dropping a `RANGE` or `LIST` partition is more straightforward than adding one, so we discuss this first.

Dropping a partition from a table that is partitioned by either `RANGE` or by `LIST` can be accomplished using the `ALTER TABLE` statement with a `DROP PARTITION` clause. Here is a very basic example, which supposes that you have already created a table which is partitioned by range and then populated with 10 records using the following `CREATE TABLE` and `INSERT` statements:

```
mysql> CREATE TABLE tr (id INT, name VARCHAR(50), purchased DATE)
-> PARTITION BY RANGE( YEAR(purchased) ) (
-> PARTITION p0 VALUES LESS THAN (1990),
-> PARTITION p1 VALUES LESS THAN (1995),
-> PARTITION p2 VALUES LESS THAN (2000),
-> PARTITION p3 VALUES LESS THAN (2005)
-> );
Query OK, 0 rows affected (0.01 sec)
mysql> INSERT INTO tr VALUES
-> (1, 'desk organiser', '2003-10-15'),
-> (2, 'CD player', '1993-11-05'),
-> (3, 'TV set', '1996-03-10'),
-> (4, 'bookcase', '1982-01-10'),
-> (5, 'exercise bike', '2004-05-09'),
-> (6, 'sofa', '1987-06-05'),
-> (7, 'popcorn maker', '2001-11-22'),
-> (8, 'aquarium', '1992-08-04'),
-> (9, 'study desk', '1984-09-16'),
-> (10, 'lava lamp', '1998-12-25');
Query OK, 10 rows affected (0.01 sec)
```

You can see which items should have been inserted into partition `p2` as shown here:

```
mysql> SELECT * FROM tr
-> WHERE purchased BETWEEN '1995-01-01' AND '1999-12-31';
+-----+-----+-----+
| id   | name      | purchased |
+-----+-----+-----+
| 3    | TV set    | 1996-03-10 |
| 10   | lava lamp | 1998-12-25 |
+-----+-----+-----+
2 rows in set (0.00 sec)
```

To drop the partition named `p2`, execute the following command:

```
mysql> ALTER TABLE tr DROP PARTITION p2;
Query OK, 0 rows affected (0.03 sec)
```

Note

The NDB storage engine does not support `ALTER TABLE ... DROP PARTITION`. It does, however, support the other partitioning-related extensions to `ALTER TABLE` that are described in this chapter.

It is very important to remember that, *when you drop a partition, you also delete all the data that was stored in that partition*. You can see that this is the case by re-running the previous `SELECT` query:

```
mysql> SELECT * FROM tr WHERE purchased
-> BETWEEN '1995-01-01' AND '1999-12-31';
Empty set (0.00 sec)
```

Because of this, you must have the `DROP` privilege for a table before you can execute `ALTER TABLE ... DROP PARTITION` on that table.

If you wish to drop all data from all partitions while preserving the table definition and its partitioning scheme, use the `TRUNCATE TABLE` statement. (See [TRUNCATE TABLE Syntax](#).)

If you intend to change the partitioning of a table *without* losing data, use `ALTER TABLE ... REORGANIZE PARTITION` instead. See below or in [ALTER TABLE Syntax](#), for information about `REORGANIZE PARTITION`.

If you now execute a `SHOW CREATE TABLE` statement, you can see how the partitioning makeup of the table has been changed:

```
mysql> SHOW CREATE TABLE tr\G
***** 1. row *****
      Table: tr
Create Table: CREATE TABLE `tr` (
  `id` int(11) default NULL,
  `name` varchar(50) default NULL,
  `purchased` date default NULL
) ENGINE=MyISAM DEFAULT CHARSET=latin1
PARTITION BY RANGE ( YEAR(purchased) ) (
  PARTITION p0 VALUES LESS THAN (1990) ENGINE = MyISAM,
  PARTITION p1 VALUES LESS THAN (1995) ENGINE = MyISAM,
  PARTITION p3 VALUES LESS THAN (2005) ENGINE = MyISAM
)
1 row in set (0.01 sec)
```

When you insert new rows into the changed table with `purchased` column values between `'1995-01-01'` and `'2004-12-31'` inclusive, those rows will be stored in partition `p3`. You can verify this as follows:

```
mysql> INSERT INTO tr VALUES (11, 'pencil holder', '1995-07-12');
Query OK, 1 row affected (0.00 sec)
mysql> SELECT * FROM tr WHERE purchased
-> BETWEEN '1995-01-01' AND '2004-12-31';
+-----+-----+-----+
| id  | name          | purchased |
+-----+-----+-----+
| 11  | pencil holder | 1995-07-12 |
| 1   | desk organiser | 2003-10-15 |
| 5   | exercise bike | 2004-05-09 |
| 7   | popcorn maker | 2001-11-22 |
+-----+-----+-----+
4 rows in set (0.00 sec)
mysql> ALTER TABLE tr DROP PARTITION p3;
Query OK, 0 rows affected (0.03 sec)
mysql> SELECT * FROM tr WHERE purchased
-> BETWEEN '1995-01-01' AND '2004-12-31';
Empty set (0.00 sec)
```

The number of rows dropped from the table as a result of `ALTER TABLE ... DROP PARTITION` is not reported by the server as it would be by the equivalent `DELETE` query.

Dropping **LIST** partitions uses exactly the same **ALTER TABLE ... DROP PARTITION** syntax as used for dropping **RANGE** partitions. However, there is one important difference in the effect this has on your use of the table afterward: You can no longer insert into the table any rows having any of the values that were included in the value list defining the deleted partition. (See [Section 3.2, “LIST Partitioning”](#), for an example.)

To add a new range or list partition to a previously partitioned table, use the **ALTER TABLE ... ADD PARTITION** statement. For tables which are partitioned by **RANGE**, this can be used to add a new range to the end of the list of existing partitions. Suppose that you have a partitioned table containing membership data for your organization, which is defined as follows:

```
CREATE TABLE members (
  id INT,
  fname VARCHAR(25),
  lname VARCHAR(25),
  dob DATE
)
PARTITION BY RANGE( YEAR(dob) ) (
  PARTITION p0 VALUES LESS THAN (1970),
  PARTITION p1 VALUES LESS THAN (1980),
  PARTITION p2 VALUES LESS THAN (1990)
);
```

Suppose further that the minimum age for members is 16. As the calendar approaches the end of 2005, you realize that you will soon be admitting members who were born in 1990 (and later in years to come). You can modify the **members** table to accommodate new members born in the years 1990 to 1999 as shown here:

```
ALTER TABLE members ADD PARTITION (PARTITION p3 VALUES LESS THAN (2000));
```

With tables that are partitioned by range, you can use **ADD PARTITION** to add new partitions to the high end of the partitions list only. Trying to add a new partition in this manner between or before existing partitions results in an error as shown here:

```
mysql> ALTER TABLE members
> ADD PARTITION (
> PARTITION n VALUES LESS THAN (1960));
ERROR 1463 (HY000): VALUES LESS THAN value must be strictly »
increasing for each partition
```

You can work around this problem by reorganizing the first partition into two new ones that split the range between them, like this:

```
ALTER TABLE members
  REORGANIZE PARTITION p0 INTO (
    PARTITION n0 VALUES LESS THAN (1960),
    PARTITION n1 VALUES LESS THAN (1970)
  );
```

Using **SHOW CREATE TABLE** you can see that the **ALTER TABLE** statement has had the desired effect:

```
mysql> SHOW CREATE TABLE members\G
***** 1. row *****
      Table: members
Create Table: CREATE TABLE `members` (
  `id` int(11) DEFAULT NULL,
  `fname` varchar(25) DEFAULT NULL,
  `lname` varchar(25) DEFAULT NULL,
  `dob` date DEFAULT NULL
) ENGINE=InnoDB DEFAULT CHARSET=latin1
```

```

/*!50100 PARTITION BY RANGE ( YEAR(dob))
(PARTITION n0 VALUES LESS THAN (1960) ENGINE = InnoDB,
PARTITION n1 VALUES LESS THAN (1970) ENGINE = InnoDB,
PARTITION p1 VALUES LESS THAN (1980) ENGINE = InnoDB,
PARTITION p2 VALUES LESS THAN (1990) ENGINE = InnoDB,
PARTITION p3 VALUES LESS THAN (2000) ENGINE = InnoDB) */
1 row in set (0.00 sec)

```

See also [ALTER TABLE Partition Operations](#).

You can also use [ALTER TABLE ... ADD PARTITION](#) to add new partitions to a table that is partitioned by [LIST](#). Suppose a table `tt` is defined using the following [CREATE TABLE](#) statement:

```

CREATE TABLE tt (
  id INT,
  data INT
)
PARTITION BY LIST(data) (
  PARTITION p0 VALUES IN (5, 10, 15),
  PARTITION p1 VALUES IN (6, 12, 18)
);

```

You can add a new partition in which to store rows having the `data` column values `7`, `14`, and `21` as shown:

```

ALTER TABLE tt ADD PARTITION (PARTITION p2 VALUES IN (7, 14, 21));

```

You *cannot* add a new [LIST](#) partition encompassing any values that are already included in the value list of an existing partition. If you attempt to do so, an error will result:

```

mysql> ALTER TABLE tt ADD PARTITION
> (PARTITION np VALUES IN (4, 8, 12));
ERROR 1465 (HY000): Multiple definition of same constant »
in list partitioning

```

Because any rows with the `data` column value `12` have already been assigned to partition `p1`, you cannot create a new partition on table `tt` that includes `12` in its value list. To accomplish this, you could drop `p1`, and add `np` and then a new `p1` with a modified definition. However, as discussed earlier, this would result in the loss of all data stored in `p1`—and it is often the case that this is not what you really want to do. Another solution might appear to be to make a copy of the table with the new partitioning and to copy the data into it using [CREATE TABLE ... SELECT ...](#), then drop the old table and rename the new one, but this could be very time-consuming when dealing with a large amounts of data. This also might not be feasible in situations where high availability is a requirement.

