MySQL Partitioning

MySQL Partitioning

Abstract

This is the MySQL Partitioning extract from the MySQL 5.1 Reference Manual.

Document generated on: 2009-06-02 (revision: 15165)

For more information on the terms of this license, for details on how the MySQL documentation is built and produced, or if you are interested in doing a translation, please contact the Documentation Team.

For additional licensing information, including licenses for libraries used by MySQL, see Preface, Notes, Licenses.

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

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

Partitioning

This chapter discusses MySQL's implementation of *user-defined partitioning*. 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)
```

> **Note**
>
> Prior to MySQL 5.1.6, this variable was named `have_partition_engine`. (Bug#16718)

You can also check the output of the `SHOW PLUGINS` statement, as shown here:

```
mysql> SHOW PLUGINS;
+------------+----------+----------------+---------+---------+
| Name       | Status   | Type           | Library | License |
+------------+----------+----------------+---------+---------+
| binlog     | ACTIVE   | STORAGE ENGINE | NULL    | GPL     |
| partition  | ACTIVE   | STORAGE ENGINE | NULL    | GPL     |
| ARCHIVE    | ACTIVE   | STORAGE ENGINE | NULL    | GPL     |
| BLACKHOLE  | ACTIVE   | STORAGE ENGINE | NULL    | GPL     |
| CSV        | ACTIVE   | STORAGE ENGINE | NULL    | GPL     |
| FEDERATED  | DISABLED | STORAGE ENGINE | NULL    | GPL     |
| MEMORY     | ACTIVE   | STORAGE ENGINE | NULL    | GPL     |
| InnoDB     | ACTIVE   | STORAGE ENGINE | NULL    | GPL     |
| MRG_MYISAM | ACTIVE   | STORAGE ENGINE | NULL    | GPL     |
| MyISAM     | ACTIVE   | STORAGE ENGINE | NULL    | GPL     |
| ndbcluster | DISABLED | STORAGE ENGINE | NULL    | GPL     |
+------------+----------+----------------+---------+---------+
11 rows in set (0.00 sec)
```

If you do not see the `have_partitioning` variable with the value `YES` listed in the output of an appropriate `SHOW VARI-ABLES` statement, or if you do not see the `partition` plugin listed with the value `ACTIVE` for the `Status` column in the output of `SHOW PLUGINS` (show in bold text in the example just given), then your version of MySQL does not support partitioning.

MySQL Community binaries provided by Sun Microsystems include partitioning support. For information about partitioning support offered in commercial MySQL Server binaries, see *MySQL Enterprise Server 5.1*, on the MySQL website.

If you are compiling MySQL 5.1 from source, the build must be configured using `--with-partition` to enable partitioning.

If your 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).

An introduction to partitioning and partitioning concepts may be found in Chapter 1, *Overview of Partitioning in MySQL*.

MySQL supports several types of partitioning, which are discussed in Chapter 2, *Partition Types*, as well as subpartitioning, which is described in Section 2.5, "Subpartitioning".

Methods of adding, removing, and altering partitions in existing partitioned tables are covered in Chapter 3, *Partition Management*.

Table maintenance commands for use with partitioned tables are discussed in Section 3.3, "Maintenance of Partitions".

Beginning with MySQL 5.1.6, the `PARTITIONS` table in the `INFORMATION_SCHEMA` database provides information about partitions and partitioned tables. See The `INFORMATION_SCHEMA PARTITIONS` Table, for more information; for some examples of queries against this table, see Section 2.6, "How MySQL Partitioning Handles `NULL`".

> **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.
>
> If you are using partitioned tables which were created in MySQL 5.1.5 or earlier, be sure to see Changes in MySQL 5.1.6 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 5, *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. Other sources of information about user-defined partitioning in MySQL include the following:

- MySQL Partitioning Forum

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

- Mikael Ronstrõm's Blog

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

- PlanetMySQL

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

MySQL 5.1 binaries are available from http://dev.mysql.com/downloads/mysql/5.1.html. However, for the latest partitioning bug-fixes and feature additions, you can obtain the source from our Bazaar 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. If you have problems compiling a partitioning-enabled MySQL 5.1 build, check the MySQL Partitioning Forum and ask for assistance there if you do not find a solution to your problem already posted.

Chapter 1. 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 5, *Restrictions and Limitations on Partitioning*.

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

Partitioning takes this notion a step further, by allowing you to distribute portions of individual tables across a file system according to rules which you can set largely as needed. In effect, different portions of a table are stored as separate tables in different locations. The user-selected rule by which the division of data is accomplished is known as a *partitioning function*, which in MySQL can be the modulus, simple matching against a set of ranges or value lists, an internal hashing function, or a linear hashing function. The function is selected according to the partitioning type specified by the user, and takes as its parameter the value of a user-supplied expression. This expression can be 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 an SQL expression that is valid in MySQL, as long as that expression returns either `NULL` or an integer `intval` such that

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

(`MAXVALUE` is used to represent the least upper bound for the type of integer in question. `-MAXVALUE` represents the greatest lower bound.) There are some additional restrictions on partitioning functions; see Chapter 5, *Restrictions and Limitations on Partitioning*, for more information about these.

Examples of partitioning functions can be found in the discussions of partitioning types later in this chapter (see Chapter 2, *Partition Types*), as well as in the partitioning syntax descriptions given in `CREATE TABLE` Syntax.

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.

For information about determining whether your MySQL Server binary supports user-defined partitioning, see Partitioning.

