The `innodb_stats_method` system variable has a global value; the `myisam_stats_method` system variable has both global and session values. Setting the global value affects statistics collection for tables from the corresponding storage engine. Setting the session value affects statistics collection only for the current client connection. This means that you can force a table's statistics to be regenerated with a given method without affecting other clients by setting the session value of `myisam_stats_method`.

To regenerate `MyISAM` table statistics, you can use any of the following methods:

- Execute `myisamchk --stats_method=method_name --analyze`
- Change the table to cause its statistics to go out of date (for example, insert a row and then delete it), and then set `myisam_stats_method` and issue an `ANALYZE TABLE` statement

Some caveats regarding the use of `innodb_stats_method` and `myisam_stats_method`:

- You can force table statistics to be collected explicitly, as just described. However, MySQL may also collect statistics automatically. For example, if during the course of executing statements for a table, some of those statements modify the table, MySQL may collect statistics. (This may occur for bulk inserts or deletes, or some `ALTER TABLE` statements, for example.) If this happens, the statistics are collected using whatever value `innodb_stats_method` or `myisam_stats_method` has at the time. Thus, if you collect statistics using one method, but the system variable is set to the other method when a table's statistics are collected automatically later, the other method is used.
- There is no way to tell which method was used to generate statistics for a given table.
- These variables apply only to `InnoDB` and `MyISAM` tables. Other storage engines have only one method for collecting table statistics. Usually it is closer to the `nulls_equal` method.

8.3.9 Comparison of B-Tree and Hash Indexes

Understanding the B-tree and hash data structures can help predict how different queries perform on different storage engines that use these data structures in their indexes, particularly for the `MEMORY` storage engine that lets you choose B-tree or hash indexes.

B-Tree Index Characteristics

A B-tree index can be used for column comparisons in expressions that use the `=`, `>`, `>=`, `<`, `<=`, or `BETWEEN` operators. The index also can be used for `LIKE` comparisons if the argument to `LIKE` is a constant string that does not start with a wildcard character. For example, the following `SELECT` statements use indexes:

```
SELECT * FROM tbl_name WHERE key_col LIKE 'Patrick%';
SELECT * FROM tbl_name WHERE key_col LIKE 'Pat%_ck%';
```

In the first statement, only rows with `'Patrick' <= key_col < 'Patricl'` are considered. In the second statement, only rows with `'Pat' <= key_col < 'Pau'` are considered.

The following `SELECT` statements do not use indexes:

```
SELECT * FROM tbl_name WHERE key_col LIKE '%Patrick%';
SELECT * FROM tbl_name WHERE key_col LIKE other_col;
```

In the first statement, the `LIKE` value begins with a wildcard character. In the second statement, the `LIKE` value is not a constant.

If you use `... LIKE '%string%'` and *`string`* is longer than three characters, MySQL uses the *Turbo Boyer-Moore algorithm* to initialize the pattern for the string and then uses this pattern to perform the search more quickly.

A search using *`col_name IS NULL`* employs indexes if *`col_name`* is indexed.

Any index that does not span all `AND` levels in the `WHERE` clause is not used to optimize the query. In other words, to be able to use an index, a prefix of the index must be used in every `AND` group.

The following `WHERE` clauses use indexes:

```
... WHERE index_part1=1 AND index_part2=2 AND other_column=3

    /* index = 1 OR index = 2 */
... WHERE index=1 OR A=10 AND index=2

    /* optimized like "index_part1='hello'" */
... WHERE index_part1='hello' AND index_part3=5

    /* Can use index on index1 but not on index2 or index3 */
... WHERE index1=1 AND index2=2 OR index1=3 AND index3=3;
```

These `WHERE` clauses do *not* use indexes:

```
    /* index_part1 is not used */
... WHERE index_part2=1 AND index_part3=2

    /*  Index is not used in both parts of the WHERE clause  */
... WHERE index=1 OR A=10

    /* No index spans all rows  */
... WHERE index_part1=1 OR index_part2=10
```

Sometimes MySQL does not use an index, even if one is available. One circumstance under which this occurs is when the optimizer estimates that using the index would require MySQL to access a very large percentage of the rows in the table. (In this case, a table scan is likely to be much faster because it requires fewer seeks.) However, if such a query uses `LIMIT` to retrieve only some of the rows, MySQL uses an index anyway, because it can much more quickly find the few rows to return in the result.