You can add multiple partitions in a single [ALTER TABLE ... ADD PARTITION](#) statement as shown here:

```

CREATE TABLE employees (
  id INT NOT NULL,
  fname VARCHAR(50) NOT NULL,
  lname VARCHAR(50) NOT NULL,
  hired DATE NOT NULL
)
PARTITION BY RANGE( YEAR(hired) ) (
  PARTITION p1 VALUES LESS THAN (1991),
  PARTITION p2 VALUES LESS THAN (1996),
  PARTITION p3 VALUES LESS THAN (2001),
  PARTITION p4 VALUES LESS THAN (2005)
);
ALTER TABLE employees ADD PARTITION (
  PARTITION p5 VALUES LESS THAN (2010),
  PARTITION p6 VALUES LESS THAN MAXVALUE
);

```

Fortunately, MySQL's partitioning implementation provides ways to redefine partitions without losing data. Let us look first at a couple of simple examples involving [RANGE](#) partitioning. Recall the [members](#) table which is now defined as shown here:

```
mysql> SHOW CREATE TABLE members\G
***** 1. row *****
      Table: members
Create Table: CREATE TABLE `members` (
  `id` int(11) default NULL,
  `fname` varchar(25) default NULL,
  `lname` varchar(25) default NULL,
  `dob` date default NULL
) ENGINE=MyISAM DEFAULT CHARSET=latin1
PARTITION BY RANGE ( YEAR(dob) ) (
  PARTITION p0 VALUES LESS THAN (1970) ENGINE = MyISAM,
  PARTITION p1 VALUES LESS THAN (1980) ENGINE = MyISAM,
  PARTITION p2 VALUES LESS THAN (1990) ENGINE = MyISAM,
  PARTITION p3 VALUES LESS THAN (2000) ENGINE = MyISAM
)
```

Suppose that you would like to move all rows representing members born before 1960 into a separate partition. As we have already seen, this cannot be done using [ALTER TABLE ... ADD PARTITION](#). However, you can use another partition-related extension to [ALTER TABLE](#) to accomplish this:

```
ALTER TABLE members REORGANIZE PARTITION p0 INTO (
  PARTITION s0 VALUES LESS THAN (1960),
  PARTITION s1 VALUES LESS THAN (1970)
);
```

In effect, this command splits partition [p0](#) into two new partitions [s0](#) and [s1](#). It also moves the data that was stored in [p0](#) into the new partitions according to the rules embodied in the two [PARTITION ... VALUES ...](#) clauses, so that [s0](#) contains only those records for which [YEAR\(dob\)](#) is less than 1960 and [s1](#) contains those rows in which [YEAR\(dob\)](#) is greater than or equal to 1960 but less than 1970.

A [REORGANIZE PARTITION](#) clause may also be used for merging adjacent partitions. You can return the [members](#) table to its previous partitioning as shown here:

```
ALTER TABLE members REORGANIZE PARTITION s0,s1 INTO (
  PARTITION p0 VALUES LESS THAN (1970)
);
```

No data is lost in splitting or merging partitions using [REORGANIZE PARTITION](#). In executing the above statement, MySQL moves all of the records that were stored in partitions [s0](#) and [s1](#) into partition [p0](#).

The general syntax for [REORGANIZE PARTITION](#) is shown here:

```
ALTER TABLE tbl_name
  REORGANIZE PARTITION partition_list
  INTO (partition_definitions);
```

Here, [tbl_name](#) is the name of the partitioned table, and [partition_list](#) is a comma-separated list of names of one or more existing partitions to be changed. [partition_definitions](#) is a comma-separated list of new partition definitions, which follow the same rules as for the [partition_definitions](#) list used in [CREATE TABLE](#) (see [CREATE TABLE Syntax](#)). It should be noted that you are not limited to merging several partitions into one, or to splitting one partition into many, when using [REORGANIZE PARTITION](#). For example, you can reorganize all four partitions of the [members](#) table into two, as follows:

```
ALTER TABLE members REORGANIZE PARTITION p0,p1,p2,p3 INTO (
  PARTITION m0 VALUES LESS THAN (1980),
  PARTITION m1 VALUES LESS THAN (2000)
```

```
);
```

You can also use `REORGANIZE PARTITION` with tables that are partitioned by `LIST`. Let us return to the problem of adding a new partition to the list-partitioned `tt` table and failing because the new partition had a value that was already present in the value-list of one of the existing partitions. We can handle this by adding a partition that contains only nonconflicting values, and then reorganizing the new partition and the existing one so that the value which was stored in the existing one is now moved to the new one:

```
ALTER TABLE tt ADD PARTITION (PARTITION np VALUES IN (4, 8));
ALTER TABLE tt REORGANIZE PARTITION p1,np INTO (
    PARTITION p1 VALUES IN (6, 18),
    PARTITION np VALUES in (4, 8, 12)
);
```

Here are some key points to keep in mind when using `ALTER TABLE ... REORGANIZE PARTITION` to repartition tables that are partitioned by `RANGE` or `LIST`:

- The `PARTITION` clauses used to determine the new partitioning scheme are subject to the same rules as those used with a `CREATE TABLE` statement.

Most importantly, you should remember that the new partitioning scheme cannot have any overlapping ranges (applies to tables partitioned by `RANGE`) or sets of values (when reorganizing tables partitioned by `LIST`).

- The combination of partitions in the `partition_definitions` list should account for the same range or set of values overall as the combined partitions named in the `partition_list`.

For instance, in the `members` table used as an example in this section, partitions `p1` and `p2` together cover the years 1980 through 1999. Therefore, any reorganization of these two partitions should cover the same range of years overall.

- For tables partitioned by `RANGE`, you can reorganize only adjacent partitions; you cannot skip over range partitions.

For instance, you could not reorganize the `members` table used as an example in this section using a statement beginning with `ALTER TABLE members REORGANIZE PARTITION p0,p2 INTO ...` because `p0` covers the years prior to 1970 and `p2` the years from 1990 through 1999 inclusive, and thus the two are not adjacent partitions.

- You cannot use `REORGANIZE PARTITION` to change the table's partitioning type; that is, you cannot (for example) change `RANGE` partitions to `HASH` partitions or vice versa. You also cannot use this command to change the partitioning expression or column. To accomplish either of these tasks without dropping and re-creating the table, you can use `ALTER TABLE ... PARTITION BY ...`. For example:

```
ALTER TABLE members
    PARTITION BY HASH( YEAR(dob) )
    PARTITIONS 8;
```

4.2 Management of HASH and KEY Partitions

Tables which are partitioned by hash or by key are very similar to one another with regard to making changes in a partitioning setup, and both differ in a number of ways from tables which have been partitioned by range or list. For that reason, this section addresses the modification of tables partitioned by hash or by key only. For a discussion of adding and dropping of partitions of tables that are partitioned by range or list, see [Section 4.1, “Management of RANGE and LIST Partitions”](#).

You cannot drop partitions from tables that are partitioned by `HASH` or `KEY` in the same way that you can from tables that are partitioned by `RANGE` or `LIST`. However, you can merge `HASH` or `KEY`

partitions using the `ALTER TABLE ... COALESCE PARTITION` statement. Suppose that you have a table containing data about clients, which is divided into twelve partitions. The `clients` table is defined as shown here:

```
CREATE TABLE clients (
  id INT,
  fname VARCHAR(30),
  lname VARCHAR(30),
  signed DATE
)
PARTITION BY HASH( MONTH(signed) )
PARTITIONS 12;
```

To reduce the number of partitions from twelve to eight, execute the following `ALTER TABLE` command:

```
mysql> ALTER TABLE clients COALESCE PARTITION 4;
Query OK, 0 rows affected (0.02 sec)
```

`COALESCE` works equally well with tables that are partitioned by `HASH`, `KEY`, `LINEAR HASH`, or `LINEAR KEY`. Here is an example similar to the previous one, differing only in that the table is partitioned by `LINEAR KEY`:

```
mysql> CREATE TABLE clients_lk (
->   id INT,
->   fname VARCHAR(30),
->   lname VARCHAR(30),
->   signed DATE
-> )
-> PARTITION BY LINEAR KEY(signed)
-> PARTITIONS 12;
Query OK, 0 rows affected (0.03 sec)
mysql> ALTER TABLE clients_lk COALESCE PARTITION 4;
Query OK, 0 rows affected (0.06 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

The number following `COALESCE PARTITION` is the number of partitions to merge into the remainder—in other words, it is the number of partitions to remove from the table.

If you attempt to remove more partitions than the table has, the result is an error like the one shown:

```
mysql> ALTER TABLE clients COALESCE PARTITION 18;
ERROR 1478 (HY000): Cannot remove all partitions, use DROP TABLE instead
```

To increase the number of partitions for the `clients` table from 12 to 18, use `ALTER TABLE ... ADD PARTITION` as shown here:

```
ALTER TABLE clients ADD PARTITION PARTITIONS 6;
```

4.3 Exchanging Partitions and Subpartitions with Tables

In MySQL 5.6, it is possible to exchange a table partition or subpartition with a table using `ALTER TABLE pt EXCHANGE PARTITION p WITH TABLE nt`, where *pt* is the partitioned table and *p* is the partition or subpartition of *pt* to be exchanged with unpartitioned table *nt*, provided that the following statements are true:

1. Table *nt* is not itself partitioned.
2. Table *nt* is not a temporary table.
3. The structures of tables *pt* and *nt* are otherwise identical.

4. Table `nt` contains no foreign key references, and no other table has any foreign keys that refer to `nt`.
5. There are no rows in `nt` that lie outside the boundaries of the partition definition for `p`.

In addition to the `ALTER`, `INSERT`, and `CREATE` privileges usually required for `ALTER TABLE` statements, you must have the `DROP` privilege to perform `ALTER TABLE ... EXCHANGE PARTITION`.

You should also be aware of the following effects of `ALTER TABLE ... EXCHANGE PARTITION`:

- Executing `ALTER TABLE ... EXCHANGE PARTITION` does not invoke any triggers on either the partitioned table or the table to be exchanged.
- Any `AUTO_INCREMENT` columns in the exchanged table are reset.
- The `IGNORE` keyword has no effect when used with `ALTER TABLE ... EXCHANGE PARTITION`.

The complete syntax of the `ALTER TABLE ... EXCHANGE PARTITION` statement is shown here, where `pt` is the partitioned table, `p` is the partition or subpartition to be exchanged, and `nt` is the nonpartitioned table to be exchanged with `p`:

```
ALTER TABLE pt
  EXCHANGE PARTITION p
  WITH TABLE nt;
```

One and only one partition or subpartition may be exchanged with one and only one nonpartitioned table in a single `ALTER TABLE EXCHANGE PARTITION` statement. To exchange multiple partitions or subpartitions, use multiple `ALTER TABLE EXCHANGE PARTITION` statements. `EXCHANGE PARTITION` may not be combined with other `ALTER TABLE` options. The partitioning and (if applicable) subpartitioning used by the partitioned table may be of any type or types supported in MySQL 5.6.

Exchanging a Partition with a Nonpartitioned Table

Suppose that a partitioned table `e` has been created and populated using the following SQL statements:

```
CREATE TABLE e (
  id INT NOT NULL,
  fname VARCHAR(30),
  lname VARCHAR(30)
)
  PARTITION BY RANGE (id) (
    PARTITION p0 VALUES LESS THAN (50),
    PARTITION p1 VALUES LESS THAN (100),
    PARTITION p2 VALUES LESS THAN (150),
    PARTITION p3 VALUES LESS THAN (MAXVALUE)
);
INSERT INTO e VALUES
  (1669, "Jim", "Smith"),
  (337, "Mary", "Jones"),
  (16, "Frank", "White"),
  (2005, "Linda", "Black");
```

Now we create a nonpartitioned copy of `e` named `e2`. This can be done using the `mysql` client as shown here:

