Guide to MySQL 5.1 Partitioning

Guide to MySQL 5.1 Partitioning

Abstract

This Guide covers MySQL's partitioning implementation as of MySQL 5.1.12-beta. It includes information on the following topics:

- Partitioning types supported by MySQL 5.1
- Creating and altering partitioned tables
- Managing partitions and partitioned tables
- Items to take under consideration when designing partitioned tables and writing applications that use them.
- Current restrictions and limitations on MySQL partitioning

This Guide is not intended to serve as a complete reference to the MySQL Server or client programs, nor to writing applications that use MySQL. For information about these and related subjects, you should refer to the *MySQL 5.1 Manual*.

The information in this Guide — generated on 2006-10-26 (revision: 3752) — was current for MySQL 5.1.12-beta. For the latest information, you should check the MySQL documentation available on the MySQL Web site, at http://dev.mysql.com/doc/.

Table of Contents

1. Partitioning 1
2. Overview of Partitioning in MySQL 3
3. Partition Types 6
    3.1. RANGE Partitioning 7
    3.2. LIST Partitioning 10
    3.3. HASH Partitioning 12
        3.3.1. LINEAR HASH Partitioning 14
    3.4. KEY Partitioning 15
    3.5. Subpartitioning 16
    3.6. How MySQL Partitioning Handles NULL Values 19
4. Partition Management 24
    4.1. Management of RANGE and LIST Partitions 24
    4.2. Management of HASH and KEY Partitions 30
    4.3. Maintenance of Partitions 31
    4.4. Obtaining Information About Partitions 32
5. Partition Pruning 35
6. Restrictions and Limitations on Partitioning 38
7. SQL Statements for Creating and Altering Partitioned Tables 42
    7.1. Partitioning Extensions to CREATE TABLE 42
        7.1.1. Using the *partition_options* Clause 43
        7.1.2. Using *partition_definition* Clauses 45
    7.2. Partitioning Extnesions to the ALTER TABLE Statement 47
8. The INFORMATION_SCHEMA PARTITIONS Table 50
Index 53

Chapter 1. Partitioning

This document discusses *user-defined partitioning*, as implemented in MySQL 5.1.

An overview of MySQL partitioning and partitioning concepts may be found in Chapter 2, *Overview of Partitioning in MySQL*.

MySQL supports several types of partitioning, which are discussed in Chapter 3, *Partition Types*, as well as subpartitioning, which is described in Section 3.5, "Subpartitioning".

Methods of adding, removing, and altering partitions in existing partitioned tables are covered in Chapter 4, *Partition Management*.

Syntax information for partitioning extensions to the `CREATE TABLE` and `ALTER TABLE` statements can be found in Section 7.1, "Partitioning Extensions to `CREATE TABLE`", and Section 7.2, "Partitioning Extnesions to the `ALTER TABLE` Statement".

Table maintenance commands for use with partitioned tables are discussed in Section 4.3, "Maintenance of Partitions".

The `INFORMATION_SCHEMA.PARTITIONS` table, which contains informatin about table partitions, is described in Chapter 8, *The `INFORMATION_SCHEMA PARTITIONS` Table*.

Important: Partitioned tables created with MySQL versions prior to 5.1.6 cannot be read by a 5.1.6 or later MySQL Server. In addition, the `INFORMATION_SCHEMA.TABLES` table cannot be used if such tables are present on a 5.1.6 server. Beginning with MySQL 5.1.7, a suitable warning message is generated instead, to alert the user that incompatible partitioned tables have been found by the server.

Important: If you are using partitioned tables which were created in MySQL 5.1.5 or earlier, be sure to see Changes in release 5.1.6 (01 February 2006) [http://dev.mysql.com/doc/refman/5.1/en/news-5-1-6.html] for more information and suggested workarounds *before* upgrading to MySQL 5.1.6 or later.

The partitioning implementation in MySQL 5.1 is still undergoing development. For known issues with MySQL partitioning, see Chapter 6, *Restrictions and Limitations on Partitioning*, where we have noted these.

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

Additional Resources:

- MySQL Partitioning Forum [http://forums.mysql.com/list.php?106]

  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 [http://mikaelronstrom.blogspot.com/]

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

- PlanetMySQL [http://www.planetmysql.org/]

  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 5.1 binaries are now available from http://dev.mysql.com/downloads/mysql/5.1.html. However, for the latest partitioning bugfixes and feature additions, you can obtain the source from our BitKeeper repository. To enable partitioning, you need to compile the server using the `--with-partition` option. For more information about building MySQL, see MySQL Installation Using a Source Distribution [http://dev.mysql.com/doc/refman/5.1/en/installing-source.html]. If you have problems compiling a partitioning-enabled MySQL 5.1 build, check the MySQL Partitioning Forum [http://forums.mysql.com/list.php?106] and ask for assistance there if you don't 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 5.1.

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 filesystem, 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 [http://dev.mysql.com/doc/refman/5.1/en/symbolic-links.html], for an explanation of how this is done).

Partitioning takes this notion a step further, by allowing you to distribute portions of individual tables across a filesystem 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 either an integer column value, or a function acting on one or more column values and returning an integer. The value of this expression 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 non-constant and non-random. It may not contain any queries, but may use virtually any SQL expression that is valid in MySQL, so long as that expression returns a positive integer less than `MAXVALUE` (the greatest possible positive integer). Examples of partitioning functions can be found in the discussions of partitioning types later in this chapter (see Chapter 3, *Partition Types*), as well as in the partitioning syntax descriptions given in Section 7.1, "Partitioning Extensions to `CREATE TABLE`".

This is known as *horizontal partitioning* — that is, different rows of a table may be assigned to different physical partitions. MySQL 5.1 does not support *vertical partitioning*, in which different columns of a table are assigned to different physical partitions. There are not at this time any plans to introduce vertical partitioning into MySQL 5.1.

Partitioning support is included in the `-max` releases of MySQL 5.1 (that is, the 5.1 `-max` binaries will be built with `--with-partition`). If the MySQL binary is built with partitioning support, nothing further needs to be done in order to enable it (for example, no special entries are required in your `my.cnf` file). You can determine whether your MySQL server supports partitioning by means of a `SHOW VARIABLES` command such as this one:

```
mysql> SHOW VARIABLES LIKE '%partition%';

+-------------------+-------+
| Variable_name     | Value |
+-------------------+-------+
| have_partitioning | YES   |
+-------------------+-------+
1 row in set (0.00 sec)
```

If you do not see the `have_partitioning` variable with the value `YES` listed as shown above in the output of an appropriate `SHOW VARIABLES`, then your version of MySQL does not support partitioning.

Prior to MySQL 5.1.6, this variable was named `have_partition_engine`. (Bug#16718 [http://bugs.mysql.com/16718])

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 5.1, 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.

Note: MySQL partitioning cannot be used with the `MERGE` or `CSV` storage engines. Prior to MySQL 5.1.6, it was also not feasible to create a partitioned table using the `BLACKHOLE` storage engine. (Bug#14524 [http://bugs.mysql.com/14524]). Partitioning by `KEY` is supported for use with the `NDB-Cluster` storage engine, but other types of user-defined partitioning are not supported for Cluster tables in MySQL 5.1.