Hash Index Characteristics

Hash indexes have somewhat different characteristics from those just discussed:

- They are used only for equality comparisons that use the `=` or `<=>` operators (but are *very* fast). They are not used for comparison operators such as `<` that find a range of values. Systems that rely on this type of single-value lookup are known as “key-value stores”; to use MySQL for such applications, use hash indexes wherever possible.
- The optimizer cannot use a hash index to speed up `ORDER BY` operations. (This type of index cannot be used to search for the next entry in order.)
- MySQL cannot determine approximately how many rows there are between two values (this is used by the range optimizer to decide which index to use). This may affect some queries if you change a `MyISAM` or `InnoDB` table to a hash-indexed `MEMORY` table.
- Only whole keys can be used to search for a row. (With a B-tree index, any leftmost prefix of the key can be used to find rows.)

8.3.10 Use of Index Extensions

`InnoDB` automatically extends each secondary index by appending the primary key columns to it. Consider this table definition:

```
CREATE TABLE t1 (
  i1 INT NOT NULL DEFAULT 0,
  i2 INT NOT NULL DEFAULT 0,
```

```
  d DATE DEFAULT NULL,
  PRIMARY KEY (i1, i2),
  INDEX k_d (d)
) ENGINE = InnoDB;
```

This table defines the primary key on columns `(i1, i2)`. It also defines a secondary index `k_d` on column `(d)`, but internally `InnoDB` extends this index and treats it as columns `(d, i1, i2)`.

The optimizer takes into account the primary key columns of the extended secondary index when determining how and whether to use that index. This can result in more efficient query execution plans and better performance.

The optimizer can use extended secondary indexes for `ref`, `range`, and `index_merge` index access, for Loose Index Scan access, for join and sorting optimization, and for `MIN()`/`MAX()` optimization.

The following example shows how execution plans are affected by whether the optimizer uses extended secondary indexes. Suppose that `t1` is populated with these rows:

```
INSERT INTO t1 VALUES
(1, 1, '1998-01-01'), (1, 2, '1999-01-01'),
(1, 3, '2000-01-01'), (1, 4, '2001-01-01'),
(1, 5, '2002-01-01'), (2, 1, '1998-01-01'),
(2, 2, '1999-01-01'), (2, 3, '2000-01-01'),
(2, 4, '2001-01-01'), (2, 5, '2002-01-01'),
(3, 1, '1998-01-01'), (3, 2, '1999-01-01'),
(3, 3, '2000-01-01'), (3, 4, '2001-01-01'),
(3, 5, '2002-01-01'), (4, 1, '1998-01-01'),
(4, 2, '1999-01-01'), (4, 3, '2000-01-01'),
(4, 4, '2001-01-01'), (4, 5, '2002-01-01'),
(5, 1, '1998-01-01'), (5, 2, '1999-01-01'),
(5, 3, '2000-01-01'), (5, 4, '2001-01-01'),
(5, 5, '2002-01-01');
```

Now consider this query:

```
EXPLAIN SELECT COUNT(*) FROM t1 WHERE i1 = 3 AND d = '2000-01-01'
```

The execution plan depends on whether the extended index is used.

When the optimizer does not consider index extensions, it treats the index `k_d` as only `(d)`. `EXPLAIN` for the query produces this result:

```
mysql> EXPLAIN SELECT COUNT(*) FROM t1 WHERE i1 = 3 AND d = '2000-01-01'\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: t1
         type: ref
possible_keys: PRIMARY,k_d
          key: k_d
      key_len: 4
          ref: const
         rows: 5
        Extra: Using where; Using index
```