```
mysql> CREATE TABLE e2 LIKE e;
Query OK, 0 rows affected (1.34 sec)
mysql> ALTER TABLE e2 REMOVE PARTITIONING;
```

```
Query OK, 0 rows affected (0.90 sec)
Records: 0 Duplicates: 0 Warnings: 0
```

You can see which partitions in table `e` contain rows by querying the `INFORMATION_SCHEMA.PARTITIONS` table, like this:

```
mysql> SELECT PARTITION_NAME, TABLE_ROWS
-> FROM INFORMATION_SCHEMA.PARTITIONS
-> WHERE TABLE_NAME = 'e';
+-----+-----+
| PARTITION_NAME | TABLE_ROWS |
+-----+-----+
| p0             | 1           |
| p1             | 0           |
| p2             | 0           |
| p3             | 3           |
+-----+-----+
4 rows in set (0.00 sec)
```

Note

For partitioned `InnoDB` tables, the row count given in the `TABLE_ROWS` column of the `INFORMATION_SCHEMA.PARTITIONS` table is only an estimated value used in SQL optimization, and is not always exact.

To exchange partition `p0` in table `e` with table `e2`, you can use the `ALTER TABLE` statement shown here:

```
mysql> ALTER TABLE e EXCHANGE PARTITION p0 WITH TABLE e2;
Query OK, 0 rows affected (0.28 sec)
```

More precisely, the statement just issued causes any rows found in the partition to be swapped with those found in the table. You can observe how this has happened by querying the `INFORMATION_SCHEMA.PARTITIONS` table, as before. The table row that was previously found in partition `p0` is no longer present:

```
mysql> SELECT PARTITION_NAME, TABLE_ROWS
-> FROM INFORMATION_SCHEMA.PARTITIONS
-> WHERE TABLE_NAME = 'e';
+-----+-----+
| PARTITION_NAME | TABLE_ROWS |
+-----+-----+
| p0             | 0           |
| p1             | 0           |
| p2             | 0           |
| p3             | 3           |
+-----+-----+
4 rows in set (0.00 sec)
```

If you query table `e2`, you can see that the “missing” row can now be found there:

```
mysql> SELECT * FROM e2;
+-----+-----+-----+
| id | fname | lname |
+-----+-----+-----+
| 16 | Frank | White |
+-----+-----+-----+
1 row in set (0.00 sec)
```

The table to be exchanged with the partition does not necessarily have to be empty. To demonstrate this, we first insert a new row into table `e`, making sure that this row is stored in partition `p0` by choosing an `id` column value that is less than 50, and verifying this afterwards by querying the `PARTITIONS` table:

```
mysql> INSERT INTO e VALUES (41, "Michael", "Green");
Query OK, 1 row affected (0.05 sec)
mysql> SELECT PARTITION_NAME, TABLE_ROWS
      -> FROM INFORMATION_SCHEMA.PARTITIONS
      -> WHERE TABLE_NAME = 'e';
+-----+-----+
| PARTITION_NAME | TABLE_ROWS |
+-----+-----+
| p0             |            1 |
| p1             |            0 |
| p2             |            0 |
| p3             |            3 |
+-----+-----+
4 rows in set (0.00 sec)
```

Now we once again exchange partition `p0` with table `e2` using the same `ALTER TABLE` statement as previously:

```
mysql> ALTER TABLE e EXCHANGE PARTITION p0 WITH TABLE e2;
Query OK, 0 rows affected (0.28 sec)
```

The output of the following queries shows that the table row that was stored in partition `p0` and the table row that was stored in table `e2`, prior to issuing the `ALTER TABLE` statement, have now switched places:

```
mysql> SELECT * FROM e;
+-----+-----+
| id  | fname | lname |
+-----+-----+
| 16  | Frank | White |
| 1669 | Jim   | Smith |
| 337  | Mary  | Jones |
| 2005 | Linda | Black |
+-----+-----+
4 rows in set (0.00 sec)
mysql> SELECT PARTITION_NAME, TABLE_ROWS
      -> FROM INFORMATION_SCHEMA.PARTITIONS
      -> WHERE TABLE_NAME = 'e';
+-----+-----+
| PARTITION_NAME | TABLE_ROWS |
+-----+-----+
| p0             |            1 |
| p1             |            0 |
| p2             |            0 |
| p3             |            3 |
+-----+-----+
4 rows in set (0.00 sec)
mysql> SELECT * FROM e2;
+-----+-----+
| id | fname | lname |
+-----+-----+
| 41 | Michael | Green |
+-----+-----+
1 row in set (0.00 sec)
```

Non-Matching Rows

You should keep in mind that any rows found in the nonpartitioned table prior to issuing the `ALTER TABLE ... EXCHANGE PARTITION` statement must meet the conditions required for them to be stored in the target partition; otherwise, the statement fails. To see how this occurs, first insert a row into `e2` that is outside the boundaries of the partition definition for partition `p0` of table `e`. For example, insert a row with an `id` column value that is too large; then, try to exchange the table with the partition again:

```
mysql> INSERT INTO e2 VALUES (51, "Ellen", "McDonald");
Query OK, 1 row affected (0.08 sec)
mysql> ALTER TABLE e EXCHANGE PARTITION p0 WITH TABLE e2;
ERROR 1707 (HY000): Found row that does not match the partition
```

The **IGNORE** keyword is accepted, but has no effect when used with **EXCHANGE PARTITION**, as shown here:

```
mysql> ALTER IGNORE TABLE e EXCHANGE PARTITION p0 WITH TABLE e2;
ERROR 1707 (HY000): Found row that does not match the partition
```

Exchanging a Subpartition with a Nonpartitioned Table

You can also exchange a subpartition of a subpartitioned table (see [Section 3.6, "Subpartitioning"](#)) with a nonpartitioned table using an **ALTER TABLE ... EXCHANGE PARTITION** statement. In the following example, we first create a table **es** that is partitioned by **RANGE** and subpartitioned by **KEY**, populate this table as we did table **e**, and then create an empty, nonpartitioned copy **es2** of the table, as shown here:

```
mysql> CREATE TABLE es (
->   id INT NOT NULL,
->   fname VARCHAR(30),
->   lname VARCHAR(30)
-> )
-> PARTITION BY RANGE (id)
-> SUBPARTITION BY KEY (lname)
-> SUBPARTITIONS 2 (
->   PARTITION p0 VALUES LESS THAN (50),
->   PARTITION p1 VALUES LESS THAN (100),
->   PARTITION p2 VALUES LESS THAN (150),
->   PARTITION p3 VALUES LESS THAN (MAXVALUE)
-> );
Query OK, 0 rows affected (2.76 sec)
mysql> INSERT INTO es VALUES
->   (1669, "Jim", "Smith"),
->   (337, "Mary", "Jones"),
->   (16, "Frank", "White"),
->   (2005, "Linda", "Black");
Query OK, 4 rows affected (0.04 sec)
Records: 4 Duplicates: 0 Warnings: 0
mysql> CREATE TABLE es2 LIKE es;
Query OK, 0 rows affected (1.27 sec)
mysql> ALTER TABLE es2 REMOVE PARTITIONING;
Query OK, 0 rows affected (0.70 sec)
Records: 0 Duplicates: 0 Warnings: 0
```

Although we did not explicitly name any of the subpartitions when creating table **es**, we can obtain generated names for these by including the **SUBPARTITION_NAME** of the **PARTITIONS** table from **INFORMATION_SCHEMA** when selecting from that table, as shown here:

```
mysql> SELECT PARTITION_NAME, SUBPARTITION_NAME, TABLE_ROWS
->   FROM INFORMATION_SCHEMA.PARTITIONS
->   WHERE TABLE_NAME = 'es';
+-----+-----+-----+
| PARTITION_NAME | SUBPARTITION_NAME | TABLE_ROWS |
+-----+-----+-----+
| p0             | p0sp0             | 1           |
| p0             | p0sp1             | 0           |
| p1             | p1sp0             | 0           |
| p1             | p1sp1             | 0           |
| p2             | p2sp0             | 0           |
| p2             | p2sp1             | 0           |
| p3             | p3sp0             | 3           |
| p3             | p3sp1             | 0           |
+-----+-----+-----+
8 rows in set (0.00 sec)
```

The following `ALTER TABLE` statement exchanges subpartition `p3sp0` table `es` with the nonpartitioned table `es2`:

```
mysql> ALTER TABLE es EXCHANGE PARTITION p3sp0 WITH TABLE es2;
Query OK, 0 rows affected (0.29 sec)
```

You can verify that the rows were exchanged by issuing the following queries:

```
mysql> SELECT PARTITION_NAME, SUBPARTITION_NAME, TABLE_ROWS
-> FROM INFORMATION_SCHEMA.PARTITIONS
-> WHERE TABLE_NAME = 'es';
+-----+-----+-----+
| PARTITION_NAME | SUBPARTITION_NAME | TABLE_ROWS |
+-----+-----+-----+
| p0              | p0sp0              | 1           |
| p0              | p0sp1              | 0           |
| p1              | p1sp0              | 0           |
| p1              | p1sp1              | 0           |
| p2              | p2sp0              | 0           |
| p2              | p2sp1              | 0           |
| p3              | p3sp0              | 0           |
| p3              | p3sp1              | 0           |
+-----+-----+-----+
8 rows in set (0.00 sec)
mysql> SELECT * FROM es2;
+-----+-----+-----+
| id   | fname | lname |
+-----+-----+-----+
| 1669 | Jim   | Smith |
| 337  | Mary  | Jones |
| 2005 | Linda | Black |
+-----+-----+-----+
3 rows in set (0.00 sec)
```

If a table is subpartitioned, you can exchange only a subpartition of the table—not an entire partition—with an unpartitioned table, as shown here:

```
mysql> ALTER TABLE es EXCHANGE PARTITION p3 WITH TABLE es2;
ERROR 1704 (HY000): Subpartitioned table, use subpartition instead of partition
```

The comparison of table structures used by MySQL is very strict. The number, order, names, and types of columns and indexes of the partitioned table and the nonpartitioned table must match exactly. In addition, both tables must use the same storage engine:

```
mysql> CREATE TABLE es3 LIKE e;
Query OK, 0 rows affected (1.31 sec)
mysql> ALTER TABLE es3 REMOVE PARTITIONING;
Query OK, 0 rows affected (0.53 sec)
Records: 0 Duplicates: 0 Warnings: 0
mysql> SHOW CREATE TABLE es3\G
***** 1. row *****
      Table: es3
Create Table: CREATE TABLE `es3` (
  `id` int(11) NOT NULL,
  `fname` varchar(30) DEFAULT NULL,
  `lname` varchar(30) DEFAULT NULL
) ENGINE=InnoDB DEFAULT CHARSET=latin1
1 row in set (0.00 sec)
mysql> ALTER TABLE es3 ENGINE = MyISAM;
Query OK, 0 rows affected (0.15 sec)
Records: 0 Duplicates: 0 Warnings: 0
mysql> ALTER TABLE es EXCHANGE PARTITION p3sp0 WITH TABLE es3;
ERROR 1497 (HY000): The mix of handlers in the partitions is not allowed in this version of MySQL
```

4.4 Maintenance of Partitions

A number of table and partition maintenance tasks can be carried out using SQL statements intended for such purposes on partitioned tables in MySQL 5.6.

Table maintenance of partitioned tables can be accomplished using the statements [CHECK TABLE](#), [OPTIMIZE TABLE](#), [ANALYZE TABLE](#), and [REPAIR TABLE](#), which are supported for partitioned tables.

You can use a number of extensions to [ALTER TABLE](#) for performing operations of this type on one or more partitions directly, as described in the following list:

- **Rebuilding partitions.** Rebuilds the partition; this has the same effect as dropping all records stored in the partition, then reinserting them. This can be useful for purposes of defragmentation.

Example:

```
ALTER TABLE t1 REBUILD PARTITION p0, p1;
```

- **Optimizing partitions.** If you have deleted a large number of rows from a partition or if you have made many changes to a partitioned table with variable-length rows (that is, having [VARCHAR](#), [BLOB](#), or [TEXT](#) columns), you can use [ALTER TABLE ... OPTIMIZE PARTITION](#) to reclaim any unused space and to defragment the partition data file.

Example:

```
ALTER TABLE t1 OPTIMIZE PARTITION p0, p1;
```

Using [OPTIMIZE PARTITION](#) on a given partition is equivalent to running [CHECK PARTITION](#), [ANALYZE PARTITION](#), and [REPAIR PARTITION](#) on that partition.

Some MySQL storage engines, including [InnoDB](#), do not support per-partition optimization; in these cases, [ALTER TABLE ... OPTIMIZE PARTITION](#) rebuilds the entire table. In MySQL 5.6.9 and later, running this statement on such a table causes the entire table to be rebuilt and analyzed, and an appropriate warning to be issued. (Bug #11751825, Bug #42822) Use [ALTER TABLE ... REBUILD PARTITION](#) and [ALTER TABLE ... ANALYZE PARTITION](#) instead, to avoid this issue.

- **Analyzing partitions.** This reads and stores the key distributions for partitions.

Example:

```
ALTER TABLE t1 ANALYZE PARTITION p3;
```

- **Repairing partitions.** This repairs corrupted partitions.

Example:

```
ALTER TABLE t1 REPAIR PARTITION p0,p1;
```

- **Checking partitions.** You can check partitions for errors in much the same way that you can use [CHECK TABLE](#) with nonpartitioned tables.

Example:

```
ALTER TABLE trb3 CHECK PARTITION p1;
```

This command will tell you if the data or indexes in partition [p1](#) of table [t1](#) are corrupted. If this is the case, use [ALTER TABLE ... REPAIR PARTITION](#) to repair the partition.

Each of the statements in the list just shown also supports the keyword [ALL](#) in place of the list of partition names. Using [ALL](#) causes the statement to act on all partitions in the table.

The use of `mysqlcheck` and `myisamchk` is not supported with partitioned tables.

In MySQL 5.6, you can also truncate partitions using `ALTER TABLE ... TRUNCATE PARTITION`. This statement can be used to delete all rows from one or more partitions in much the same way that `TRUNCATE TABLE` deletes all rows from a table.

`ALTER TABLE ... TRUNCATE PARTITION ALL` truncates all partitions in the table.

`ANALYZE`, `CHECK`, `OPTIMIZE`, `REBUILD`, `REPAIR`, and `TRUNCATE` operations are not supported for subpartitions.

4.5 Obtaining Information About Partitions

This section discusses obtaining information about existing partitions, which can be done in a number of ways. Methods of obtaining such information include the following:

- Using the `SHOW CREATE TABLE` statement to view the partitioning clauses used in creating a partitioned table.
- Using the `SHOW TABLE STATUS` statement to determine whether a table is partitioned.
- Querying the `INFORMATION_SCHEMA.PARTITIONS` table.
- Using the statement `EXPLAIN PARTITIONS SELECT` to see which partitions are used by a given `SELECT`.

As discussed elsewhere in this chapter, `SHOW CREATE TABLE` includes in its output the `PARTITION BY` clause used to create a partitioned table. For example:

```
mysql> SHOW CREATE TABLE trb3\G
***** 1. row *****
      Table: trb3
Create Table: CREATE TABLE `trb3` (
  `id` int(11) default NULL,
  `name` varchar(50) default NULL,
  `purchased` date default NULL
) ENGINE=MyISAM DEFAULT CHARSET=latin1
PARTITION BY RANGE (YEAR(purchased)) (
  PARTITION p0 VALUES LESS THAN (1990) ENGINE = MyISAM,
  PARTITION p1 VALUES LESS THAN (1995) ENGINE = MyISAM,
  PARTITION p2 VALUES LESS THAN (2000) ENGINE = MyISAM,
  PARTITION p3 VALUES LESS THAN (2005) ENGINE = MyISAM
)
1 row in set (0.00 sec)
```

The output from `SHOW TABLE STATUS` for partitioned tables is the same as that for nonpartitioned tables, except that the `Create_options` column contains the string `partitioned`. The `Engine` column contains the name of the storage engine used by all partitions of the table. (See [SHOW TABLE STATUS Syntax](#), for more information about this statement.)

You can also obtain information about partitions from `INFORMATION_SCHEMA`, which contains a `PARTITIONS` table. See [The INFORMATION_SCHEMA PARTITIONS Table](#).

It is possible to determine which partitions of a partitioned table are involved in a given `SELECT` query using `EXPLAIN PARTITIONS`. The `PARTITIONS` keyword adds a `partitions` column to the output of `EXPLAIN` listing the partitions from which records would be matched by the query.

Suppose that you have a table `trb1` created and populated as follows:

```
CREATE TABLE trb1 (id INT, name VARCHAR(50), purchased DATE)
PARTITION BY RANGE(id)
(
```

```
        PARTITION p0 VALUES LESS THAN (3),
        PARTITION p1 VALUES LESS THAN (7),
        PARTITION p2 VALUES LESS THAN (9),
        PARTITION p3 VALUES LESS THAN (11)
    );
INSERT INTO trb1 VALUES
    (1, 'desk organiser', '2003-10-15'),
    (2, 'CD player', '1993-11-05'),
    (3, 'TV set', '1996-03-10'),
    (4, 'bookcase', '1982-01-10'),
    (5, 'exercise bike', '2004-05-09'),
    (6, 'sofa', '1987-06-05'),
    (7, 'popcorn maker', '2001-11-22'),
    (8, 'aquarium', '1992-08-04'),
    (9, 'study desk', '1984-09-16'),
    (10, 'lava lamp', '1998-12-25');
```

You can see which partitions are used in a query such as `SELECT * FROM trb1;`, as shown here:

```
mysql> EXPLAIN PARTITIONS SELECT * FROM trb1\G
***** 1. row *****
      id: 1
    select_type: SIMPLE
        table: trb1
    partitions: p0,p1,p2,p3
         type: ALL
possible_keys: NULL
          key: NULL
        key_len: NULL
          ref: NULL
         rows: 10
      Extra: Using filesort
```

In this case, all four partitions are searched. However, when a limiting condition making use of the partitioning key is added to the query, you can see that only those partitions containing matching values are scanned, as shown here:

```
mysql> EXPLAIN PARTITIONS SELECT * FROM trb1 WHERE id < 5\G
***** 1. row *****
      id: 1
    select_type: SIMPLE
        table: trb1
    partitions: p0,p1
         type: ALL
possible_keys: NULL
          key: NULL
        key_len: NULL
          ref: NULL
         rows: 10
      Extra: Using where
```

`EXPLAIN PARTITIONS` provides information about keys used and possible keys, just as with the standard `EXPLAIN SELECT` statement:

```
mysql> ALTER TABLE trb1 ADD PRIMARY KEY (id);
Query OK, 10 rows affected (0.03 sec)
Records: 10 Duplicates: 0 Warnings: 0
mysql> EXPLAIN PARTITIONS SELECT * FROM trb1 WHERE id < 5\G
***** 1. row *****
      id: 1
    select_type: SIMPLE
        table: trb1
    partitions: p0,p1
         type: range
possible_keys: PRIMARY
          key: PRIMARY
        key_len: 4
          ref: NULL
```

```
rows: 7  
Extra: Using where
```

You should take note of the following restrictions and limitations on [EXPLAIN PARTITIONS](#):

- You cannot use the [PARTITIONS](#) and [EXTENDED](#) keywords together in the same [EXPLAIN ... SELECT](#) statement. Attempting to do so produces a syntax error.
- If [EXPLAIN PARTITIONS](#) is used to examine a query against a nonpartitioned table, no error is produced, but the value of the [partitions](#) column is always [NULL](#).

The [rows](#) column of [EXPLAIN PARTITIONS](#) output displays the total number of rows in the table.

See also [EXPLAIN Syntax](#).

Chapter 5 Partition Pruning

This section discusses an optimization known as *partition pruning*. The core concept behind partition pruning is relatively simple, and can be described as “Do not scan partitions where there can be no matching values”. Suppose that you have a partitioned table `t1` defined by this statement:

```
CREATE TABLE t1 (  
    fname VARCHAR(50) NOT NULL,  
    lname VARCHAR(50) NOT NULL,  
    region_code TINYINT UNSIGNED NOT NULL,  
    dob DATE NOT NULL  
)  
PARTITION BY RANGE( region_code ) (  
    PARTITION p0 VALUES LESS THAN (64),  
    PARTITION p1 VALUES LESS THAN (128),  
    PARTITION p2 VALUES LESS THAN (192),  
    PARTITION p3 VALUES LESS THAN MAXVALUE  
);
```

Consider the case where you wish to obtain results from a `SELECT` statement such as this one:

```
SELECT fname, lname, region_code, dob  
FROM t1  
WHERE region_code > 125 AND region_code < 130;
```

It is easy to see that none of the rows which ought to be returned will be in either of the partitions `p0` or `p3`; that is, we need to search only in partitions `p1` and `p2` to find matching rows. By doing so, it is possible to expend much less time and effort in finding matching rows than would be required to scan all partitions in the table. This “cutting away” of unneeded partitions is known as *pruning*. When the optimizer can make use of partition pruning in performing this query, execution of the query can be an order of magnitude faster than the same query against a nonpartitioned table containing the same column definitions and data.

Note

When pruning is performed on a partitioned `MyISAM` table, all partitions are opened, whether or not they are examined, due to the design of the `MyISAM` storage engine. This means that you must have a sufficient number of file descriptors available to cover all partitions of the table. See [MyISAM and partition file descriptor usage](#).

This limitation does not apply to partitioned tables using other MySQL storage engines such as `InnoDB`.

The optimizer can perform pruning whenever a `WHERE` condition can be reduced to either one of the following two cases:

- `partition_column = constant`
- `partition_column IN (constant1, constant2, ..., constantN)`

In the first case, the optimizer simply evaluates the partitioning expression for the value given, determines which partition contains that value, and scans only this partition. In many cases, the equal sign can be replaced with another arithmetic comparison, including `<`, `>`, `<=`, `>=`, and `<>`. Some queries using `BETWEEN` in the `WHERE` clause can also take advantage of partition pruning. See the examples later in this section.

In the second case, the optimizer evaluates the partitioning expression for each value in the list, creates a list of matching partitions, and then scans only the partitions in this partition list.

MySQL can apply partition pruning to `SELECT`, `DELETE`, and `UPDATE` statements. `INSERT` statements currently cannot be pruned.

Pruning can also be applied to short ranges, which the optimizer can convert into equivalent lists of values. For instance, in the previous example, the `WHERE` clause can be converted to `WHERE region_code IN (126, 127, 128, 129)`. Then the optimizer can determine that the first two values in the list are found in partition `p1`, the remaining two values in partition `p2`, and that the other partitions contain no relevant values and so do not need to be searched for matching rows.

In MySQL 5.6, the optimizer can also perform pruning for `WHERE` conditions that involve comparisons of the preceding types on multiple columns for tables that use `RANGE COLUMNS` or `LIST COLUMNS` partitioning.

This type of optimization can be applied whenever the partitioning expression consists of an equality or a range which can be reduced to a set of equalities, or when the partitioning expression represents an increasing or decreasing relationship. Pruning can also be applied for tables partitioned on a `DATE` or `DATETIME` column when the partitioning expression uses the `YEAR()` or `TO_DAYS()` function. In addition, in MySQL 5.6, pruning can be applied for such tables when the partitioning expression uses the `TO_SECONDS()` function.

Suppose that table `t2`, defined as shown here, is partitioned on a `DATE` column:

```
CREATE TABLE t2 (
  fname VARCHAR(50) NOT NULL,
  lname VARCHAR(50) NOT NULL,
  region_code TINYINT UNSIGNED NOT NULL,
  dob DATE NOT NULL
)
PARTITION BY RANGE( YEAR(dob) ) (
  PARTITION d0 VALUES LESS THAN (1970),
  PARTITION d1 VALUES LESS THAN (1975),
  PARTITION d2 VALUES LESS THAN (1980),
  PARTITION d3 VALUES LESS THAN (1985),
  PARTITION d4 VALUES LESS THAN (1990),
  PARTITION d5 VALUES LESS THAN (2000),
  PARTITION d6 VALUES LESS THAN (2005),
  PARTITION d7 VALUES LESS THAN MAXVALUE
);
```

The following statements using `t2` can make use of partition pruning:

```
SELECT * FROM t2 WHERE dob = '1982-06-23';
UPDATE t2 SET region_code = 8 WHERE dob BETWEEN '1991-02-15' AND '1997-04-25';
DELETE FROM t2 WHERE dob >= '1984-06-21' AND dob <= '1999-06-21'
```

In the case of the last statement, the optimizer can also act as follows:

1. Find the partition containing the low end of the range.

`YEAR('1984-06-21')` yields the value `1984`, which is found in partition `d3`.

2. Find the partition containing the high end of the range.

`YEAR('1999-06-21')` evaluates to `1999`, which is found in partition `d5`.

3. Scan only these two partitions and any partitions that may lie between them.

In this case, this means that only partitions `d3`, `d4`, and `d5` are scanned. The remaining partitions may be safely ignored (and are ignored).

Important

Invalid `DATE` and `DATETIME` values referenced in the `WHERE` condition of a statement against a partitioned table are treated as `NULL`. This means that a query such as `SELECT * FROM partitioned_table WHERE`

`date_column < '2008-12-00'` does not return any values (see Bug #40972).

So far, we have looked only at examples using **RANGE** partitioning, but pruning can be applied with other partitioning types as well.

Consider a table that is partitioned by **LIST**, where the partitioning expression is increasing or decreasing, such as the table `t3` shown here. (In this example, we assume for the sake of brevity that the `region_code` column is limited to values between 1 and 10 inclusive.)

```
CREATE TABLE t3 (  
    fname VARCHAR(50) NOT NULL,  
    lname VARCHAR(50) NOT NULL,  
    region_code TINYINT UNSIGNED NOT NULL,  
    dob DATE NOT NULL  
)  
PARTITION BY LIST(region_code) (  
    PARTITION r0 VALUES IN (1, 3),  
    PARTITION r1 VALUES IN (2, 5, 8),  
    PARTITION r2 VALUES IN (4, 9),  
    PARTITION r3 VALUES IN (6, 7, 10)  
);
```

For a statement such as `SELECT * FROM t3 WHERE region_code BETWEEN 1 AND 3`, the optimizer determines in which partitions the values 1, 2, and 3 are found (`r0` and `r1`) and skips the remaining ones (`r2` and `r3`).

For tables that are partitioned by **HASH** or **[LINEAR] KEY**, partition pruning is also possible in cases in which the `WHERE` clause uses a simple `=` relation against a column used in the partitioning expression. Consider a table created like this:

```
CREATE TABLE t4 (  
    fname VARCHAR(50) NOT NULL,  
    lname VARCHAR(50) NOT NULL,  
    region_code TINYINT UNSIGNED NOT NULL,  
    dob DATE NOT NULL  
)  
PARTITION BY KEY(region_code)  
PARTITIONS 8;
```

A statement that compares a column value with a constant can be pruned:

```
UPDATE t4 WHERE region_code = 7;
```

Pruning can also be employed for short ranges, because the optimizer can turn such conditions into **IN** relations. For example, using the same table `t4` as defined previously, queries such as these can be pruned:

```
SELECT * FROM t4 WHERE region_code > 2 AND region_code < 6;  
SELECT * FROM t4 WHERE region_code BETWEEN 3 AND 5;
```

In both these cases, the `WHERE` clause is transformed by the optimizer into `WHERE region_code IN (3, 4, 5)`.

Important

This optimization is used only if the range size is smaller than the number of partitions. Consider this statement:

```
DELETE FROM t4 WHERE region_code BETWEEN 4 AND 12;
```

The range in the `WHERE` clause covers 9 values (4, 5, 6, 7, 8, 9, 10, 11, 12), but `t4` has only 8 partitions. This means that the `DELETE` cannot be pruned.

When a table is partitioned by `HASH` or `[LINEAR] KEY`, pruning can be used only on integer columns. For example, this statement cannot use pruning because `dob` is a `DATE` column:

```
SELECT * FROM t4 WHERE dob >= '2001-04-14' AND dob <= '2005-10-15';
```

However, if the table stores year values in an `INT` column, then a query having `WHERE year_col >= 2001 AND year_col <= 2005` can be pruned.

In MySQL 5.6.8 and later, partition pruning is disabled for all tables using a storage engine that provides automatic partitioning, such as the `NDB` storage engine used by MySQL Cluster. (Bug #14672885) Beginning with MySQL 5.6.10, such tables can be pruned if they are explicitly partitioned. (Bug #14827952)

Chapter 6 Restrictions and Limitations on Partitioning

Table of Contents

6.1 Partitioning Keys, Primary Keys, and Unique Keys	63
6.2 Partitioning Limitations Relating to Storage Engines	66
6.3 Partitioning Limitations Relating to Functions	67
6.4 Partitioning and Locking	68

This section discusses current restrictions and limitations on MySQL partitioning support.

Prohibited constructs. The following constructs are not permitted in partitioning expressions:

- Stored procedures, stored functions, UDFs, or plugins.
- Declared variables or user variables.

For a list of SQL functions which are permitted in partitioning expressions, see [Section 6.3, “Partitioning Limitations Relating to Functions”](#).

Arithmetic and logical operators. Use of the arithmetic operators `+`, `-`, and `*` is permitted in partitioning expressions. However, the result must be an integer value or `NULL` (except in the case of [\[`LINEAR`\] `KEY`](#) partitioning, as discussed elsewhere in this chapter; see [Chapter 3, *Partitioning Types*](#), for more information).

The `DIV` operator is also supported, and the `/` operator is not permitted. (Bug #30188, Bug #33182)

The bit operators `|`, `&`, `^`, `<<`, `>>`, and `~` are not permitted in partitioning expressions.

HANDLER statements. In MySQL 5.6, the `HANDLER` statement is not supported with partitioned tables.

Server SQL mode. Tables employing user-defined partitioning do not preserve the SQL mode in effect at the time that they were created. As discussed in [Server SQL Modes](#), the results of many MySQL functions and operators may change according to the server SQL mode. Therefore, a change in the SQL mode at any time after the creation of partitioned tables may lead to major changes in the behavior of such tables, and could easily lead to corruption or loss of data. For these reasons, *it is strongly recommended that you never change the server SQL mode after creating partitioned tables.*

Examples. The following examples illustrate some changes in behavior of partitioned tables due to a change in the server SQL mode:

1. **Error handling.** Suppose that you create a partitioned table whose partitioning expression is one such as `column DIV 0` or `column MOD 0`, as shown here:

```
mysql> CREATE TABLE tn (c1 INT)
-> PARTITION BY LIST(1 DIV c1) (
-> PARTITION p0 VALUES IN (NULL),
-> PARTITION p1 VALUES IN (1)
-> );
Query OK, 0 rows affected (0.05 sec)
```

The default behavior for MySQL is to return `NULL` for the result of a division by zero, without producing any errors:

```
mysql> SELECT @@sql_mode;
+-----+
| @@sql_mode |
```

```

+-----+
|       |
+-----+
1 row in set (0.00 sec)
mysql> INSERT INTO tn VALUES (NULL), (0), (1);
Query OK, 3 rows affected (0.00 sec)
Records: 3  Duplicates: 0  Warnings: 0

```

However, changing the server SQL mode to treat division by zero as an error and to enforce strict error handling causes the same `INSERT` statement to fail, as shown here:

```

mysql> SET sql_mode='STRICT_ALL_TABLES,ERROR_FOR_DIVISION_BY_ZERO';
Query OK, 0 rows affected (0.00 sec)
mysql> INSERT INTO tn VALUES (NULL), (0), (1);
ERROR 1365 (22012): Division by 0

```

2. **Table accessibility.** Sometimes a change in the server SQL mode can make partitioned tables unusable. The following `CREATE TABLE` statement can be executed successfully only if the `NO_UNSIGNED_SUBTRACTION` mode is in effect:

```

mysql> SELECT @@sql_mode;
+-----+
| @@sql_mode |
+-----+
|           |
+-----+
1 row in set (0.00 sec)
mysql> CREATE TABLE tu (c1 BIGINT UNSIGNED)
-> PARTITION BY RANGE(c1 - 10) (
-> PARTITION p0 VALUES LESS THAN (-5),
-> PARTITION p1 VALUES LESS THAN (0),
-> PARTITION p2 VALUES LESS THAN (5),
-> PARTITION p3 VALUES LESS THAN (10),
-> PARTITION p4 VALUES LESS THAN (MAXVALUE)
-> );
ERROR 1563 (HY000): Partition constant is out of partition function domain
mysql> SET sql_mode='NO_UNSIGNED_SUBTRACTION';
Query OK, 0 rows affected (0.00 sec)
mysql> SELECT @@sql_mode;
+-----+
| @@sql_mode |
+-----+
| NO_UNSIGNED_SUBTRACTION |
+-----+
1 row in set (0.00 sec)
mysql> CREATE TABLE tu (c1 BIGINT UNSIGNED)
-> PARTITION BY RANGE(c1 - 10) (
-> PARTITION p0 VALUES LESS THAN (-5),
-> PARTITION p1 VALUES LESS THAN (0),
-> PARTITION p2 VALUES LESS THAN (5),
-> PARTITION p3 VALUES LESS THAN (10),
-> PARTITION p4 VALUES LESS THAN (MAXVALUE)
-> );
Query OK, 0 rows affected (0.05 sec)

```

If you remove the `NO_UNSIGNED_SUBTRACTION` server SQL mode after creating `tu`, you may no longer be able to access this table:

```

mysql> SET sql_mode='';
Query OK, 0 rows affected (0.00 sec)
mysql> SELECT * FROM tu;
ERROR 1563 (HY000): Partition constant is out of partition function domain
mysql> INSERT INTO tu VALUES (20);
ERROR 1563 (HY000): Partition constant is out of partition function domain

```

Server SQL modes also impact replication of partitioned tables. Differing SQL modes on master and slave can lead to partitioning expressions being evaluated differently; this can cause the distribution of

data among partitions to be different in the master's and slave's copies of a given table, and may even cause inserts into partitioned tables that succeed on the master to fail on the slave. For best results, you should always use the same server SQL mode on the master and on the slave.

Performance considerations. Some affects of partitioning operations on performance are given in the following list:

- **File system operations.** Partitioning and repartitioning operations (such as `ALTER TABLE` with `PARTITION BY ...`, `REORGANIZE PARTITIONS`, or `REMOVE PARTITIONING`) depend on file system operations for their implementation. This means that the speed of these operations is affected by such factors as file system type and characteristics, disk speed, swap space, file handling efficiency of the operating system, and MySQL server options and variables that relate to file handling. In particular, you should make sure that `large_files_support` is enabled and that `open_files_limit` is set properly. For partitioned tables using the `MyISAM` storage engine, increasing `myisam_max_sort_file_size` may improve performance; partitioning and repartitioning operations involving `InnoDB` tables may be made more efficient by enabling `innodb_file_per_table`.

See also [Maximum number of partitions](#).

- **MyISAM and partition file descriptor usage.** For a partitioned `MyISAM` table, MySQL uses 2 file descriptors for each partition, for each such table that is open. This means that you need many more file descriptors to perform operations on a partitioned `MyISAM` table than on a table which is identical to it except that the latter table is not partitioned, particularly when performing `ALTER TABLE` operations.

Assume a `MyISAM` table `t` with 100 partitions, such as the table created by this SQL statement:

```
CREATE TABLE t (c1 VARCHAR(50))
PARTITION BY KEY (c1) PARTITIONS 100
ENGINE=MYISAM;
```

Note

For brevity, we use `KEY` partitioning for the table shown in this example, but file descriptor usage as described here applies to all partitioned `MyISAM` tables, regardless of the type of partitioning that is employed. Partitioned tables using other storage engines such as `InnoDB` are not affected by this issue.

Now assume that you wish to repartition `t` so that it has 101 partitions, using the statement shown here:

```
ALTER TABLE t PARTITION BY KEY (c1) PARTITIONS 101;
```

To process this `ALTER TABLE` statement, MySQL uses 402 file descriptors—that is, two for each of the 100 original partitions, plus two for each of the 101 new partitions. This is because all partitions (old and new) must be opened concurrently during the reorganization of the table data. It is recommended that, if you expect to perform such operations, you should make sure that `--open-files-limit` is not set too low to accommodate them.

- **Table locks.** The process executing a partitioning operation on a table takes a write lock on the table. Reads from such tables are relatively unaffected; pending `INSERT` and `UPDATE` operations are performed as soon as the partitioning operation has completed.
- **Storage engine.** Partitioning operations, queries, and update operations generally tend to be faster with `MyISAM` tables than with `InnoDB` or `NDB` tables.
- **Indexes; partition pruning.** As with nonpartitioned tables, proper use of indexes can speed up queries on partitioned tables significantly. In addition, designing partitioned tables and queries

on these tables to take advantage of *partition pruning* can improve performance dramatically. See [Chapter 5, Partition Pruning](#), for more information.

Index condition pushdown is not supported for partitioned tables. See [Index Condition Pushdown Optimization](#).

- **Performance with LOAD DATA.** In MySQL 5.6, [LOAD DATA](#) uses buffering to improve performance. You should be aware that the buffer uses 130 KB memory per partition to achieve this.

Maximum number of partitions.

Prior to MySQL 5.6.7, the maximum possible number of partitions for a given table not using the [NDB](#) storage engine was 1024. Beginning with MySQL 5.6.7, this limit is increased to 8192 partitions. Regardless of the MySQL Server version, this maximum includes subpartitions.

The maximum possible number of user-defined partitions for a table using the [NDB](#) storage engine is determined according to the version of the MySQL Cluster software being used, the number of data nodes, and other factors. See [NDB and user-defined partitioning](#), for more information.

If, when creating tables with a large number of partitions (but less than the maximum), you encounter an error message such as `Got error ... from storage engine: Out of resources when opening file`, you may be able to address the issue by increasing the value of the [open_files_limit](#) system variable. However, this is dependent on the operating system, and may not be possible or advisable on all platforms; see [File Not Found and Similar Errors](#), for more information. In some cases, using large numbers (hundreds) of partitions may also not be advisable due to other concerns, so using more partitions does not automatically lead to better results.

See also [File system operations](#).

Query cache not supported.

The query cache is not supported for partitioned tables. Beginning with MySQL 5.6.5, the query cache is automatically disabled for queries involving partitioned tables, and cannot be enabled for such queries. (Bug #53775)

Per-partition key caches.

In MySQL 5.6, key caches are supported for partitioned [MyISAM](#) tables, using the [CACHE INDEX](#) and [LOAD INDEX INTO CACHE](#) statements. Key caches may be defined for one, several, or all partitions, and indexes for one, several, or all partitions may be preloaded into key caches.

Foreign keys not supported for partitioned InnoDB tables.

Partitioned tables using the [InnoDB](#) storage engine do not support foreign keys. More specifically, this means that the following two statements are true:

1. No definition of an [InnoDB](#) table employing user-defined partitioning may contain foreign key references; no [InnoDB](#) table whose definition contains foreign key references may be partitioned.
2. No [InnoDB](#) table definition may contain a foreign key reference to a user-partitioned table; no [InnoDB](#) table with user-defined partitioning may contain columns referenced by foreign keys.

The scope of the restrictions just listed includes all tables that use the [InnoDB](#) storage engine. [CREATE TABLE](#) and [ALTER TABLE](#) statements that would result in tables violating these restrictions are not allowed.

ALTER TABLE ... ORDER BY. An [ALTER TABLE ... ORDER BY column](#) statement run against a partitioned table causes ordering of rows only within each partition.

Effects on REPLACE statements by modification of primary keys. It can be desirable in some cases (see [Section 6.1, “Partitioning Keys, Primary Keys, and Unique Keys”](#)) to modify a table's primary key. Be aware that, if your application uses [REPLACE](#) statements and you do this, the results of these statements can be drastically altered. See [REPLACE Syntax](#), for more information and an example.

FULLTEXT indexes.

Partitioned tables do not support [FULLTEXT](#) indexes or searches, even for partitioned tables employing the [InnoDB](#) or [MyISAM](#) storage engine.

Spatial columns. Columns with spatial data types such as [POINT](#) or [GEOMETRY](#) cannot be used in partitioned tables.

Temporary tables.

Temporary tables cannot be partitioned. (Bug #17497)

Log tables. It is not possible to partition the log tables; an [ALTER TABLE ... PARTITION BY ...](#) statement on such a table fails with an error.

Data type of partitioning key.

A partitioning key must be either an integer column or an expression that resolves to an integer. Expressions employing [ENUM](#) columns cannot be used. The column or expression value may also be [NULL](#). (See [Section 3.7, “How MySQL Partitioning Handles NULL”](#).)

There are two exceptions to this restriction:

1. When partitioning by [\[LINEAR\] KEY](#), it is possible to use columns of any valid MySQL data type other than [TEXT](#) or [BLOB](#) as partitioning keys, because MySQL's internal key-hashing functions produce the correct data type from these types. For example, the following two [CREATE TABLE](#) statements are valid:

```
CREATE TABLE tkc (c1 CHAR)
PARTITION BY KEY(c1)
PARTITIONS 4;
CREATE TABLE tke
  ( c1 ENUM('red', 'orange', 'yellow', 'green', 'blue', 'indigo', 'violet') )
PARTITION BY LINEAR KEY(c1)
PARTITIONS 6;
```

2. When partitioning by [RANGE COLUMNS](#) or [LIST COLUMNS](#), it is possible to use string, [DATE](#), and [DATETIME](#) columns. For example, each of the following [CREATE TABLE](#) statements is valid:

```
CREATE TABLE rc (c1 INT, c2 DATE)
PARTITION BY RANGE COLUMNS(c2) (
  PARTITION p0 VALUES LESS THAN('1990-01-01'),
  PARTITION p1 VALUES LESS THAN('1995-01-01'),
  PARTITION p2 VALUES LESS THAN('2000-01-01'),
  PARTITION p3 VALUES LESS THAN('2005-01-01'),
  PARTITION p4 VALUES LESS THAN(MAXVALUE)
);
CREATE TABLE lc (c1 INT, c2 CHAR(1))
PARTITION BY LIST COLUMNS(c2) (
  PARTITION p0 VALUES IN('a', 'd', 'g', 'j', 'm', 'p', 's', 'v', 'y'),
  PARTITION p1 VALUES IN('b', 'e', 'h', 'k', 'n', 'q', 't', 'w', 'z'),
  PARTITION p2 VALUES IN('c', 'f', 'i', 'l', 'o', 'r', 'u', 'x', NULL)
);
```

Neither of the preceding exceptions applies to [BLOB](#) or [TEXT](#) column types.

Subqueries.

A partitioning key may not be a subquery, even if that subquery resolves to an integer value or [NULL](#).

Issues with subpartitions.

Subpartitions must use [HASH](#) or [KEY](#) partitioning. Only [RANGE](#) and [LIST](#) partitions may be subpartitioned; [HASH](#) and [KEY](#) partitions cannot be subpartitioned.

[SUBPARTITION BY KEY](#) requires that the subpartitioning column or columns be specified explicitly, unlike the case with [PARTITION BY KEY](#), where it can be omitted (in which case the table's primary key column is used by default). Consider the table created by this statement:

```
CREATE TABLE ts (
  id INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
  name VARCHAR(30)
);
```

You can create a table having the same columns, partitioned by [KEY](#), using a statement such as this one:

```
CREATE TABLE ts (
  id INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
  name VARCHAR(30)
)
PARTITION BY KEY()
PARTITIONS 4;
```

The previous statement is treated as though it had been written like this, with the table's primary key column used as the partitioning column:

```
CREATE TABLE ts (
  id INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
  name VARCHAR(30)
)
PARTITION BY KEY(id)
PARTITIONS 4;
```

However, the following statement that attempts to create a subpartitioned table using the default column as the subpartitioning column fails, and the column must be specified for the statement to succeed, as shown here:

```
mysql> CREATE TABLE ts (
->   id INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
->   name VARCHAR(30)
-> )
-> PARTITION BY RANGE(id)
-> SUBPARTITION BY KEY()
-> SUBPARTITIONS 4
-> (
->   PARTITION p0 VALUES LESS THAN (100),
->   PARTITION p1 VALUES LESS THAN (MAXVALUE)
-> );
ERROR 1064 (42000): You have an error in your SQL syntax; check the manual that
corresponds to your MySQL server version for the right syntax to use near ')'
mysql> CREATE TABLE ts (
->   id INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
->   name VARCHAR(30)
-> )
-> PARTITION BY RANGE(id)
-> SUBPARTITION BY KEY(id)
-> SUBPARTITIONS 4
-> (
->   PARTITION p0 VALUES LESS THAN (100),
->   PARTITION p1 VALUES LESS THAN (MAXVALUE)
-> );
Query OK, 0 rows affected (0.07 sec)
```

This is a known issue (see Bug #51470).

DELAYED option not supported. Use of [INSERT DELAYED](#) to insert rows into a partitioned table is not supported. Attempting to do so fails with an error.

DATA DIRECTORY and INDEX DIRECTORY options. [DATA DIRECTORY](#) and [INDEX DIRECTORY](#) are subject to the following restrictions when used with partitioned tables:

- Table-level [DATA DIRECTORY](#) and [INDEX DIRECTORY](#) options are ignored (see Bug #32091).

- On Windows, the [DATA DIRECTORY](#) and [INDEX DIRECTORY](#) options are not supported for individual partitions or subpartitions of [MyISAM](#) tables (Bug #30459). However, you can use [DATA DIRECTORY](#) for individual partitions or subpartitions of [InnoDB](#) tables.

Repairing and rebuilding partitioned tables. The statements [CHECK TABLE](#), [OPTIMIZE TABLE](#), [ANALYZE TABLE](#), and [REPAIR TABLE](#) are supported for partitioned tables.

In addition, you can use [ALTER TABLE ... REBUILD PARTITION](#) to rebuild one or more partitions of a partitioned table; [ALTER TABLE ... REORGANIZE PARTITION](#) also causes partitions to be rebuilt. See [ALTER TABLE Syntax](#), for more information about these two statements.

[mysqlcheck](#), [myisamchk](#), and [myisampack](#) are not supported with partitioned tables.

FOR EXPORT option (FLUSH TABLES). The [FLUSH TABLES](#) statement's [FOR EXPORT](#) option is not supported for partitioned [InnoDB](#) tables in MySQL 5.6.16 and earlier. (Bug #16943907)

6.1 Partitioning Keys, Primary Keys, and Unique Keys

This section discusses the relationship of partitioning keys with primary keys and unique keys. The rule governing this relationship can be expressed as follows: All columns used in the partitioning expression for a partitioned table must be part of every unique key that the table may have.

In other words, *every unique key on the table must use every column in the table's partitioning expression*. (This also includes the table's primary key, since it is by definition a unique key. This particular case is discussed later in this section.) For example, each of the following table creation statements is invalid:

```
CREATE TABLE t1 (
  col1 INT NOT NULL,
  col2 DATE NOT NULL,
  col3 INT NOT NULL,
  col4 INT NOT NULL,
  UNIQUE KEY (col1, col2)
)
PARTITION BY HASH(col3)
PARTITIONS 4;
CREATE TABLE t2 (
  col1 INT NOT NULL,
  col2 DATE NOT NULL,
  col3 INT NOT NULL,
  col4 INT NOT NULL,
  UNIQUE KEY (col1),
  UNIQUE KEY (col3)
)
PARTITION BY HASH(col1 + col3)
PARTITIONS 4;
```

In each case, the proposed table would have at least one unique key that does not include all columns used in the partitioning expression.

Each of the following statements is valid, and represents one way in which the corresponding invalid table creation statement could be made to work:

```
CREATE TABLE t1 (
  col1 INT NOT NULL,
  col2 DATE NOT NULL,
  col3 INT NOT NULL,
  col4 INT NOT NULL,
  UNIQUE KEY (col1, col2, col3)
)
PARTITION BY HASH(col3)
PARTITIONS 4;
CREATE TABLE t2 (
  col1 INT NOT NULL,
```

```
col2 DATE NOT NULL,
col3 INT NOT NULL,
col4 INT NOT NULL,
UNIQUE KEY (col1, col3)
)
PARTITION BY HASH(col1 + col3)
PARTITIONS 4;
```

This example shows the error produced in such cases:

```
mysql> CREATE TABLE t3 (
->   col1 INT NOT NULL,
->   col2 DATE NOT NULL,
->   col3 INT NOT NULL,
->   col4 INT NOT NULL,
->   UNIQUE KEY (col1, col2),
->   UNIQUE KEY (col3)
-> )
-> PARTITION BY HASH(col1 + col3)
-> PARTITIONS 4;
ERROR 1491 (HY000): A PRIMARY KEY must include all columns in the table's partitioning function
```

The `CREATE TABLE` statement fails because both `col1` and `col3` are included in the proposed partitioning key, but neither of these columns is part of both of unique keys on the table. This shows one possible fix for the invalid table definition:

```
mysql> CREATE TABLE t3 (
->   col1 INT NOT NULL,
->   col2 DATE NOT NULL,
->   col3 INT NOT NULL,
->   col4 INT NOT NULL,
->   UNIQUE KEY (col1, col2, col3),
->   UNIQUE KEY (col3)
-> )
-> PARTITION BY HASH(col3)
-> PARTITIONS 4;
Query OK, 0 rows affected (0.05 sec)
```

In this case, the proposed partitioning key `col3` is part of both unique keys, and the table creation statement succeeds.

The following table cannot be partitioned at all, because there is no way to include in a partitioning key any columns that belong to both unique keys:

```
CREATE TABLE t4 (
col1 INT NOT NULL,
col2 INT NOT NULL,
col3 INT NOT NULL,
col4 INT NOT NULL,
UNIQUE KEY (col1, col3),
UNIQUE KEY (col2, col4)
);
```

Since every primary key is by definition a unique key, this restriction also includes the table's primary key, if it has one. For example, the next two statements are invalid:

```
CREATE TABLE t5 (
col1 INT NOT NULL,
col2 DATE NOT NULL,
col3 INT NOT NULL,
col4 INT NOT NULL,
PRIMARY KEY(col1, col2)
)
PARTITION BY HASH(col3)
PARTITIONS 4;
CREATE TABLE t6 (
```

```

col1 INT NOT NULL,
col2 DATE NOT NULL,
col3 INT NOT NULL,
col4 INT NOT NULL,
PRIMARY KEY(col1, col3),
UNIQUE KEY(col2)
)
PARTITION BY HASH( YEAR(col2) )
PARTITIONS 4;

```

In both cases, the primary key does not include all columns referenced in the partitioning expression. However, both of the next two statements are valid:

```

CREATE TABLE t7 (
    col1 INT NOT NULL,
    col2 DATE NOT NULL,
    col3 INT NOT NULL,
    col4 INT NOT NULL,
    PRIMARY KEY(col1, col2)
)
PARTITION BY HASH(col1 + YEAR(col2))
PARTITIONS 4;
CREATE TABLE t8 (
    col1 INT NOT NULL,
    col2 DATE NOT NULL,
    col3 INT NOT NULL,
    col4 INT NOT NULL,
    PRIMARY KEY(col1, col2, col4),
    UNIQUE KEY(col2, col1)
)
PARTITION BY HASH(col1 + YEAR(col2))
PARTITIONS 4;

```

If a table has no unique keys—this includes having no primary key—then this restriction does not apply, and you may use any column or columns in the partitioning expression as long as the column type is compatible with the partitioning type.

For the same reason, you cannot later add a unique key to a partitioned table unless the key includes all columns used by the table's partitioning expression. Consider the partitioned table created as shown here:

```

mysql> CREATE TABLE t_no_pk (c1 INT, c2 INT)
->     PARTITION BY RANGE(c1) (
->         PARTITION p0 VALUES LESS THAN (10),
->         PARTITION p1 VALUES LESS THAN (20),
->         PARTITION p2 VALUES LESS THAN (30),
->         PARTITION p3 VALUES LESS THAN (40)
->     );
Query OK, 0 rows affected (0.12 sec)

```

It is possible to add a primary key to `t_no_pk` using either of these `ALTER TABLE` statements:

```

# possible PK
mysql> ALTER TABLE t_no_pk ADD PRIMARY KEY(c1);
Query OK, 0 rows affected (0.13 sec)
Records: 0 Duplicates: 0 Warnings: 0
# drop this PK
mysql> ALTER TABLE t_no_pk DROP PRIMARY KEY;
Query OK, 0 rows affected (0.10 sec)
Records: 0 Duplicates: 0 Warnings: 0
# use another possible PK
mysql> ALTER TABLE t_no_pk ADD PRIMARY KEY(c1, c2);
Query OK, 0 rows affected (0.12 sec)
Records: 0 Duplicates: 0 Warnings: 0
# drop this PK
mysql> ALTER TABLE t_no_pk DROP PRIMARY KEY;
Query OK, 0 rows affected (0.09 sec)

```