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 non-partitioned 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;
```

(Note that each `PARTITION` clause can include a `[STORAGE] ENGINE` option, but in MySQL 5.1 this has no effect.)

Note: 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. 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. See Chapter 4, *Partition Management*, for more information on these options.

Some of the advantages of partitioning include:

- Being able to store more data in one table than can be held on a single disk or filesystem partition.
- Data that loses its usefulness can often be easily be removed from the table by dropping the partition containing only that data. Conversely, the process of adding new data can in some cases be greatly facilitated by adding a new partition specifically for 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, thereby excluding 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 so when the partitioning scheme was first set up. This capability, sometimes referred to as *partition pruning*, was implemented in MySQL 5.1.6. For additional information, see Chapter 5, *Partition Pruning*.

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 Partitioning development continues.

Chapter 3. Partition Types

This section discusses the types of partitioning which are available in MySQL 5.1. These include:

- **`RANGE` partitioning**: Assigns rows to partitions based on column values falling within a given range. See Section 3.1, “`RANGE` 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”.
- **`HASH` 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 non-negative integer value. See Section 3.3, “`HASH` Partitioning”.
- **`KEY` partitioning**: 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. See Section 3.4, “`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 5.1. 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;
```

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 here:

- Section 3.1, “RANGE Partitioning”
- Section 3.3, “HASH Partitioning”
- Section 3.3.1, “LINEAR HASH Partitioning”

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

- Chapter 5, *Partition Pruning*
- Section 3.5, “Subpartitioning”

MySQL partitioning is optimised for use with the TO_DAYS() and YEAR() functions. However, you can use other date and time functions that return an integer or NULL, such as WEEKDAY(), DAY-OFYEAR(), or MONTH(). See Date and Time Functions [http://dev.mysql.com/doc/refman/5.1/en/date-and-time-functions.html], 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, you should note that 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, non-zero integer literal with no leading zeroes, and may not be an expression such as 0.8E+01 or 6-2, even if it evaluates as an integer. (Beginning with MySQL 5.1.12, decimal fractions are no longer truncated, but instead are disallowed entirely.)

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 Section 7.1, “Partitioning Extensions to CREATE TABLE”.

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
);
```

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 `switch ... case` in C or Java in this regard.

It is easy to determine that a new row containing the data `(72, 'Michael', 'Widenius', '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 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
);
```

`MAXVALUE` represents the greatest possible integer value. 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:

```
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`.

Range partitioning is particularly useful when:

- You want or need to delete “old” data. If you are using the partitioning scheme shown immediately above, 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 Section 7.2, “Partitioning Extnesions to the `ALTER TABLE` Statement”, 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 `SELECT COUNT(*) FROM employees WHERE YEAR(separated) = 2000 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.

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 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 5.1, it is possible to match against only a list of integers (and possibly `NULL` — see Section 3.6, "How MySQL Partitioning Handles `NULL` Values") when partitioning by `LIST`.

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 Section 7.1, "Partitioning Extensions to `CREATE TABLE`".

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
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. All rows relating to employees working at stores in that region can be deleted with the query `ALTER TABLE employees DROP PARTITION pWest;`, which can be executed much more efficiently than the equivalent `DELETE` statement `DELETE FROM employees WHERE store_id IN (4,12,13,14,18);`.

As with `RANGE` and `HASH` partitioning, if you wish to partition a table by a column whose value is not an integer or `NULL`, you must employ a partitioning expression based on that column which returns such a value. For example, suppose that the table containing employee data is defined 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 CHAR(1),
    store_id INT
);
```

In this version of the `employees` table, the job code is a letter rather than a number. Each letter corresponds to a specific job, and we wish to partition the table in such a way that records for employees having similar jobs or working in the same department are grouped into the same partition, according to the following scheme:

Job Category or Department	**Job Codes**
Management	D, M, O, P
Sales	B, L, S
Technical	A, E, G, I, T
Clerical	K, N, Y
Support	C, F, J, R, V
Unassigned	“Empty”

Since we cannot use character values in value-lists, we need to convert these into integers or `NULL`s. For this purpose, we can use the `ASCII()` function on the column value. In addition — due to the use of different applications at different times and locations — these codes may be either uppercase or lowercase, and the “empty” value representing “currently unassigned” may actually be a `NULL`, an empty string, or a space character. A partitioned table that implements this 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 CHAR(1),
    store_id INT
)
PARTITION BY LIST(ASCII( UCASE(job_code) )) (
    PARTITION management VALUES IN(68, 77, 79, 80),
    PARTITION sales VALUES IN(66, 76, 83),
    PARTITION technical VALUES IN(65, 69, 71, 73, 84),
    PARTITION clerical VALUES IN(75, 78, 89),
    PARTITION support VALUES IN(67, 70, 74, 82, 86),
    PARTITION unassigned VALUES IN(NULL, 0, 32)
);
```

Since expressions are not permitted in partition value lists, you must list the ASCII codes for the letters that are to be matched. Note that `ASCII(NULL)` returns `NULL`.

Important: If you try to insert a row such that the column value (or the partitioning expression's return value) is not found in any of the partitioning value lists, the `INSERT` query will fail with an error. For example, given the `LIST` partitioning scheme just outlined, this query will fail:

```
INSERT INTO employees VALUES
    (224, 'Linus', 'Torvalds', '2002-05-01', '2004-10-12', 'Q', 21);
```

Failure occurs because 81 (the ASCII code for the uppercase letter `'Q'` is not found in any of the value lists used to define any of the partitions. *There is no "catch-all" definition for list partitions* analogous to `VALUES LESS THAN(MAXVALUE)` which accommodates values not found in any of the value lists. In other words, *any value which is to be matched must be found in one of the value lists*.

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 3.5, "Subpartitioning".

3.3. `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 will most likely want to follow this with a `PARTITIONS num` clause, where *`num`* is a non-negative integer representing the number of partitions into which the table is to be divided.

For example, 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;
```

You may use any function or other expression for *expr* that is valid in MySQL, so long as it returns a non-constant, non-random integer value. (In other words, it should be varying but deterministic.) However, you should 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=nx` where *n* 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 = MOD(expr, num)*. For example, suppose 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 5.1 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 3.3.1, “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 the 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. **Exception**: This restriction does not apply to tables using the NDBCluster storage engine.

3.3.1. 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)))
   ```

   (For example, suppose that *num* is 13. Then LOG(2,13) is 3.7004397181411. CEILING(3.7004397181411) is 4, and *V* = POWER(2,4), which is 16.)

2. Set *N* = *F*(*column_list*) & (*V* - 1).

3. While *N* >= *num*:

   - Set *V* = CEIL(*V* / 2)
   - Set *N* = *N* & (*V* - 1)

For example, 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(2,7) )) = 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(5 / 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.

3.4. 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 that:

- `KEY` is used rather than `HASH`.
- `KEY` takes only a list of one or more column names. Beginning with MySQL 5.1.5, the column or columns used as the partitioning key must comprise part or all of the table's primary key, if the table has one.

  Beginning with MySQL 5.1.6, `KEY` takes a list of zero or more column names. 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 5.1.6 or later:

  ```
  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. (**Note**: 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: Also beginning with MySQL 5.1.6, tables using the `NDB Cluster` storage engine are implicitly partitioned by `KEY`, again using the table's primary key as the partitioning key. In the event that the Cluster table has no explicit primary key, the “hidden” primary key generated by the `NDB` storage engine for each Cluster table is used as the partitioning key.

Important: For a key-partitioned table using any MySQL storage engine other than `NDB Cluster`, 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 [http://dev.mysql.com/doc/refman/5.1/en/mysql-cluster.html].

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 3.3.1, “`LINEAR HASH` Partitioning”, for a description of this algorithm and its implications.

3.5. Subpartitioning

Subpartitioning — also known as *composite partitioning* — is the further division of each partition in a partitioned table. For example, 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 5.1, 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*.

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:

- 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.
- In MySQL 5.1.7 and earlier, names of subpartitions must be unique within each partition, but do not have to be unique within the table as a whole. Beginning with MySQL 5.1.8, subpartition names

must be unique across the entire table. For example, the following `CREATE TABLE` statement is valid in MySQL 5.1.8 and later:

```
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
        )
    );
```

(The previous statement is also valid for versions of MySQL prior to 5.1.8.)

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.