For creating partitioned tables, you can use most storage engines that are supported by your MySQL server; the MySQL partitioning engine runs in a separate layer and can interact with any of these. In MySQL 5.1, all partitions of the same partitioned table must use the same storage engine; for example, Â 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. Beginning with MySQL 5.1.15, `FEDERATED` tables also cannot be partitioned (Bug#22451). Prior to MySQL 5.1.6, it was also not feasible to create a partitioned table using the `BLACKHOLE` storage engine (Bug#14524).
>
> Partitioning by `KEY` or `LINEAR KEY` is possible with `NDBCLUSTER`, but other types of user-defined partitioning are not supported for tables using this storage engine. In addition, an `NDBCLUSTER` table that employs user-defined partitioning must have an explicit primary key, and any columns referenced in the table's partitioning expression must be part of the primary key. However, if no columns are listed in the `PARTITION BY KEY` or `PARTITION BY LINEAR KEY` clause of the `CREATE TABLE` or `ALTER TABLE` statement used to create or modify a user-partitioned `NDBCLUSTER` table, then the table is not required to have an explicit primary key.
>
> For more information, see Section 5.2, "Partitioning Limitations Relating to Storage Engines", and Non-Compliance with SQL Syntax in MySQL Cluster.

To employ a particular storage engine for a partitioned table, it is necessary only to use the `[STORAGE] ENGINE` option just as you would for a 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

Each `PARTITION` clause can include a `[STORAGE] ENGINE` option, but in MySQL 5.1 this has no effect.

Important

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

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

Note

Prior to MySQL 5.1.18, these options were permitted even when the `NO_DIR_IN_CREATE` server SQL mode was in effect. (Bug#24633)

The `DATA DIRECTORY` and `INDEX DIRECTORY` options have no effect when defining partitions for tables using the `InnoDB` storage engine.

`DATA DIRECTORY` and `INDEX DIRECTORY` are not supported for individual partitions or subpartitions on Windows. Beginning with MySQL 5.1.24, these options are ignored on Windows, except that a warning is generated. (Bug#30459)

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 3, *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 file system 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 more information, see Chapter 4, *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 2. 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 2.1, "`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 2.2, "`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 2.3, "`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 2.4, "`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 2.1, "`RANGE` Partitioning"

- Section 2.3, "`HASH` Partitioning"

- Section 2.3.1, "`LINEAR HASH` Partitioning"

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

- Chapter 4, *Partition Pruning*

- Section 2.5, "Subpartitioning"

MySQL partitioning is optimized 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()`, `DAYOFYEAR()`, or `MONTH()`. See Date and Time Functions, for more information about such functions.

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

Names of partitions generally follow the rules governing other MySQL identifiers, such as those for tables and databases. However, 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, nonzero integer literal with no leading zeroes, and may not be an expression such as `0.8E+01` or `6-2`, even if it evaluates to an integer value. (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 `CREATE TABLE` Syntax.

2.1. `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 series of `if ... elseif ...` statements in C or Java in this regard.

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

> **Note**
>
> Another way to avoid an error when no matching value is found is to use the IGNORE keyword as part of the IN-SERT statement. For an example, see Section 2.2, "LIST Partitioning". Also see INSERT Syntax, for general information about IGNORE.

MAXVALUE represents an integer value that is always greater than the largest possible integer value (in mathematical language, it serves as a *least upper bound*). Now, any rows whose store_id column value is greater than or equal to 16 (the highest value defined) are stored in partition p3. At some point in the future — when the number of stores has increased to 25, 30, or more — you can use an ALTER TABLE statement to add new partitions for stores 21-25, 26-30, and so on (see Chapter 3, *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 `ALTER TABLE` Syntax, and Chapter 3, *Partition Management*, for more information.) For a table with a great many rows, this can be much more efficient than running a `DELETE` query such as `DELETE FROM employees WHERE YEAR(separated) <= 1990;`.

- You want to use a column containing date or time values, or containing values arising from some other series.

- You frequently run queries that depend directly on the column used for partitioning the table. For example, when executing a query such as `EXPLAIN PARTITIONS SELECT COUNT(*) FROM employees WHERE separated BETWEEN '2000-01-01' AND '2000-12-31' GROUP BY store_id;`, MySQL can quickly determine that only partition `p2` needs to be scanned because the remaining partitions cannot contain any records satisfying the `WHERE` clause. See Chapter 4, *Partition Pruning*, for more information about how this is accomplished.

2.2. `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 2.6, "How MySQL Partitioning Handles `NULL`") 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 `CREATE TABLE` Syntax.

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

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

(This is the same table used as a basis for the examples in Section 2.1, "`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 partitioning, it is possible to combine LIST partitioning with partitioning by hash or key to produce a composite partitioning (subpartitioning). See Section 2.5, "Subpartitioning".

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

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

When inserting multiple rows using a single INSERT statement, any rows coming before the row containing the unmatched value are inserted, but any coming after it are not:

```
mysql> SELECT * FROM h2;
Empty set (0.00 sec)
mysql> INSERT INTO h2 VALUES (4, 7), (3, 5), (6, 0);
ERROR 1525 (HY000): TABLE HAS NO PARTITION FOR VALUE 3
mysql> SELECT * FROM h2;
+------+------+
| c1   | c2   |
+------+------+
|    4 |    7 |
+------+------+
1 row in set (0.00 sec)
```

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

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

2.3. 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 positive 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;
```

`expr` must return a non-constant, non-random integer value (in other words, it should be varying but deterministic), and must not contain any prohibited constructs as described in Chapter 5, *Restrictions and Limitations on Partitioning*. You should also keep in mind that this expression is evaluated each time a row is inserted or updated (or possibly deleted); this means that very complex expressions may give rise to performance issues, particularly when performing operations (such as batch inserts) that affect a great many rows at one time.

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

For example, where `date_col` is a column of type `DATE`, then the expression `TO_DAYS(date_col)` is said to vary directly with the value of `date_col`, because for every change in the value of `date_col`, the value of the expression changes in a consistent manner. The variance of the expression `YEAR(date_col)` with respect to `date_col` is not quite as direct as that of `TO_DAYS(date_col)`, because not every possible change in `date_col` produces an equivalent change in `YEAR(date_col)`. Even so, `YEAR(date_col)` is a good candidate for a hashing function, because it varies directly with a portion of `date_col` and there is no possible change in `date_col` that produces a disproportionate change in `YEAR(date_col)`.

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

In other words, the more closely the graph of the column value *versus* the value of the expression follows a straight line as traced by the equation $y=nx$ 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 2.3.1, "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 a table to be partitioned has a UNIQUE key, then any columns supplied as arguments to the HASH user function or to the KEY's column_list must be part of that key.

2.3.1. 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 $V = \text{CEIL}(V / 2)$

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

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

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

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

2.4. `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 `PASS-WORD()`.

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 5, *Restrictions and Limitations on Partitioning*.

  > **Note**
  >
  > Also beginning with MySQL 5.1.6, tables using the `NDBCLUSTER` storage engine are implicitly partitioned by `KEY`, again using the table's primary key as the partitioning key. In the event that the MySQL Cluster table has no explicit primary key, the "hidden" primary key generated by the `NDBCLUSTER` storage engine for each MySQL Cluster table is used as the partitioning key.
  >
  > Beginning with MySQL Cluster NDB 6.2.18, MySQL Cluster NDB 6.3.25, and MySQL Cluster NDB 7.0.6, if you

define an explicit partitioning scheme for an `NDBCLUSTER` table, the table must have an explicit primary key, and any columns used in the partitioning expression must be part of this key. However, if the table uses an "empty" partitioning expression — that is, `PARTITION BY KEY()` with no column references — then no explicit primary key is required.

Important

For a key-partitioned table using any MySQL storage engine other than `NDBCLUSTER`, you cannot execute an `ALTER TABLE DROP PRIMARY KEY`, as doing so generates the error `ERROR 1466 (HY000): FIELD IN LIST OF FIELDS FOR PARTITION FUNCTION NOT FOUND IN TABLE`. This is not an issue for MySQL Cluster tables which are partitioned by `KEY`; in such cases, the table is reorganized using the "hidden" primary key as the table's new partitioning key. See MySQL Cluster NDB 6.X/7.X.