When the optimizer takes index extensions into account, it treats `k_d` as `(d, i1, i2)`. In this case, it can use the leftmost index prefix `(d, i1)` to produce a better execution plan:

```
mysql> EXPLAIN SELECT COUNT(*) FROM t1 WHERE i1 = 3 AND d = '2000-01-01'\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: t1
         type: ref
possible_keys: PRIMARY,k_d
          key: k_d
      key_len: 8
```

```
          ref: const,const
         rows: 1
        Extra: Using index
```

In both cases, `key` indicates that the optimizer uses secondary index `k_d` but the `EXPLAIN` output shows these improvements from using the extended index:

- `key_len` goes from 4 bytes to 8 bytes, indicating that key lookups use columns `d` and `i1`, not just `d`.
- The `ref` value changes from `const` to `const,const` because the key lookup uses two key parts, not one.
- The `rows` count decreases from 5 to 1, indicating that `InnoDB` should need to examine fewer rows to produce the result.
- The `Extra` value changes from `Using where; Using index` to `Using index`. This means that rows can be read using only the index, without consulting columns in the data row.

Differences in optimizer behavior for use of extended indexes can also be seen with `SHOW STATUS`:

```
FLUSH TABLE t1;
FLUSH STATUS;
SELECT COUNT(*) FROM t1 WHERE i1 = 3 AND d = '2000-01-01';
SHOW STATUS LIKE 'handler_read%'
```

The preceding statements include `FLUSH TABLES` and `FLUSH STATUS` to flush the table cache and clear the status counters.

Without index extensions, `SHOW STATUS` produces this result:

```
+-----------------------+-------+
| Variable_name         | Value |
+-----------------------+-------+
| Handler_read_first    | 0     |
| Handler_read_key      | 1     |
| Handler_read_last     | 0     |
| Handler_read_next     | 5     |
| Handler_read_prev     | 0     |
| Handler_read_rnd      | 0     |
| Handler_read_rnd_next | 0     |
+-----------------------+-------+
```

With index extensions, `SHOW STATUS` produces this result. The `Handler_read_next` value decreases from 5 to 1, indicating more efficient use of the index:

```
+-----------------------+-------+
| Variable_name         | Value |
+-----------------------+-------+
| Handler_read_first    | 0     |
| Handler_read_key      | 1     |
| Handler_read_last     | 0     |
| Handler_read_next     | 1     |
| Handler_read_prev     | 0     |
| Handler_read_rnd      | 0     |
| Handler_read_rnd_next | 0     |
+-----------------------+-------+
```

The `use_index_extensions` flag of the `optimizer_switch` system variable permits control over whether the optimizer takes the primary key columns into account when determining how to use an `InnoDB` table's secondary indexes. By default, `use_index_extensions` is enabled. To check whether disabling use of index extensions can improve performance, use this statement:

```
SET optimizer_switch = 'use_index_extensions=off';
```

Use of index extensions by the optimizer is subject to the usual limits on the number of key parts in an index (16) and the maximum key length (3072 bytes).

8.3.11 Optimizer Use of Generated Column Indexes

MySQL supports indexes on generated columns. For example:

```
CREATE TABLE t1 (f1 INT, gc INT AS (f1 + 1) STORED, INDEX (gc));
```

The generated column, `gc`, is defined as the expression `f1 + 1`. The column is also indexed and the optimizer can take that index into account during execution plan construction. In the following query, the `WHERE` clause refers to `gc` and the optimizer considers whether the index on that column yields a more efficient plan:

```
SELECT * FROM t1 WHERE gc > 9;
```

The optimizer can use indexes on generated columns to generate execution plans, even in the absence of direct references in queries to those columns by name. This occurs if the `WHERE`, `ORDER BY`, or `GROUP BY` clause refers to an expression that matches the definition of some indexed generated column. The following query does not refer directly to `gc` but does use an expression that matches the definition of `gc`:

```
SELECT * FROM t1 WHERE f1 + 1 > 9;
```

The optimizer recognizes that the expression `f1 + 1` matches the definition of `gc` and that `gc` is indexed, so it considers that index during execution plan construction. You can see this using `EXPLAIN`:

```
mysql> EXPLAIN SELECT * FROM t1 WHERE f1 + 1 > 9\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: t1
   partitions: NULL
         type: range
possible_keys: gc
          key: gc
      key_len: 5
          ref: NULL
         rows: 1
     filtered: 100.00
        Extra: Using index condition
```