3.6. How MySQL Partitioning Handles `NULL` Values

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. Beginning version 5.1.8, MySQL Partitioning treats `NULL` as being less than any non-`NULL` value, just as `ORDER BY` does.

Because of this, this 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 types handles `NULL` values when determining the partition in which a row should be stored, and provide examples for each.

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. For example, consider these two tables, created and populated 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 t1 (
    ->     c1 INT,
    ->     c2 VARCHAR(20)
    -> )
    -> PARTITION BY RANGE(c1) (
    ->     PARTITION p0 VALUES LESS THAN (-5),
    ->     PARTITION p1 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> 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 the rows are stored in by inspecting the filesystem and comparing the sizes of the .MYD files correpsonding to the partitions:

```
/var/lib/mysql/test> ls -l *.MYD
-rw-rw----  1 mysql mysql 20 2006-03-10 03:27 t1#P#p0.MYD
-rw-rw----  1 mysql mysql  0 2006-03-10 03:17 t1#P#p1.MYD
-rw-rw----  1 mysql mysql  0 2006-03-10 03:17 t1#P#p2.MYD
-rw-rw----  1 mysql mysql 20 2006-03-10 03:27 t2#P#p0.MYD
-rw-rw----  1 mysql mysql  0 2006-03-10 03:17 t2#P#p1.MYD
-rw-rw----  1 mysql mysql  0 2006-03-10 03:17 t2#P#p2.MYD
-rw-rw----  1 mysql mysql  0 2006-03-10 03:17 t2#P#p3.MYD
```

(Partition files are named according to the format *table_name*#P#*partition_name*.*extension*, so that t1#P#p0.MYD is the file in which data belonging to partition p0 of table t1 is stored. **Note**: Prior to MySQL 5.1.5, these files would have been named t1_p0.MYD and t2_p0.MYD, respectively. See Changes in release 5.1.6 (01 February 2006) [http://dev.mysql.com/doc/refman/5.1/en/news-5-1-6.html] and Bug#13437 [http://bugs.mysql.com/13437] for information regarding how this change impacts upgrades.)

You can also demonstrate that these rows were stored in the lowest partition of the 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 Section 7.2, "Partitioning Extnesions to the `ALTER TABLE` Statement".)

Such treatment also holds true for partitioning expressions that use SQL functions. Suppose that we have a table 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`.

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 treat `NULL` just as you would any other value, and so `VALUES IN (NULL)` and `VALUES IN (1, 4, 7, NULL)` are both 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 inspecting the filesystem, you can verify that the first of these statements inserted a new row into partition `p3` of table `ts2`, and that the second statement inserted a new row into partition `p1` of table `ts3`:

```
/var/lib/mysql/test> ls -l ts2*.MYD
-rw-rw----  1 mysql mysql  0 2006-03-10 10:35 ts2#P#p0.MYD
-rw-rw----  1 mysql mysql  0 2006-03-10 10:35 ts2#P#p1.MYD
-rw-rw----  1 mysql mysql  0 2006-03-10 10:35 ts2#P#p2.MYD
-rw-rw----  1 mysql mysql 20 2006-03-10 10:35 ts2#P#p3.MYD

/var/lib/mysql/test> ls -l ts3*.MYD
-rw-rw----  1 mysql mysql  0 2006-03-10 10:36 ts3#P#p0.MYD
-rw-rw----  1 mysql mysql 20 2006-03-10 10:36 ts3#P#p1.MYD
-rw-rw----  1 mysql mysql  0 2006-03-10 10:36 ts3#P#p2.MYD
```

As in earlier examples, we assume the use of the `bash` shell on a Unix operating system for listing files; use whatever your platform provides in this regard. For example, if you are using a DOS shell on a Windows operating system, the equivalent for the last listing might be obtained by running the command `dir ts3*.MYD` in the directory `C:\Program Files\MySQL\MySQL Server 5.1\data\test`.

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

`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 filesystem of creating a table partitioned by `HASH` and populating it with a record containing appropriate values. Suppose that you have a table `th`, created in the `test` database, using this statement:

```
mysql> CREATE TABLE th (
    ->     c1 INT,
    ->     c2 VARCHAR(20)
    -> )
    -> PARTITION BY HASH(c1)
    -> PARTITIONS 2;
Query OK, 0 rows affected (0.00 sec)
```

Assuming an RPM installation of MySQL on Linux, this statement creates two `.MYD` files in `/var/lib/mysql/test`, which can be viewed in the `bash` shell as follows:

```
/var/lib/mysql/test> ls th*.MYD -l
-rw-rw----  1 mysql mysql 0 2005-11-04 18:41 th#P#p0.MYD
-rw-rw----  1 mysql mysql 0 2005-11-04 18:41 th#P#p1.MYD
```

Note that the size of each file is 0 bytes. Now insert a row into `th` whose `c1` column value is `NULL`, and

verify that this row was inserted:

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

mysql> SELECT * FROM th;
+------+---------+
| c1   | c2      |
+------+---------+
| NULL | mothra  |
+------+---------+
1 row 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`. Returning to the system shell (still assuming `bash` for this purpose), we can see that the value was inserted into the first partition (named `p0` by default) by listing the data files once again:

```
var/lib/mysql/test> ls *.MYD -l
-rw-rw----  1 mysql mysql 20 2005-11-04 18:44 th#P#p0.MYD
-rw-rw----  1 mysql mysql  0 2005-11-04 18:41 th#P#p1.MYD
```

You can see that the `INSERT` statement modified only the file `th#P#p0.MYD` (increasing its size on disk), without affecting the other data file.

Important: Prior to MySQL 5.1.8, `RANGE` partitioning treated a partitioning expression value of `NULL` as a zero with respect to determining placement (the only way to circumvent this was to design tables so as not to allow nulls, usually by declaring columns `NOT NULL`). If you have a `RANGE` partitioning scheme that depends on this earlier behavior, you will need to re-implement it when upgrading to MySQL 5.1.8 or later.

Chapter 4. Partition Management

MySQL 5.1 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` command (see Section 7.2, “Partitioning Extnesions to the `ALTER TABLE` Statement”, for syntax definitions). 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 4.4, “Obtaining Information About Partitions”, for a discussion of mechanisms provided in MySQL 5.1 for obtaining information about partitioned tables and partitions.
- For a discussion of performing maintenance operations on partitions, see Section 4.3, “Maintenance of Partitions”.

`Note`: In MySQL 5.1, 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.

The statement `ALTER TABLE ... PARTITION BY ...` is available and is functional beginning with MySQL 5.1.6; previously in MySQL 5.1, this was accepted as valid syntax, but the statement did nothing.

To change a table's partitioning scheme, it is necessary only to use the `ALTER TABLE` command 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`. For example, 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;`.

Important: In MySQL 5.1.7 and earlier MySQL 5.1 releases, `ALTER TABLE ... ENGINE = ...` removed all partitioning from the affected table. Beginning with MySQL 5.1.8, this statement changes only the storage engine used by the table, and leaves the table's partitioning scheme intact. As of MySQL 5.1.8, use `ALTER TABLE ... REMOVE PARTITIONING` to remove a table's partitioning. See Section 7.2, “Partitioning Extnesions to the `ALTER TABLE` Statement”.

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: In MySQL 5.1, the NDB Cluster 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, the requirement was added in MySQL 5.1.10 that you 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` command. (See `TRUNCATE` Syntax [http://dev.mysql.com/doc/refman/5.1/en/truncate.html].)

If you intend to change the partitioning of a table *without* losing data, use `ALTER TABLE ... REORGANIZE PARTITION` instead. See below or in Section 7.2, "Partitioning Extnesions to the `ALTER TABLE` Statement", for information about `REORGANIZE PARTITION`.