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 2.3.1, "`LINEAR HASH` Partitioning", for a description of this algorithm and its implications.

2.5. 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 were required to be unique within each partition, but did 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 3, *Partition Management*, for an explanation of how this can be done.

Beginning with MySQL 5.1.18, the `DATA DIRECTORY` and `INDEX DIRECTORY` options are disallowed when the `NO_DIR_IN_CREATE` server SQL mode is in effect. This is true for partitions and subpartitions.

2.6. How MySQL Partitioning Handles `NULL`

Partitioning in MySQL does nothing to disallow `NULL` as the value of a partitioning expression, whether it is a column value or the value of a user-supplied expression. Even though it is permitted to use `NULL` as the value of an expression that must otherwise yield an integer, it is important to keep in mind that `NULL` is not a number. Beginning with MySQL 5.1.8, the partitioning implementation treats `NULL` as being less than any non-`NULL` value, just as `ORDER BY` does.

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

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

```
mysql> CREATE TABLE t1 (
    ->     c1 INT,
    ->     c2 VARCHAR(20)
    -> )
    -> PARTITION BY RANGE(c1) (
    ->     PARTITION p0 VALUES LESS THAN (0),
    ->     PARTITION p1 VALUES LESS THAN (10),
    ->     PARTITION p2 VALUES LESS THAN MAXVALUE
    -> );
```

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

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

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

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

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

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

```
mysql> SELECT TABLE_NAME, PARTITION_NAME, TABLE_ROWS, AVG_ROW_LENGTH, DATA_LENGTH
    >    FROM INFORMATION_SCHEMA.PARTITIONS
    >    WHERE TABLE_SCHEMA = 'p' AND TABLE_NAME LIKE 't_';
+------------+----------------+------------+----------------+-------------+
| TABLE_NAME | PARTITION_NAME | TABLE_ROWS | AVG_ROW_LENGTH | DATA_LENGTH |
+------------+----------------+------------+----------------+-------------+
| t1         | p0             |          1 |             20 |          20 |
| t1         | p1             |          0 |              0 |           0 |
| t1         | p2             |          0 |              0 |           0 |
| t2         | p0             |          1 |             20 |          20 |
| t2         | p1             |          0 |              0 |           0 |
| t2         | p2             |          0 |              0 |           0 |
| t2         | p3             |          0 |              0 |           0 |
+------------+----------------+------------+----------------+-------------+
7 rows in set (0.01 sec)
```

You can also demonstrate that these rows were stored in the lowest partition of each table by dropping these partitions, and then rerunning the SELECT statements:

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

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

> **Important**
>
> Prior to MySQL 5.1.8, RANGE partitioning treated a partitioning expression value of NULL as 0 with respect to de-

termining placement. (The only way to circumvent this behavior 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 must re-implement it when upgrading to MySQL 5.1.8 or later. (Bug#15447)

NULL is also treated in this way for partitioning expressions that use SQL functions. Suppose that we define a table using a CRE-ATE TABLE statement such as this one:

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

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

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

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

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

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

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

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

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

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

TABLE_NAME	PARTITION_NAME	TABLE_ROWS	AVG_ROW_LENGTH	DATA_LENGTH
ts2	p0	0	0	0
ts2	p1	0	0	0
ts2	p2	0	0	0
ts2	p3	1	20	20
ts3	p0	0	0	0
ts3	p1	1	20	20
ts3	p2	0	0	0

```
+-----------+---------------+-----------+----------------+-------------+
7 rows in set (0.01 sec)
```

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

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

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

The partitions belonging to this table can be viewed like this:

```
mysql> SELECT TABLE_NAME,PARTITION_NAME,TABLE_ROWS,AVG_ROW_LENGTH,DATA_LENGTH
    >    FROM INFORMATION_SCHEMA.PARTITIONS
    >    WHERE TABLE_SCHEMA = 'p' AND TABLE_NAME ='th';
+-----------+---------------+-----------+----------------+-------------+
| TABLE_NAME | PARTITION_NAME | TABLE_ROWS | AVG_ROW_LENGTH | DATA_LENGTH |
+-----------+---------------+-----------+----------------+-------------+
|  th       | p0            |         0 |              0 |           0 |
|  th       | p1            |         0 |              0 |           0 |
+-----------+---------------+-----------+----------------+-------------+
2 rows in set (0.00 sec)
```

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

```
mysql> INSERT INTO th VALUES (NULL, 'mothra'), (0, 'gigan');
Query OK, 1 row affected (0.00 sec)
mysql> SELECT * FROM th;
+------+---------+
| c1   | c2      |
+------+---------+
| NULL | mothra  |
+------+---------+
|    0 | gigan   |
+------+---------+
2 rows in set (0.01 sec)
```