In effect, the optimizer has replaced the expression `f1 + 1` with the name of the generated column that matches the expression. That is also apparent in the rewritten query available in the extended `EXPLAIN` information displayed by `SHOW WARNINGS`:

```
mysql> SHOW WARNINGS\G
*************************** 1. row ***************************
  Level: Note
   Code: 1003
Message: /* select#1 */ select `test`.`t1`.`f1` AS `f1`,`test`.`t1`.`gc`
         AS `gc` from `test`.`t1` where (`test`.`t1`.`gc` > 9)
```

The following restrictions and conditions apply to the optimizer's use of generated column indexes:

- For a query expression to match a generated column definition, the expression must be identical and it must have the same result type. For example, if the generated column expression is `f1 + 1`, the optimizer does not recognize a match if the query uses `1 + f1`, or if `f1 + 1` (an integer expression) is compared with a string.

- The optimization applies to these operators: `=`, `<`, `<=`, `>`, `>=`, `BETWEEN`, and `IN()`.

  For operators other than `BETWEEN` and `IN()`, either operand can be replaced by a matching generated column. For `BETWEEN` and `IN()`, only the first argument can be replaced by a matching generated column, and the other arguments must have the same result type. `BETWEEN` and `IN()` are not yet supported for comparisons involving JSON values.

- The generated column must be defined as an expression that contains at least a function call or one of the operators mentioned in the preceding item. The expression cannot consist of a simple reference to another column. For example, `gc INT AS (f1) STORED` consists only of a column reference, so indexes on `gc` are not considered.

- For comparisons of strings to indexed generated columns that compute a value from a JSON function that returns a quoted string, `JSON_UNQUOTE()` is needed in the column definition to remove the extra quotes from the function value. (For direct comparison of a string to the function result, the JSON comparator handles quote removal, but this does not occur for index lookups.) For example, instead of writing a column definition like this:

  ```
  doc_name TEXT AS (JSON_EXTRACT(jdoc, '$.name')) STORED
  ```

  Write it like this:

  ```
  doc_name TEXT AS (JSON_UNQUOTE(JSON_EXTRACT(jdoc, '$.name'))) STORED
  ```

  With the latter definition, the optimizer can detect a match for both of these comparisons:

  ```
  ... WHERE JSON_EXTRACT(jdoc, '$.name') = 'some_string' ...
  ... WHERE JSON_UNQUOTE(JSON_EXTRACT(jdoc, '$.name')) = 'some_string' ...
  ```

  Without `JSON_UNQUOTE()` in the column definition, the optimizer detects a match only for the first of those comparisons.

- If the optimizer picks the wrong index, an index hint can be used to disable it and force the optimizer to make a different choice.

8.3.12 Invisible Indexes

MySQL supports invisible indexes; that is, indexes that are not used by the optimizer. The feature applies to indexes other than primary keys (either explicit or implicit).

Indexes are visible by default. To control visibility explicitly for a new index, use a `VISIBLE` or `INVISIBLE` keyword as part of the index definition for `CREATE TABLE`, `CREATE INDEX`, or `ALTER TABLE`:

```
CREATE TABLE t1 (
  i INT,
  j INT,
  k INT,
  INDEX i_idx (i) INVISIBLE
) ENGINE = InnoDB;
CREATE INDEX j_idx ON t1 (j) INVISIBLE;
ALTER TABLE t1 ADD INDEX k_idx (k) INVISIBLE;
```

To alter the visibility of an existing index, use a `VISIBLE` or `INVISIBLE` keyword with the `ALTER TABLE ... ALTER INDEX` operation:

```
ALTER TABLE t1 ALTER INDEX i_idx INVISIBLE;
ALTER TABLE t1 ALTER INDEX i_idx VISIBLE;
```

Information about whether an index is visible or invisible is available from the Information Schema `STATISTICS` table or `SHOW INDEX` output. For example:

```
mysql> SELECT INDEX_NAME, IS_VISIBLE
       FROM INFORMATION_SCHEMA.STATISTICS
       WHERE TABLE_SCHEMA = 'db1' AND TABLE_NAME = 't1';
+------------+------------+
| INDEX_NAME | IS_VISIBLE |
+------------+------------+
| i_idx      | YES        |
| j_idx      | NO         |
| k_idx      | NO         |
+------------+------------+
```

Invisible indexes make it possible to test the effect of removing an index on query performance, without making a destructive change that must be undone should the index turn out to be required. Dropping and re-adding an index can be expensive for a large table, whereas making it invisible and visible are fast, in-place operations.