If you now execute a `SHOW CREATE TABLE` command, 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)
```

Note that 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. For example, suppose that you have a partitioned table containing membership data for your organisation, 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 1990-1999 as shown here:

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

Important: 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 will result in an error as shown here:

```
mysql> ALTER TABLE members
     >     ADD PARTITION (
     >     PARTITION p3 VALUES LESS THAN (1960));
ERROR 1463 (HY000): VALUES LESS THAN value must be strictly »
   increasing for each partition
```

In a similar fashion, you can add new partitions to a table that is partitioned by `LIST`. For example, given a table defined like so:

```
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));
```

Note that 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.

Beginning with MySQL 5.1.6, 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` in order 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:

```
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 Section 7.1, "Partitioning Extensions to `CREATE TABLE`"). 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 non-conflicting 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`).

  Note: Prior to MySQL 5.1.4, you could not reuse the names of existing partitions in the `INTO` clause, even when those partitions were being dropped or redefined. See Changes in release 5.1.4 (21 December 2005) [http://dev.mysql.com/doc/refman/5.1/en/news-5-1-4.html], for more information.

- 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` command. For example, 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
```

Note that 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. Maintenance of Partitions

A number of partitioning maintenance tasks can be carried out in MySQL 5.1. MySQL does not support the commands `CHECK TABLE`, `OPTIMIZE TABLE`, `ANALYZE TABLE`, or `REPAIR TABLE` for partitioned tables. Instead, you can use a number of extensions to `ALTER TABLE` which were implemented in MySQL 5.1.5. These can be used 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.

- **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 non-partitioned 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.

You can also use the `mysqlcheck` or `myisamchk` utility to accomplish these tasks, operating on the separate `.MYI` files generated by partitioning a table. See mysqlcheck [http://dev.mysql.com/doc/refman/5.1/en/mysqlcheck.html].

4.4. Obtaining Information About Partitions

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

- 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)
```

Note: In early MySQL 5.1 releases, the `PARTITIONS` clause was not shown for tables partitioned by `HASH` or `KEY`. This issue was fixed in MySQL 5.1.6.

`SHOW TABLE STATUS` works with partitioned tables. Beginning with MySQL 5.1.9, its output is the same as that for non-partitioned tables, except that the `Create_options` column contains the string `partitioned`. In MySQL 5.1.8 and earlier, the `Engine` column always contained the value `PARTITION`; beginning with MySQL 5.1.9, this column contains the name of the storage engine used by all partitions of the table. (See `SHOW TABLE STATUS` Syntax [http://dev.mysql.com/doc/refman/5.1/en/show-table-status.html], for more information about this command.)

You can also obtain information about partitions from `INFORMATION_SCHEMA`, which contains a `PARTITIONS` table. See The `INFORMATION_SCHEMA PARTITIONS` Table [http://dev.mysql.com/doc/refman/5.1/en/partitions-table.html].

Beginning with MySQL 5.1.5, 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` defined 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 non-partitioned table, no error is produced, but the value of the `partitions` column is always `NULL`.

See also Optimizing Queries with `EXPLAIN` [http://dev.mysql.com/doc/refman/5.1/en/explain.html].

Chapter 5. Partition Pruning

This section discusses *partition pruning*, an opimisation which was implemented for partitioned tables in MySQL 5.1.6.

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”. For example, suppose 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 query such as this one:

```
SELECT fname, lname, postcode, 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 more time and effort in finding matching rows than it is to scan all partitions in the table. This “cutting away” of unneeded partitions is known as *pruning*. When the optimiser can make use of partition pruning in performing a query, execution of the query can be an order of magnitude faster than the same query against a non-partitioned table containing the same column definitions and data.

The query optimiser can perform pruning whenever a `WHERE` condition can be reduced to either one of the following:

- *`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 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.

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 (125, 126, 127, 128, 129, 130)`. Then the optimizer can determine that the first three values in the list are found in partition `p1`, the remaining three values in partition `p2`, and that the other partitions contain no relevant values and so do not need to be searched for matching rows.

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. It can also be employed when the partitioning expression represents an increasing or decreasing relationship or uses a function such as `YEAR()` or `TO_DAYS()` that produces an integer value when applied to a `DATE` or `DATETIME` column value. For

example, 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 queries on `t2` can make of use partition pruning:

```
SELECT * FROM t2 WHERE dob = '1982-06-23';

SELECT * FROM t2 WHERE dob BETWEEN '1991-02-15' AND '1997-04-25';

SELECT * FROM t2 WHERE YEAR(dob)
         IN (1979, 1980, 1983, 1985, 1986, 1988);

SELECT * FROM t2 WHERE dob >= '1984-06-21' AND dob <= '1999-06-21'
```

In the case of the last query, 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).

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 query such as `SELECT * FROM t3 WHERE region_code BETWEEN 1 AND 3`, the op-

timizer 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 `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;
```

Any query such as this one can be pruned:

```
SELECT * FROM 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 query:

```
SELECT * FROM t4 WHERE region_code BETWEEN 4 AND 8;
```

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

Pruning can be used only on integer columns of tables partitioned by `HASH` or `KEY`. For example, this query on table `t4` 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.

Chapter 6. Restrictions and Limitations on Partitioning

This section discusses current restrictions and limitations on MySQL partitioning support, as listed here:

- If, when creating tables with a very large number of partitions, you encounter an error message such as Got error 24 from storage engine, you may need to increase the value of the `open_files_limit` system variable. See 'File' Not Found and Similar Errors [http://dev.mysql.com/doc/refman/5.1/en/not-enough-file-handles.html].
- Partitioned tables do not support foreign keys. This includes partitioned tables employing the `InnoDB` storage engine.
- Partitioned tables do not support `FULLTEXT` indexes. This includes partitioned tables employing the `MyISAM` storage engine.
- Partitioned tables do not support `GEOMETRY` columns.
- As of MySQL 5.1.8, temporary tables cannot be partitioned. (Bug#17497 [http://bugs.mysql.com/17497])
- Tables using the `MERGE` storage engine cannot be partitioned.

  Partitioned tables using the `CSV` storage engine are not supported. Starting with MySQL 5.1.12, it is not possible to create partitioned `CSV` tables at all.

  Prior to MySQL 5.1.6, tables using the `BLACKHOLE` storage engine also could not be partitioned.

  Partitioning by `KEY` (or `LINEAR KEY`) is the only type of partitioning supported for the `NDB` storage engine. Beginning with MySQL 5.1.12, it is not possible to create a Cluster table using any partitioning type other than [`LINEAR`] `KEY`, and attempting to do so gives rise to an error.
- When performing an upgrade, tables using any storage engine other than `NDBCLUSTER` and which are partitioned by `KEY` must be dumped and reloaded.
- All of a table's partitions and subpartitions (if there are any of the latter) must use the same storage engine.
- A partitioning key must be either an integer column or an expression that resolves to an integer. The column or expression value may also be `NULL`. (See Section 3.6, “How MySQL Partitioning Handles `NULL` Values”.)

  The one exception to this restriction occurs when partitioning by [`LINEAR`] `KEY` — where it is possible to use columns of other types types as partitioning keys — because MySQL's internal key-hashing functions produce the correct datatype from these types. For example, the following `CREATE TABLE` statement is valid:

  ```
  CREATE TABLE tkc (c1 CHAR)
  PARTITION BY KEY(c1)
  PARTITIONS 4;
  ```

  This exception does *not* apply to `BLOB` or `TEXT` column types.
- A partitioning key may not be a subquery, even if that subquery resolves to an integer value or `NULL`.

- 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 tables partitioning expression. 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;

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;
```

  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;

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;
```

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 t4 (
    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 t5 (
    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 t6 (
    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 t7 (
    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 given the partitioned table defined as shown here:

```
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)
    );
```

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

```
#  possible PK
ALTER TABLE t_no_pk ADD PRIMARY KEY(c1);

#  also a possible PK
```

```
ALTER TABLE t_no_pk ADD PRIMARY KEY(c1, c2);
```

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 1482
ALTER TABLE t_no_pk ADD PRIMARY KEY(c2);
```

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 non-partitioned tables that you wish to partition using `ALTER TABLE ... PARTITION BY`. Consider a table `np_pk` defined as shown here:

```
CREATE TABLE np_pk (
    id INT NOT NULL AUTO_INCREMENT,
    name VARCHAR(50),
    added DATE,
    PRIMARY KEY (id)
);
```

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

```
ALTER TABLE np_pk
    PARTITION BY HASH( TO_DAYS(added) )
    PARTITIONS 4;
```

This statement, however, would be valid:

```
ALTER TABLE np_pk
    PARTITION BY HASH(id)
    PARTITIONS 4;
```

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.