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

```
mysql> SELECT TABLE_NAME, PARTITION_NAME, TABLE_ROWS, AVG_ROW_LENGTH, DATA_LENGTH
    >    FROM INFORMATION_SCHEMA.PARTITIONS
    >    WHERE TABLE_SCHEMA = 'p' AND TABLE_NAME ='th';
+-----------+---------------+-----------+----------------+-------------+
| TABLE_NAME | PARTITION_NAME | TABLE_ROWS | AVG_ROW_LENGTH | DATA_LENGTH |
+-----------+---------------+-----------+----------------+-------------+
|  th       | p0            |         2 |             20 |          20 |
|  th       | p1            |         0 |              0 |           0 |
+-----------+---------------+-----------+----------------+-------------+
2 rows in set (0.00 sec)
```

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

Chapter 3. 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 `ALTER TABLE Syntax`, 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 3.1, "Management of `RANGE` and `LIST` Partitions".

- For a discussion of managing `HASH` and `KEY` partitions, see Section 3.2, "Management of `HASH` and `KEY` Partitions".

- See Section 3.4, "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 3.3, "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;`.

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 `ALTER TABLE Syntax`.

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

3.1. 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 3.2, "Management of `HASH` and `KEY` Partitions". Dropping a `RANGE` or `LIST` partition is more straightforward than adding one, so we discuss this first.

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

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

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

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

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

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

> **Note**
>
> The NDBCLUSTER 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.)

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

If you now execute a SHOW CREATE TABLE 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 2.2, "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 `CREATE TABLE` Syntax). It should be noted that you are not limited to merging several partitions into one, or to splitting one partition into many, when using `REORGANIZE PARTITION`. For example, you can reorganize all four partitions of the `members` table into two, as follows:

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

You can also use `REORGANIZE PARTITION` with tables that are partitioned by `LIST`. Let us return to the problem of adding a new partition to the list-partitioned `tt` table and failing because the new partition had a value that was already present in the value-list of one of the existing partitions. We can handle this by adding a partition that contains only 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 MySQL 5.1.4, 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;
```

3.2. 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 3.1, "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;
```

3.3. Maintenance of Partitions

A number of table and partition maintenance tasks can be carried out using SQL statements intended for such purposes on partitioned tables in MySQL 5.1.

Table maintenance of partitioned tables can be accomplished using the statements CHECK TABLE, OPTIMIZE TABLE, ANALYZE TABLE, and REPAIR TABLE, which are supported for partitioned tables as of MySQL 5.1.27.

Also beginning with MySQL 5.1.27, you can use a number of extensions to ALTER TABLE for performing operations of this type on one or more partitions directly, as described in the following list:

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

  Example:

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

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

  Example:

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

  Using OPTIMIZE PARTITION on a given partition is equivalent to running CHECK PARTITION, ANALYZE PARTITION, and REPAIR PARTITION on that partition.

- **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 TA-BLE ... REPAIR PARTITION to repair the partition.

  > **Note**
  >
  > The statements ALTER TABLE ... ANALYZE PARTITION, ALTER TABLE ... CHECK PARTITION, ALTER TABLE ... OPTIMIZE PARTITION, and ALTER TABLE ... REPAIR PARTITION were origin-ally introduced in MySQL 5.1.5, but did not work properly and were disabled in MySQL 5.1.24. They were re-introduced in MySQL 5.1.27. (Bug#20129) The use of these partitioning-specific ALTER TABLE statements with tables which are not partitioned is not supported; beginning with MySQL 5.1.31, it is expressly disallowed. (Bug#39434)
  >
  > ALTER TABLE ... REBUILD PARTITION was also introduced in MySQL 5.1.5.

3.4. 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, for more information about this statement.)

You can also obtain information about partitions from INFORMATION_SCHEMA, which contains a PARTITIONS table. See The INFORMATION_SCHEMA PARTITIONS Table.

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`.

As of MySQL 5.1.28, the `rows` column of `EXPLAIN PARTITIONS` output always displays the total number of records in the table. Previously, this was the number of matching rows. (Bug#35745)

See also `EXPLAIN` Syntax.