```
Records: 0 Duplicates: 0 Warnings: 0
```

However, the next statement fails, because `c1` is part of the partitioning key, but is not part of the proposed primary key:

```
# fails with error 1503
mysql> ALTER TABLE t_no_pk ADD PRIMARY KEY(c2);
ERROR 1503 (HY000): A PRIMARY KEY must include all columns in the table's partitioning function
```

Since `t_no_pk` has only `c1` in its partitioning expression, attempting to adding a unique key on `c2` alone fails. However, you can add a unique key that uses both `c1` and `c2`.

These rules also apply to existing nonpartitioned tables that you wish to partition using `ALTER TABLE ... PARTITION BY`. Consider a table `np_pk` created as shown here:

```
mysql> CREATE TABLE np_pk (
->   id INT NOT NULL AUTO_INCREMENT,
->   name VARCHAR(50),
->   added DATE,
->   PRIMARY KEY (id)
-> );
Query OK, 0 rows affected (0.08 sec)
```

The following `ALTER TABLE` statement fails with an error, because the `added` column is not part of any unique key in the table:

```
mysql> ALTER TABLE np_pk
->   PARTITION BY HASH( TO_DAYS(added) )
->   PARTITIONS 4;
ERROR 1503 (HY000): A PRIMARY KEY must include all columns in the table's partitioning function
```

However, this statement using the `id` column for the partitioning column is valid, as shown here:

```
mysql> ALTER TABLE np_pk
->   PARTITION BY HASH(id)
->   PARTITIONS 4;
Query OK, 0 rows affected (0.11 sec)
Records: 0 Duplicates: 0 Warnings: 0
```

In the case of `np_pk`, the only column that may be used as part of a partitioning expression is `id`; if you wish to partition this table using any other column or columns in the partitioning expression, you must first modify the table, either by adding the desired column or columns to the primary key, or by dropping the primary key altogether.

6.2 Partitioning Limitations Relating to Storage Engines

The following limitations apply to the use of storage engines with user-defined partitioning of tables.

MERGE storage engine. User-defined partitioning and the `MERGE` storage engine are not compatible. Tables using the `MERGE` storage engine cannot be partitioned. Partitioned tables cannot be merged.

FEDERATED storage engine. Partitioning of `FEDERATED` tables is not supported; it is not possible to create partitioned `FEDERATED` tables.

CSV storage engine. Partitioned tables using the `CSV` storage engine are not supported; it is not possible to create partitioned `CSV` tables.

InnoDB storage engine. `InnoDB` foreign keys and MySQL partitioning are not compatible. Partitioned `InnoDB` tables cannot have foreign key references, nor can they have columns referenced

by foreign keys. **InnoDB** tables which have or which are referenced by foreign keys cannot be partitioned.

In addition, **ALTER TABLE ... OPTIMIZE PARTITION** does not work correctly with partitioned tables that use the **InnoDB** storage engine. Use **ALTER TABLE ... REBUILD PARTITION** and **ALTER TABLE ... ANALYZE PARTITION**, instead, for such tables. For more information, see [ALTER TABLE Partition Operations](#).

User-defined partitioning and the NDB storage engine (MySQL Cluster). Partitioning by **KEY** (including **LINEAR KEY**) is the only type of partitioning supported for the **NDB** storage engine. It is not possible under normal circumstances in MySQL Cluster NDB 7.4 to create a MySQL Cluster table using any partitioning type other than **[LINEAR] KEY**, and attempting to do so fails with an error.

Exception (not for production): It is possible to override this restriction by setting the **new** system variable on MySQL Cluster SQL nodes to **ON**. If you choose to do this, you should be aware that tables using partitioning types other than **[LINEAR] KEY** are not supported in production. *In such cases, you can create and use tables with partitioning types other than **KEY** or **LINEAR KEY**, but you do this entirely at your own risk.*

The maximum number of partitions that can be defined for an **NDB** table depends on the number of data nodes and node groups in the cluster, the version of the MySQL Cluster software in use, and other factors. See [NDB and user-defined partitioning](#), for more information.

The maximum amount of fixed-size data that can be stored per partition in an **NDB** table is 16 GB.

CREATE TABLE and **ALTER TABLE** statements that would cause a user-partitioned **NDB** table not to meet either or both of the following two requirements are not permitted, and fail with an error:

1. The table must have an explicit primary key.
2. All columns listed in the table's partitioning expression must be part of the primary key.

Exception. If a user-partitioned **NDB** table is created using an empty column-list (that is, using **PARTITION BY KEY()** or **PARTITION BY LINEAR KEY()**), then no explicit primary key is required.

Upgrading partitioned tables. When performing an upgrade, tables which are partitioned by **KEY** and which use any storage engine other than **NDB** must be dumped and reloaded.

Same storage engine for all partitions. All partitions of a partitioned table must use the same storage engine and it must be the same storage engine used by the table as a whole. In addition, if one does not specify an engine on the table level, then one must do either of the following when creating or altering a partitioned table:

- Do *not* specify any engine for *any* partition or subpartition
- Specify the engine for *all* partitions or subpartitions

6.3 Partitioning Limitations Relating to Functions

This section discusses limitations in MySQL Partitioning relating specifically to functions used in partitioning expressions.

Only the MySQL functions shown in the following table are allowed in partitioning expressions.

ABS()	CEILING() (see CEILING() and FLOOR())	DAY()
DAYOFMONTH()	DAYOFWEEK()	DAYOFYEAR()
DATEDIFF()	EXTRACT() (see EXTRACT() function with WEEK specifier)	FLOOR() (see CEILING() and FLOOR())

HOUR()	MICROSECOND()	MINUTE()
MOD()	MONTH()	QUARTER()
SECOND()	TIME_TO_SEC()	TO_DAYS()
TO_SECONDS()	UNIX_TIMESTAMP() (permitted beginning with MySQL 5.6.1 and fully supported beginning with MySQL 5.6.3, with TIMESTAMP columns)	WEEKDAY()
YEAR()		YEARWEEK()

In MySQL 5.6, range optimization can be used for the [TO_DAYS\(\)](#), [TO_SECONDS\(\)](#), and [YEAR\(\)](#) functions. In addition, beginning with MySQL 5.6.3, [UNIX_TIMESTAMP\(\)](#) is treated as monotonic in partitioning expressions. See [Chapter 5, Partition Pruning](#), for more information.

CEILING() and FLOOR(). Each of these functions returns an integer only if it is passed an argument of an exact numeric type, such as one of the [INT](#) types or [DECIMAL](#). This means, for example, that the following [CREATE TABLE](#) statement fails with an error, as shown here:

```
mysql> CREATE TABLE t (c FLOAT) PARTITION BY LIST( FLOOR(c) )(
->     PARTITION p0 VALUES IN (1,3,5),
->     PARTITION p1 VALUES IN (2,4,6)
-> );
ERROR 1490 (HY000): The PARTITION function returns the wrong type
```

EXTRACT() function with WEEK specifier. The value returned by the [EXTRACT\(\)](#) function, when used as [EXTRACT\(WEEK FROM col\)](#), depends on the value of the [default_week_format](#) system variable. For this reason, beginning with MySQL 5.6.2, [EXTRACT\(\)](#) is no longer permitted as a partitioning function when it specifies the unit as [WEEK](#). (Bug #54483)

See [Mathematical Functions](#), for more information about the return types of these functions, as well as [Numeric Types](#).

6.4 Partitioning and Locking

In MySQL 5.6.5 and earlier, for storage engines such as [MyISAM](#) that actually execute table-level locks when executing DML or DDL statements, such a statement affecting a partitioned table imposed a lock on the table as a whole; that is, all partitions were locked until the statement was finished. MySQL 5.6.6 implements *partition lock pruning*, which eliminates unneeded locks in many cases. In MySQL 5.6.6 and later, most statements reading from or updating a partitioned [MyISAM](#) table cause only the effected partitions to be locked. For example, prior to MySQL 5.6.6, a [SELECT](#) from a partitioned [MyISAM](#) table caused a lock on the entire table; in MySQL 5.6.6 and later, only those partitions actually containing rows that satisfy the [SELECT](#) statement's [WHERE](#) condition are locked. This has the effect of increasing the speed and efficiency of concurrent operations on partitioned [MyISAM](#) tables. This improvement becomes particularly noticeable when working with [MyISAM](#) tables that have many (32 or more) partitions.

This change in behavior does not have any impact on statements effecting partitioned tables using storage engines such as [InnoDB](#), that employ row-level locking and do not actually perform (or need to perform) the locks prior to partition pruning.

The next few paragraphs discuss the effects of partition lock pruning for various MySQL statements on tables using storage engines that employ table-level locks.

Effects on DML statements

[SELECT](#) statements (including those containing unions or joins) now lock only those partitions that actually need to be read. This also applies to [SELECT ... PARTITION](#).

An **UPDATE** prunes locks only for tables on which no partitioning columns are updated.

REPLACE and **INSERT** now lock only those partitions having rows to be inserted or replaced. However, if an **AUTO_INCREMENT** value is generated for any partitioning column then all partitions are locked.

INSERT ... ON DUPLICATE KEY UPDATE is pruned as long as no partitioning column is updated.

INSERT ... SELECT now locks only those partitions in the source table that need to be read, although all partitions in the target table are locked.

Note

INSERT DELAYED is not supported for partitioned tables.

Locks imposed by **LOAD DATA** statements on partitioned tables cannot be pruned.

The presence of **BEFORE INSERT** or **BEFORE UPDATE** triggers using any partitioning column of a partitioned table means that locks on **INSERT** and **UPDATE** statements updating this table cannot be pruned, since the trigger can alter its values: A **BEFORE INSERT** trigger on any of the table's partitioning columns means that locks set by **INSERT** or **REPLACE** cannot be pruned, since the **BEFORE INSERT** trigger may change a row's partitioning columns before the row is inserted, forcing the row into a different partition than it would be otherwise. A **BEFORE UPDATE** trigger on a partitioning column means that locks imposed by **UPDATE** or **INSERT ... ON DUPLICATE KEY UPDATE** cannot be pruned.

Affected DDL statements

CREATE VIEW no longer causes any locks.

ALTER TABLE ... EXCHANGE PARTITION now prunes locks; only the exchanged table and the exchanged partition are locked.

ALTER TABLE ... TRUNCATE PARTITION now prunes locks; only the partitions to be emptied are locked.

ALTER TABLE statements still take metadata locks on the table level.

Other statements

LOCK TABLES cannot prune partition locks.

CALL stored_procedure(expr) supports lock pruning, but evaluating *expr* does not.

DO and **SET** statements do not support partitioning lock pruning.