We are working to remove this limitation in a future MySQL release series.

- Subpartitions are limited to `HASH` or `KEY` partitioning. `HASH` and `KEY` partitions cannot be subpartitioned.

Chapter 7. SQL Statements for Creating and Altering Partitioned Tables

This chapter covers extensions to the MySQL `CREATE TABLE` and `ALTER TABLE` statements that are specific to partitioned tables. For complete information about `CREATE TABLE` and `ALTER TABLE` as implemented in MySQL 5.1, see `CREATE TABLE` Syntax [http://dev.mysql.com/doc/refman/5.1/en/create-table.html], and `ALTER TABLE` Syntax [http://dev.mysql.com/doc/refman/5.1/en/alter-table.html], in the *MySQL 5.1 Manual*.

For detailed information on the types of table partitioning supported in MySQL 5.1, see Chapter 3, *Partition Types*.

For limitations on partitioned tables, see Chapter 6, *Restrictions and Limitations on Partitioning*.

7.1. Partitioning Extensions to `CREATE TABLE`

The following section covers the `CREATE TABLE` statement as it relates to partitioned tables in MySQL 5.1. It assumes that you are already familiar with `CREATE TABLE`.

```
CREATE [TEMPORARY] TABLE [IF NOT EXISTS] tbl_name
    (create_definition,...)
    [table_option ...]
    [partition_options]
```

Or:

```
CREATE [TEMPORARY] TABLE [IF NOT EXISTS] tbl_name
    [(create_definition,...)]
    [table_option ...]
    [partition_options]
    select_statement
```

```
partition_options:
    PARTITION BY
          [LINEAR] HASH(expr)
        | [LINEAR] KEY(column_list)
        | RANGE(expr)
        | LIST(expr)
    [PARTITIONS num]
    [SUBPARTITION BY
          [LINEAR] HASH(expr)
        | [LINEAR] KEY(column_list)
      [SUBPARTITIONS num]
    ]
    [(partition_definition [, partition_definition] ...)]

partition_definition:
    PARTITION partition_name
        [VALUES {LESS THAN (expr) | MAXVALUE | IN (value_list)}]
        [[STORAGE] ENGINE [=] engine_name]
        [COMMENT [=] 'comment_text' ]
        [DATA DIRECTORY [=] 'data_dir']
        [INDEX DIRECTORY [=] 'index_dir']
        [MAX_ROWS [=] max_number_of_rows]
        [MIN_ROWS [=] min_number_of_rows]
        [TABLESPACE [=] (tablespace_name)]
        [NODEGROUP [=] node_group_id]
        [(subpartition_definition [, subpartition_definition] ...)]

subpartition_definition:
    SUBPARTITION logical_name
        [[STORAGE] ENGINE [=] engine_name]
        [COMMENT [=] 'comment_text' ]
        [DATA DIRECTORY [=] 'data_dir']
        [INDEX DIRECTORY [=] 'index_dir']
        [MAX_ROWS [=] max_number_of_rows]
```

```
        [MIN_ROWS [=] min_number_of_rows]
        [TABLESPACE [=] (tablespace_name)]
        [NODEGROUP [=] node_group_id]
```

`CREATE TABLE` creates a table with the given name. Rules for allowable table names are given in Database, Table, Index, Column, and Alias Names [http://dev.mysql.com/doc/refman/5.1/en/legal-names.html].

For information about other aspects of this statement not discussed here, see `CREATE TABLE` Syntax [http://dev.mysql.com/doc/refman/5.1/en/create-table.html], in the *MySQL 5.1 Manual*.

7.1.1. Using the *`partition_options`* Clause

`partition_options` can be used to control partitioning of the table created with `CREATE TABLE`, and if used, must contain at a minimum a `PARTITION BY` clause. This clause contains the function that is used to determine the partition; the function returns an integer value ranging from 1 to *`num`*, where *`num`* is the number of partitions. The choices that are available for this function in MySQL 5.1 are shown in the following list.

Important: Not all options shown in the syntax for *`partition_options`* at the beginning of this section are available for all partitioning types. Please see the listings for the following individual types for information specific to each type, and see Partitioning [http://dev.mysql.com/doc/refman/5.1/en/partitioning.html], for more complete information about the workings of and uses for partitioning in MySQL, as well as additional examples of table creation and other statements relating to MySQL partitioning.

- `HASH(`*`expr`*`)`: Hashes one or more columns to create a key for placing and locating rows. *`expr`* is an expression using one or more table columns. This can be any legal MySQL expression (including MySQL functions) that yields a single integer value. For example, these are all valid `CREATE TABLE` statements using `PARTITION BY HASH`:

  ```
  CREATE TABLE t1 (col1 INT, col2 CHAR(5))
      PARTITION BY HASH(col1);

  CREATE TABLE t1 (col1 INT, col2 CHAR(5))
      PARTITION BY HASH( ORD(col2) );

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

  You may not use either `VALUES LESS THAN` or `VALUES IN` clauses with `PARTITION BY HASH`.

  `PARTITION BY HASH` uses the remainder of *`expr`* divided by the number of partitions (that is, the modulus). For examples and additional information, see Section 3.3, “`HASH` Partitioning”.

  The `LINEAR` keyword entails a somewhat different algorithm. In this case, the number of the partition in which a row is stored is calculated as the result of one or more logical `AND` operations. For discussion and examples of linear hashing, see Section 3.3.1, “`LINEAR HASH` Partitioning”.

- `KEY(`*`column_list`*`)`: This is similar to `HASH`, except that MySQL supplies the hashing function so as to guarantee an even data distribution. The *`column_list`* argument is simply a list of table columns. This example shows a simple table partitioned by key, with 4 partitions:

  ```
  CREATE TABLE tk (col1 INT, col2 CHAR(5), col3 DATE)
      PARTITION BY KEY(col3)
      PARTITIONS 4;
  ```

  For tables that are partitioned by key, you can employ linear partitioning by using the `LINEAR` keyword. This has the same effect as with tables that are partitioned by `HASH`. That is, the partition

number is found using the & operator rather than the modulus (see Section 3.3.1, “LINEAR HASH Partitioning”, and Section 3.4, “KEY Partitioning”, for details). This example uses linear partitioning by key to distribute data between 5 partitions:

```
CREATE TABLE tk (col1 INT, col2 CHAR(5), col3 DATE)
    PARTITION BY LINEAR KEY(col3)
    PARTITIONS 5;
```

You may not use either `VALUES LESS THAN` or `VALUES IN` clauses with `PARTITION BY KEY`.

- `RANGE`: In this case, *`expr`* shows a range of values using a set of `VALUES LESS THAN` operators. When using range partitioning, you must define at least one partition using `VALUES LESS THAN`. You cannot use `VALUES IN` with range partitioning.

  `VALUES LESS THAN` can be used with either a literal value or an expression that evaluates to a single value.

  Suppose that you have a table that you wish to partition on a column containing year values, according to the following scheme:

Partition Number:	**Years Range**:
0	1990 and earlier
1	1991 – 1994
2	1995 – 1998
3	1999 – 2002
4	2003 – 2005
5	2006 and later

  A table implementing such a partitioning scheme can be realized by the `CREATE TABLE` statement shown here:

```
CREATE TABLE t1 (
    year_col  INT,
    some_data INT
)
PARTITION BY RANGE (year_col) (
    PARTITION p0 VALUES LESS THAN (1991),
    PARTITION p1 VALUES LESS THAN (1995),
    PARTITION p2 VALUES LESS THAN (1999),
    PARTITION p3 VALUES LESS THAN (2002),
    PARTITION p4 VALUES LESS THAN (2006),
    PARTITION p5 VALUES LESS THAN MAXVALUE
);
```

  `PARTITION ... VALUES LESS THAN ...` statements work in a consecutive fashion. `VALUES LESS THAN MAXVALUE` works to specify “leftover” values that are greater than the maximum value otherwise specified.

  Note that `VALUES LESS THAN` clauses work sequentially in a manner similar to that of the `case` portions of a `switch ... case` block (as found in many programming languages such as C, Java, and PHP). That is, the clauses must be arranged in such a way that the upper limit specified in each successive `VALUES LESS THAN` is greater than that of the previous one, with the one referencing `MAXVALUE` coming last of all in the list.

- `LIST(`*`expr`*`)`: This is useful when assigning partitions based on a table column with a restricted set of possible values, such as a state or country code. In such a case, all rows pertaining to a certain state or country can be assigned to a single partition, or a partition can be reserved for a certain set of

states or countries. It is similar to RANGE, except that only VALUES IN may be used to specify allowable values for each partition.

VALUES IN is used with a list of values to be matched. For instance, you could create a partitioning scheme such as the following:

```
CREATE TABLE client_firms (
    id   INT,
    name VARCHAR(35)
)
PARTITION BY LIST (id) (
    PARTITION r0 VALUES IN (1, 5, 9, 13, 17, 21),
    PARTITION r1 VALUES IN (2, 6, 10, 14, 18, 22),
    PARTITION r2 VALUES IN (3, 7, 11, 15, 19, 23),
    PARTITION r3 VALUES IN (4, 8, 12, 16, 20, 24)
);
```

When using list partitioning, you must define at least one partition using VALUES IN. You cannot use VALUES LESS THAN with PARTITION BY LIST.

Note: Currently, the value list used with VALUES IN must consist of integer values only.

- The number of partitions may optionally be specified with a PARTITIONS *num* clause, where *num* is the number of partitions. If both this clause *and* any PARTITION clauses are used, *num* must be equal to the total number of any partitions that are declared using PARTITION clauses.

  Note: Whether or not you use a PARTITIONS clause in creating a table that is partitioned by RANGE or LIST, you must still include at least one PARTITION VALUES clause in the table definition (see below).

- A partition may optionally be divided into a number of subpartitions. This can be indicated by using the optional SUBPARTITION BY clause. Subpartitioning may be done by HASH or KEY. Either of these may be LINEAR. These work in the same way as previously described for the equivalent partitioning types. (It is not possible to subpartition by LIST or RANGE.)

  The number of subpartitions can be indicated using the SUBPARTITIONS keyword followed by an integer value.

- MySQL 5.1.12 introduces rigourous checking of the value used in a PARTITIONS or SUBPARTITIONS clause. Beginning with this version, this value must adhere to the following rules:

  - The value must be a positive, non-zero integer.
  - No leading zeroes are permitted.
  - The value must be an integer literal, and cannot not be an expression. For example, PARTITIONS 0.2E+01 is not allowed, even though 0.2E+01 evaluates to 2. (Bug#15890 [http://bugs.mysql.com/15890])

7.1.2. Using *partition_definition* Clauses

Each partition may be individually defined using a *partition_definition* clause. The individual parts making up this clause are as follows:

- PARTITION *partition_name*: This specifies a logical name for the partition.
- A VALUES clause: For range partitioning, each partition must include a VALUES LESS THAN clause; for list partitioning, you must specify a VALUES IN clause for each partition. This is used to determine which rows are to be stored in this partition. See the discussions of partitioning types in

Partitioning [http://dev.mysql.com/doc/refman/5.1/en/partitioning.html], for syntax examples.

- An optional `COMMENT` clause may be used to describe the partition. The comment must be set off in single quotes. Example:

```
COMMENT = 'Data for the years previous to 1999'
```

- `DATA DIRECTORY` and `INDEX DIRECTORY` may be used to indicate the directory where, respectively, the data and indexes for this partition are to be stored. Both the *`data_dir`* and the *`index_dir`* must be absolute system pathnames. Example:

```
CREATE TABLE th (id INT, name VARCHAR(30), adate DATE)
PARTITION BY LIST(YEAR(adate))
(
  PARTITION p1999 VALUES IN (1995, 1999, 2003)
    DATA DIRECTORY = '/var/appdata/95/data'
    INDEX DIRECTORY = '/var/appdata/95/idx',
  PARTITION p2000 VALUES IN (1996, 2000, 2004)
    DATA DIRECTORY = '/var/appdata/96/data'
    INDEX DIRECTORY = '/var/appdata/96/idx',
  PARTITION p2001 VALUES IN (1997, 2001, 2005)
    DATA DIRECTORY = '/var/appdata/97/data'
    INDEX DIRECTORY = '/var/appdata/97/idx',
  PARTITION p2000 VALUES IN (1998, 2002, 2006)
    DATA DIRECTORY = '/var/appdata/98/data'
    INDEX DIRECTORY = '/var/appdata/98/idx'
);
```

  `DATA DIRECTORY` and `INDEX DIRECTORY` behave in the same way as in the `CREATE TABLE` statement's *`table_option`* clause as used for `MyISAM` tables.

  One data directory and one index directory may be specified per partition. If left unspecified, the data and indexes are stored by default in the MySQL data directory.

- `MAX_ROWS` and `MIN_ROWS` may be used to specify, respectively, the maximum and minimum number of rows to be stored in the partition. The values for *`max_number_of_rows`* and *`min_number_of_rows`* must be positive integers. As with the table-level options with the same names, these act only as “suggestions” to the server and are not hard limits.

- The optional `TABLESPACE` clause may be used to designate a tablespace for the partition. Used for MySQL Cluster only.

- **Note**: The partitioning handler accepts a `[STORAGE] ENGINE` option for both `PARTITION` and `SUBPARTITION`. Currently, the only way in which this can be used is to set all partitions or all subpartitions to the same storage engine, and an attempt to set different storage engines for partitions or subpartitions in the same table will give rise to the error ERROR 1469 (HY000): The mix of handlers in the partitions is not allowed in this version of MySQL. We expect to lift this restriction on partitioning in a future MySQL release.

- The `NODEGROUP` option can be used to make this partition act as part of the node group identified by *`node_group_id`*. This option is applicable only to MySQL Cluster. See MySQL Cluster [http://dev.mysql.com/doc/refman/5.1/en/mysql-cluster.html], in the *MySQL 5.1 Manual*.

- The partition definition may optionally contain one or more *`subpartition_definition`* clauses. Each of these consists at a minimum of the `SUBPARTITION` *`name`*, where *`name`* is an identifier for the subpartition. Except for the replacement of the `PARTITION` keyword with `SUBPARTITION`, the syntax for a subpartition definition is identical to that for a partition definition.

  Subpartitioning must be done by `HASH` or `KEY`, and can be done only on `RANGE` or `LIST` partitions. See Section 3.5, “Subpartitioning”.

Partitions can be modified, merged, added to tables, and dropped from tables. For basic information about the MySQL statements to accomplish these tasks, see Section 7.2, “Partitioning Extnesions to the ALTER TABLE Statement”. For more detailed descriptions and examples, see Chapter 4, *Partition Management*.

7.2. Partitioning Extnesions to the ALTER TABLE Statement

A number of partitioning-related extensions to ALTER TABLE, added in MySQL versions 5.1.5 and later, are discussed in this section. ALTER TABLE options not relating directly to partitioned tables are not covered here — for these, you should refer to ALTER TABLE Syntax [http://dev.mysql.com/doc/refman/5.1/en/alter-table.html], in the *MySQL 5.1 Manual*.