Chapter 4. Partition Pruning

This section discusses an optimization known as *partition pruning*, 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, region_code, dob
    FROM t1
    WHERE region_code > 125 AND region_code < 130;
```

It is easy to see that none of the rows which ought to be returned will be in either of the partitions `p0` or `p3`; that is, we need to search only in partitions `p1` and `p2` to find matching rows. By doing so, it is possible to expend much 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 optimizer 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 optimizer 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 many cases, the equals sign can be replaced with another arithmetic comparison, including `<`, `>`, `<=`, `>=`, and `<>`. Some queries using `BETWEEN` in the `WHERE` clause can also take advantage of partition pruning. See the examples later in this section.

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

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, or when the partitioning expression represents an increasing or decreasing relationship. Pruning can also be applied for tables partitioned on a `DATE` or `DATETIME` column when the partitioning expression uses the `YEAR()` or `TO_DAYS()` function.

> **Note**
>
> We plan to add pruning support in a future MySQL release for additional functions that act on a `DATE` or `DATETIME` value, return an integer, and are increasing or decreasing.

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 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).

> **Important**
>
> Invalid `DATE` and `DATETIME` values referenced in the `WHERE` clause of a query on a partitioned table are treated as `NULL`. This means that a query such as `SELECT * FROM partitioned_table WHERE date_column < '2008-12-00'` does not return any values (see Bug#40972).

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

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

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

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

For tables that are partitioned by `HASH` or `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 5. Restrictions and Limitations on Partitioning

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

- **Prohibited constructs.** Beginning with MySQL 5.1.12, the following constructs are not permitted in partitioning expressions:

  - Stored functions, stored procedures, UDFs, or plugins.

  - Declared variables or user variables.

  For a list of SQL functions which are permitted in partitioning expressions, see Section 5.3, "Partitioning Limitations Relating to Functions".

- **Arithmetic and logical operators.** Use of the arithmetic operators `+`, `−`, and `*` is permitted in partitioning expressions. However, the result must be an integer value or `NULL` (except in the case of `[LINEAR] KEY` partitioning, as discussed elswhere in this chapter — see Chapter 2, *Partition Types*, for more information).

  Beginning with MySQL 5.1.23, the `DIV` operator is also supported, and the `/` operator is disallowed. (Bug#30188, Bug#33182)

  Beginning with MySQL 5.1.12, the bit operators `|`, `&`, `^`, `<<`, `>>`, and `~` are not permitted in partitioning expressions.

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

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

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

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

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

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

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

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

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

     ```
     mysql> SELECT @@SQL_MODE;
     +------------+
     | @@SQL_MODE |
     +------------+
     |            |
     +------------+
     1 row in set (0.00 sec)
     mysql> CREATE TABLE tu (c1 BIGINT UNSIGNED)
         ->     PARTITION BY RANGE(c1 - 10) (
         ->       PARTITION p0 VALUES LESS THAN (-5),
         ->       PARTITION p1 VALUES LESS THAN (0),
         ->       PARTITION p2 VALUES LESS THAN (5),
         ->       PARTITION p3 VALUES LESS THAN (10),
         ->       PARTITION p4 VALUES LESS THAN (MAXVALUE)
         -> );
     ```

```
ERROR 1563 (HY000): PARTITION CONSTANT IS OUT OF PARTITION FUNCTION DOMAIN
mysql> SET SQL_MODE='NO_UNSIGNED_SUBTRACTION';
Query OK, 0 rows affected (0.00 sec)
mysql> SELECT @@SQL_MODE;
+------------------------+
| @@SQL_MODE             |
+------------------------+
| NO_UNSIGNED_SUBTRACTION |
+------------------------+
1 row in set (0.00 sec)
mysql> CREATE TABLE tu (c1 BIGINT UNSIGNED)
    ->   PARTITION BY RANGE(c1 - 10) (
    ->     PARTITION p0 VALUES LESS THAN (-5),
    ->     PARTITION p1 VALUES LESS THAN (0),
    ->     PARTITION p2 VALUES LESS THAN (5),
    ->     PARTITION p3 VALUES LESS THAN (10),
    ->     PARTITION p4 VALUES LESS THAN (MAXVALUE)
    -> );
Query OK, 0 rows affected (0.05 sec)
```

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