```
ALTER [IGNORE] TABLE tbl_name
    ...
    | PARTITION BY partition_options
    | ADD PARTITION (partition_definition)
    | DROP PARTITION partition_names
    | COALESCE PARTITION number
    | REORGANIZE PARTITION partition_names INTO (partition_definitions)
    | ANALYZE PARTITION partition_names
    | CHECK PARTITION partition_names
    | OPTIMIZE PARTITION partition_names
    | REBUILD PARTITION partition_names
    | REPAIR PARTITION partition_names
    | REMOVE PARTITIONING
```

The options listed above can be used with partitioned tables for repartitioning tables; for adding, dropping, merging, and splitting table partitions; and for performing partitioning maintenance.

Simply using a *partition_options* clause with ALTER TABLE on a partitioned table repartitions the table according to the partitioning scheme defined by the *partition_options*. This clause always begins with PARTITION BY, and follows the same syntax and other rules as apply to the *partition_options* clause for CREATE TABLE (see Section 7.1, “Partitioning Extensions to CREATE TABLE”, for more detailed information), and can also be used to partition an existing table that is not already partitioned. For example, consider a (non-partitioned) table defined as shown here:

```
CREATE TABLE t1 (
    id INT,
    year_col INT
);
```

This table can be partitioned by HASH, using the id column as the partitioning key, into 8 partitions by means of this statement:

```
ALTER TABLE t1
    PARTITION BY HASH(id)
    PARTITIONS 8;
```

The table that results from using an ALTER TABLE ... PARTITION BY statement must follow the same rules as one created using CREATE TABLE ... PARTITION BY. This includes the rules governing the relationship between any unique keys (including any primary key) that the table might have, and the column or columns used in the partitioning expression, as discussed in Partitioning Limitations: Partitioning Keys and Unique Keys. The CREATE TABLE ... PARTITION BY rules for specifying the number of partitions also apply to ALTER TABLE ... PARTITION BY.

ALTER TABLE ... PARTITION BY became available in MySQL 5.1.6.

The *partition_definition* clause for ALTER TABLE ADD PARTITION supports the same options as the clause of the same name does for the CREATE TABLE statement clause of the same

name. (See Section 7.1, “Partitioning Extensions to CREATE TABLE”, for the syntax and description.) Suppose that you have the partitioned table created as shown here:

```
CREATE TABLE t1 (
    id INT,
    year_col INT
)
PARTITION BY RANGE (year_col) (
    PARTITION p0 VALUES LESS THAN (1991),
    PARTITION p1 VALUES LESS THAN (1995),
    PARTITION p2 VALUES LESS THAN (1999)
);
```

You can add a new partition `p3` to this table for storing values less then `2002` as follows:

```
ALTER TABLE t1 ADD PARTITION (PARTITION p3 VALUES LESS THAN (2002));
```

`DROP PARTITION` can be used to drop one or more `RANGE` or `LIST` partitions. This statement cannot be used with `HASH` or `KEY` partitions; instead, use `COALESCE PARTITION` (see below). Any data that was stored in the dropped partitions named in the *`partition_names`* list is discarded. For example, given the table `t1` defined previously, you can drop the partitions named `p0` and `p1` as shown here:

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

Note that `DROP PARTITION` does not work with tables that use the `NDB Cluster` storage engine. See Section 4.1, “Management of RANGE and LIST Partitions”, and Known Limitations of MySQL Cluster [http://dev.mysql.com/doc/refman/5.1/en/mysql-cluster-limitations.html].

`ADD PARTITION` and `DROP PARTITION` do not currently support `IF [NOT] EXISTS`. It is also not possible to rename a partition or a partitioned table. Instead, if you wish to rename a partition, you must drop and re-create the partition; if you wish to rename a partitioned table, you must instead drop all partitions, rename the table, and then add back the partitions that were dropped.

`COALESCE PARTITION` can be used with a table that is partitioned by `HASH` or `KEY` to reduce the number of partitions by *`number`*. Suppose that you have created table `t2` using the following definition:

```
CREATE TABLE t2 (
    name VARCHAR (30),
    started DATE
)
PARTITION BY HASH( YEAR(started) )
PARTITIONS 6;
```

You can reduce the number of partitions used by `t2` from 6 to 4 using the following statement:

```
ALTER TABLE t2 COALESCE PARTITION 2;
```

The data contained in the last *`number`* partitions will be merged into the remaining partitions. In this case, partitions 4 and 5 will be merged into the first 4 partitions (the partitions numbered 0, 1, 2, and 3.

To change some but not all the partitions used by a partitioned table, you can use `REORGANIZE PARTITION`. This statement can be used in several ways:

- To merge a set of partitions into a single partition. This can be done by naming several partitions in the *`partition_names`* list and supplying a single definition for *`partition_definition`*.
- To split an existing partition into several partitions. You can accomplish this by naming a single partition for *`partition_names`* and providing multiple *`partition_definitions`*.

- To change the ranges for a subset of partitions defined using `VALUES LESS THAN` or the value lists for a subset of partitions defined using `VALUES IN`.

Note: For partitions that have not been explicitly named, MySQL automatically provides the default names `p0`, `p1`, `p2`, and so on. As of MySQL 5.1.7, the same is true with regard to subpartitions.

For more detailed information about and examples of `ALTER TABLE ... REORGANIZE PARTITION` statements, see Chapter 4, *Partition Management*.

Several additional clauses provide partition maintenance and repair functionality analogous to that implemented for non-partitioned tables by statements such as `CHECK TABLE` and `REPAIR TABLE` (which are *not* supported for partitioned tables). These include `ANALYZE PARTITION`, `CHECK PARTITION`, `OPTIMIZE PARTITION`, `REBUILD PARTITION`, and `REPAIR PARTITION`. Each of these options takes a *`partition_names`* clause consisting of one or more names of partitions, separated by commas. The partitions must already exist in the table to be altered. For more information, and for examples of these, see Section 4.3, “Maintenance of Partitions”.

`REMOVE PARTITIONING` was introduced in MySQL 5.1.8 for the purpose of removing a table's partitioning without otherwise affecting the table or its data. (Previously. this was done using the `ENGINE` option.) This option can be combined with other `ALTER TABLE` options such as those used to add, drop, or rename drop columns or indexes.

In MySQL 5.1.7 and earlier, using the `ENGINE` option with `ALTER TABLE` caused any partitioning that a table might have had to be removed. Beginning with MySQL 5.1.8, this option merely changes the storage engine used by the table and no longer affects partitioning in any way.

Chapter 8. The INFORMATION_SCHEMA PARTITIONS Table

The PARTITIONS table provides information about table partitions.

INFORMATION_SCHEMA Name	SHOW Name	Remarks
TABLE_CATALOG		MySQL extension
TABLE_SCHEMA		MySQL extension
TABLE_NAME		MySQL extension
PARTITION_NAME		MySQL extension
SUBPARTITION_NAME		MySQL extension
PARTITION_ORDINAL_POSITION		MySQL extension
SUBPARTI- TION_ORDINAL_POSITION		MySQL extension
PARTITION_METHOD		MySQL extension
SUBPARTITION_METHOD		MySQL extension
PARTITION_EXPRESSION		MySQL extension
SUBPARTITION_EXPRESSION		MySQL extension
PARTITION_DESCRIPTION		MySQL extension
TABLE_ROWS		MySQL extension
AVG_ROW_LENGTH		MySQL extension
DATA_LENGTH		MySQL extension
MAX_DATA_LENGTH		MySQL extension
INDEX_LENGTH		MySQL extension
DATA_FREE		MySQL extension
CREATE_TIME		MySQL extension
UPDATE_TIME		MySQL extension
CHECK_TIME		MySQL extension
CHECKSUM		MySQL extension
PARTITION_COMMENT		MySQL extension
NODEGROUP		MySQL extension
TABLESPACE_NAME		MySQL extension

Notes:

- The PARTITIONS table is a non-standard table. It was added in MySQL 5.1.6.

  Each record in this table corresponds to an individual partition or subpartition of a partitioned table.

- TABLE_CATALOG: This column is always NULL.
- TABLE_SCHEMA: This column contains the name of the database to which the table belongs.
- TABLE_NAME: This column contains the name of the table containing the partition.

- PARTITION_NAME: The name of the partition.
- SUBPARTITION_NAME: If the PARTITIONS table record represents a subpartition, then this column contains the name of subpartition; otherwise it is NULL.
- PARTITION_ORDINAL_POSITION: All partitions are indexed in the same order as they are defined, with 1 being the number assigned to the first partition. The indexing can change as partitions are added, dropped, and reorganized; the number shown is this column reflects the current order, taking into account any indexing changes.
- SUBPARTITION_ORDINAL_POSITION: Subpartitions within a given partition are also indexed and reindexed in the same manner as partitions are indexed within a table.
- PARTITION_METHOD: One of the values RANGE, LIST, HASH, LINEAR HASH, KEY, or LINEAR KEY; that is, one of the available partitioning types as discussed in Chapter 3, *Partition Types*.
- SUBPARTITION_METHOD: One of the values HASH, LINEAR HASH, KEY, or LINEAR KEY; that is, one of the available subpartitioning types as discussed in Section 3.5, “Subpartitioning”.
- PARTITION_EXPRESSION: This is the expression for the partitioning function used in the CREATE TABLE or ALTER TABLE statement that created the table's current partitioning scheme.

  For example, consider a partitioned table created in the test database using this statement:

```
CREATE TABLE tp (
    c1 INT,
    c2 INT,
    c3 VARCHAR(25)
)
PARTITION BY HASH(c1 + c2)
PARTITIONS 4;
```

  The PARTITION_EXPRESSION column in a PARTITIONS table record for a partition from this table displays c1 + c2, as shown here:

```
mysql> SELECT DISTINCT PARTITION_EXPRESSION
     >     FROM INFORMATION_SCHEMA.PARTITIONS
     >     WHERE TABLE_NAME='tp' AND TABLE_SCHEMA='test';
+----------------------+
| PARTITION_EXPRESSION |
+----------------------+
| c1 + c2              |
+----------------------+
1 row in set (0.09 sec)
```

- SUBPARTITION_EXPRESSION: This works in the same fashion for the subpartitioning expression that defines the subpartitioning for a table as PARTITION_EXPRESSION does for the partitioning expression used to define a table's partitioning.

  If the table has no subpartitions, then this column is NULL.
- PARTITION_DESCRIPTION: This column is used for RANGE and LIST partitions. For a RANGE partition, it contains the value set in the partition's VALUES LESS THAN clause, which can be either an integer or MAXVALUE. For a LIST partition, this column contains the values defined in the partition's VALUES IN clause, which is a comma-separated list of integer values.

  For partitions whose PARTITION_METHOD is other than RANGE or LIST, this column is always NULL.
- TABLE_ROWS: The number of table rows in the partition.
- AVG_ROW_LENGTH: The average length of the rows stored in this partition or subpartition, in bytes.

This is the same as DATA_LENGTH divided by TABLE_ROWS.

- DATA_LENGTH: The total length of all rows stored in this partition or subpartition, in bytes — that is, the total number of bytes stored in the partition or subpartition.
- MAX_DATA_LENGTH: The maximum number of bytes that can be stored in this partition or subpartition.
- INDEX_LENGTH: The length of the index file for this partition or subpartition, in bytes.
- DATA_FREE: The number of bytes allocated to the partition or subpartition but not used.
- CREATE_TIME: The time of the partition's or subpartition's creation.
- UPDATE_TIME: The time that the partition or subpartition was last modified.
- CHECK_TIME: The last time that the table to which this partition or subpartition belongs was checked.

  Note: Some storage engines do not update this time; for tables using these storage engines, this value is always NULL.
- CHECKSUM: The checksum value, if any; otherwise, this column is NULL.
- PARTITION_COMMENT: This column contains the text of any comment made for the partition.

  The default value for this column is an empty string.
- NODEGROUP: This is the nodegroup to which the partition belongs. This is relevant only to MySQL Cluster tables; otherwise the value of this column is always 0.
- TABLESPACE_NAME: This column contains the name of tablespace to which the partition belongs. In MySQL 5.1, the value of this column is always DEFAULT.
- **Important**: If any partitioned tables created in a MySQL version prior to MySQL 5.1.6 are present following an upgrade to MySQL 5.1.6 or later, it is not possible to SELECT from, SHOW, or DESCRIBE the PARTITIONS table. See Changes in release 5.1.6 (01 February 2006) [http://dev.mysql.com/doc/refman/5.1/en/news-5-1-6.html] *before* upgrading from MySQL 5.1.5 or earlier to MySQL 5.1.6 or later.
- A non-partitioned table has one record in INFORMATION_SCHEMA.PARTITIONS; however, the values of the PARTITION_NAME, SUBPARTITION_NAME, PARTITION_ORDINAL_POSITION, SUBPARTITION_ORDINAL_POSITION, PARTITION_METHOD, SUBPARTITION_METHOD, PARTITION_EXPRESSION, SUBPARTITION_EXPRESSION, and PARTITION_DESCRIPTION columns are all NULL. (The PARTITION_COMMENT column in this case is blank.)

  In MySQL 5.1, there is also only one record in the PARTITIONS table for a table using the NDBCluster storage engine. The same columns are also NULL (or empty) as for a non-partitioned table.

Index

A
ALTER TABLE, 47

C
changing
  table, 47
composite partitioning, 17
CREATE TABLE, 42

D
dates
  used with partitioning, 6
  used with partitioning (examples), 9, 12, 17, 35

E
EXPLAIN PARTITIONS, 32, 32
EXPLAIN used with partitioned tables, 32

H
hash partitioning, 12
hash partitions
  managing, 30
  splitting and merging, 30

K
key partitioning, 15
key partitions
  managing, 30
  splitting and merging, 30

L
linear hash partitioning, 14
linear key partitioning, 16
list partitioning, 10
list partitions
  adding and dropping, 24
  managing, 24

O
obtaining information about partitions, 32

P
PARTITION, 1
partition management, 24
partition pruning, 35
partitioning, 1
  advantages, 4
  and dates, 6
  by hash, 12
  by key, 15
  by linear hash, 14
  by linear key, 16
  by list, 10
  by range, 7
  concepts, 3
  enabling, 3
  limitations, 38
  optimization, 32, 35
  resources, 1
  storage engines (limitations), 38
  support, 3
  types, 6
partitioning information statements, 32
partitioning keys and primary keys, 39
partitioning keys and unique keys, 39
partitions
  adding and dropping, 24
  analyzing, 31
  checking, 31
  managing, 24
  modifying, 24
  optimizing, 31
  repairing, 31
  splitting and merging, 24
PARTITIONS
  INFORMATION_SCHEMA table, 50

R
range partitioning, 7
range partitions
  adding and dropping, 24
  managing, 24

S
subpartitioning, 16
subpartitions, 16

T
table
  changing, 47

U
unique keys
  and partitioning keys, 39, 39