```
mysql> SET SQL_MODE='';
Query OK, 0 rows affected (0.00 sec)
mysql> SELECT * FROM tu;
ERROR 1563 (HY000): PARTITION CONSTANT IS OUT OF PARTITION FUNCTION DOMAIN
mysql> INSERT INTO tu VALUES (20);
ERROR 1563 (HY000): PARTITION CONSTANT IS OUT OF PARTITION FUNCTION DOMAIN
```

- **Performance considerations.**

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

  - **Table locks.** The process executing a partitioning operation on a table takes a write lock on the table. Reads from such tables are relatively unaffected; pending `INSERT` and `UPDATE` operations are performed as soon as the partitioning operation has completed.

  - **Storage engine.** Partitioning operations, queries, and update operations generally tend to be faster with `MyISAM` tables than with `InnoDB` or `NDB` tables.

  - **Use of indexes and partition pruning.** As with non-partitioned tables, proper use of indexes can speed up queries on partitioned tables significantly. In addition, designing partitioned tables and queries on these tables to take advantage of *partition pruning* can improve performance dramatically. See Chapter 4, *Partition Pruning*, for more information.

  - **Performance with `LOAD DATA`.** Prior to MySQL 5.1.23, `LOAD DATA` performed very poorly when importing into partitioned tables. The statement now uses buffering to improve performance; however, the buffer uses 130 KB memory per partition to achieve this. (Bug#26527)

- **Maximum number of partitions.** The maximum number of partitions possible for a given table is 1024. This includes subpartitions.

  If, when creating tables with a very large number of partitions (but which is less than the maximum stated previously), you encounter an error message such as GOT ERROR 24 FROM STORAGE ENGINE, this means that you may need to increase the value of the `open_files_limit` system variable. See '*FILE*' NOT FOUND and Similar Errors.

- **Foreign keys not supported.** Partitioned tables do not support foreign keys. This means that:

  1. Definitions of tables employing user-defined partitioning may not contain foreign key references to other tables.

  2. No table definition may contain a foreign key reference to a partitioned table.
  The scope of these restrictions includes tables that use the `InnoDB` storage engine.

- **`ALTER TABLE ... ORDER BY`.** An `ALTER TABLE ... ORDER BY column` statement run against a partitioned table causes ordering of rows only within each partition.

- **FULLTEXT indexes.** Partitioned tables do not support `FULLTEXT` indexes. This includes partitioned tables employing the `MyISAM` storage engine.

- **Spatial columns.** Columns with spatial data types such as POINT or GEOMETRY cannot be used in partitioned tables.

- **Temporary tables.** As of MySQL 5.1.8, temporary tables cannot be partitioned. (Bug#17497)

- **Log tables.** Beginning with MySQL 5.1.20, it is no longer possible to partition the log tables; beginning with that version, an ALTER TABLE ... PARTITION BY ... statement on such a table fails with an error. (Bug#27816)

- **Data type of partitioning key.** 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 2.6, "How MySQL Partitioning Handles NULL".)

  The lone exception to this restriction occurs when partitioning by [LINEAR] KEY, where it is possible to use columns of other types as partitioning keys, because MySQL's internal key-hashing functions produce the correct data type 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.

- **Subqueries.** A partitioning key may not be a subquery, even if that subquery resolves to an integer value or NULL.

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

- **Key caches not supported.** Key caches are not supported for partitioned tables. The CACHE INDEX and LOAD INDEX INTO CACHE statements, when you attempt to use them on tables having user-defined partitioning, fail with the errors THE STORAGE ENGINE FOR THE TABLE DOESN'T SUPPORT ASSIGN_TO_KEYCACHE and THE STORAGE ENGINE FOR THE TABLE DOESN'T SUPPORT PRELOAD_KEYS, respectively.

- **DELAYED option not supported.** Use of INSERT DELAYED to insert rows into a partitioned table is not supported. Beginning with MySQL 5.1.23, attempting to do so fails with an error. (Bug#31210)

- **DATA DIRECTORY and INDEX DIRECTORY options.** DATA DIRECTORY and INDEX DIRECTORY are subject to the following restrictions when used with partitioned tables:

  - Beginning with MySQL 5.1.23, table-level DATA DIRECTORY and INDEX DIRECTORY options are ignored. (Bug#32091)

  - On Windows, the DATA DIRECTORY and INDEX DIRECTORY options are not supported for individual partitions or subpartitions (Bug#30459).

- **Repairing and rebuilding partitioned tables.** The statements CHECK TABLE, OPTIMIZE TABLE, ANALYZE TABLE, and REPAIR TABLE are supported for partitioned tables beginning with MySQL 5.1.27. (See Bug#20129.) mysqlcheck and myisamchk are not supported with partitioned tables.

  In addition, you can use ALTER TABLE ... REBUILD PARTITION to rebuild one or more partitions of a partitioned table; ALTER TABLE ... REORGANIZE PARTITION also causes partitions to be rebuilt. Both of these statements were added in MySQL 5.1.5. See ALTER TABLE Syntax, for more information about these two statements.

5.1. Partitioning Keys, Primary Keys, and Unique Keys

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

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

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

```
    UNIQUE KEY (col1),
    UNIQUE KEY (col3)
)
PARTITION BY HASH(col1 + col3)
PARTITIONS 4;
```

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

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

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

This example shows the error produced in such cases:

```
mysql> CREATE TABLE t3 (
    ->     col1 INT NOT NULL,
    ->     col2 DATE NOT NULL,
    ->     col3 INT NOT NULL,
    ->     col4 INT NOT NULL,
    ->     UNIQUE KEY (col1, col2),
    ->     UNIQUE KEY (col3)
    -> )
    -> PARTITION BY HASH(col1 + col3)
    -> PARTITIONS 4;
ERROR 1491 (HY000): A PRIMARY KEY MUST INCLUDE ALL COLUMNS IN THE TABLE'S PARTITIONING FUNCTION
```

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

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

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

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:

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

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

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

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

```
#  fails with error 1503
mysql> ALTER TABLE t_no_pk ADD PRIMARY KEY(c2);
ERROR 1503 (HY000): A PRIMARY KEY MUST INCLUDE ALL COLUMNS IN THE TABLE'S PARTITIONING FUNCTION
```
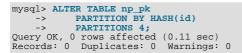
Since `t_no_pk` has only `c1` in its partitioning expression, attempting to adding a unique key on `c2` alone fails. However, you can add a unique key that uses both `c1` and `c2`.

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

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

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

```
mysql> ALTER TABLE np_pk
    ->     PARTITION BY HASH( TO_DAYS(added) )
    ->     PARTITIONS 4;
ERROR 1503 (HY000): A PRIMARY KEY MUST INCLUDE ALL COLUMNS IN THE TABLE'S PARTITIONING FUNCTION
```

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

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

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

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

5.2. Partitioning Limitations Relating to Storage Engines

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

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

FEDERATED storage engine. Partitioning of `FEDERATED` tables is not supported. Beginning with MySQL 5.1.15, it is not possible to create partitioned `FEDERATED` tables at all. We are working to remove this limitation in a future MySQL release.

CSV storage engine. 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.

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

NDBCLUSTER storage engine (MySQL Cluster). Partitioning by `KEY` (or `LINEAR KEY`) is the only type of partitioning supported for the `NDBCLUSTER` storage engine. Beginning with MySQL 5.1.12, it is not possible to create a MySQL Cluster table using any partitioning type other than `[LINEAR] KEY`, and attempting to do so fails with an error.

In addition, the maximum number of partitions that can be defined for an `NDBCLUSTER` table is 8 times the number of node groups in the cluster. (See MySQL Cluster Nodes, Node Groups, Replicas, and Partitions, for more information about node groups in MySQL Cluster.)

Beginning with MySQL Cluster NDB 6.2.18, MySQL Cluster NDB 6.3.25, and MySQL Cluster NDB 7.0.6, `CREATE TABLE` and `ALTER TABLE` statements that would cause a user-partitioned `NDBCLUSTER` table not to meet either or both of the following two requirements are disallowed, and fail with an error (Bug#40709):

1. The table must have an explicit primary key.

2. All columns listed in the table's partitioning expression must be part of the primary key.

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

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

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

•   Do *not* specify any engine for *any* partition or subpartition

•   Specify the engine for *all* partitions or subpartitions

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

5.3. Partitioning Limitations Relating to Functions

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

Beginning with MySQL 5.1.12, only the following MySQL functions are supported in partitioning expressions:

- ABS()

- CEILING() (see *CEILING( ) and FLOOR( )*, immediately following this list)

- DAY()

- DAYOFMONTH()

- DAYOFWEEK()

- DAYOFYEAR()

- DATEDIFF()

- EXTRACT()

- FLOOR() (see *CEILING( ) and FLOOR( )*, immediately following this list)

- HOUR()

- MICROSECOND()

- MINUTE()

- MOD()

- MONTH()

- QUARTER()

- SECOND()

- TIME_TO_SEC()

- TO_DAYS()

- WEEKDAY()

- YEAR()

- YEARWEEK()

> **Note**
>
> **CEILING()** and **FLOOR().** Each of these functions returns an integer only if it is passed an integer argument. This means, for example, that the following CREATE TABLE statement fails with an error, as shown here:
>
> ```
> mysql> CREATE TABLE t (c FLOAT) PARTITION BY LIST( FLOOR(c) )(
>     ->     PARTITION p0 VALUES IN (1,3,5),
>     ->     PARTITION p1 VALUES IN (2,4,6)
>     -> );
> ERROR 1490 (HY000): THE PARTITION FUNCTION RETURNS THE WRONG TYPE
> ```
>
> See Mathematical Functions, for more information about the return types of these functions.