If an index made invisible actually is needed or used by the optimizer, there are several ways to notice the effect of its absence on queries for the table:

- Errors occur for queries that include index hints that refer to the invisible index.
- Performance Schema data shows an increase in workload for affected queries.
- Queries have different `EXPLAIN` execution plans.
- Queries appear in the slow query log that did not appear there previously.

The `use_invisible_indexes` flag of the `optimizer_switch` system variable controls whether the optimizer uses invisible indexes for query execution plan construction. If the flag is `off` (the default), the optimizer ignores invisible indexes (the same behavior as prior to the introduction of this flag). If the flag is `on`, invisible indexes remain invisible but the optimizer takes them into account for execution plan construction.

Using the `SET_VAR` optimizer hint to update the value of `optimizer_switch` temporarily, you can enable invisible indexes for the duration of a single query only, like this:

```
mysql> EXPLAIN SELECT /*+ SET_VAR(optimizer_switch = 'use_invisible_indexes=on') */
     >     i, j FROM t1 WHERE j >= 50\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: t1
   partitions: NULL
         type: range
possible_keys: j_idx
          key: j_idx
      key_len: 5
          ref: NULL
         rows: 2
     filtered: 100.00
        Extra: Using index condition

mysql> EXPLAIN SELECT i, j FROM t1 WHERE j >= 50\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: t1
   partitions: NULL
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 5
     filtered: 33.33
        Extra: Using where
```

Index visibility does not affect index maintenance. For example, an index continues to be updated per changes to table rows, and a unique index prevents insertion of duplicates into a column, regardless of whether the index is visible or invisible.

A table with no explicit primary key may still have an effective implicit primary key if it has any `UNIQUE` indexes on `NOT NULL` columns. In this case, the first such index places the same constraint on table rows as an explicit primary key and that index cannot be made invisible. Consider the following table definition:

```
CREATE TABLE t2 (
```

```
  i INT NOT NULL,
  j INT NOT NULL,
  UNIQUE j_idx (j)
) ENGINE = InnoDB;
```

The definition includes no explicit primary key, but the index on `NOT NULL` column `j` places the same constraint on rows as a primary key and cannot be made invisible:

```
mysql> ALTER TABLE t2 ALTER INDEX j_idx INVISIBLE;
ERROR 3522 (HY000): A primary key index cannot be invisible.
```

Now suppose that an explicit primary key is added to the table:

```
ALTER TABLE t2 ADD PRIMARY KEY (i);
```

The explicit primary key cannot be made invisible. In addition, the unique index on `j` no longer acts as an implicit primary key and as a result can be made invisible:

```
mysql> ALTER TABLE t2 ALTER INDEX j_idx INVISIBLE;
Query OK, 0 rows affected (0.03 sec)
```

8.3.13 Descending Indexes

MySQL supports descending indexes: `DESC` in an index definition is no longer ignored but causes storage of key values in descending order. Previously, indexes could be scanned in reverse order but at a performance penalty. A descending index can be scanned in forward order, which is more efficient. Descending indexes also make it possible for the optimizer to use multiple-column indexes when the most efficient scan order mixes ascending order for some columns and descending order for others.

Consider the following table definition, which contains two columns and four two-column index definitions for the various combinations of ascending and descending indexes on the columns:

```
CREATE TABLE t (
  c1 INT, c2 INT,
  INDEX idx1 (c1 ASC, c2 ASC),
  INDEX idx2 (c1 ASC, c2 DESC),
  INDEX idx3 (c1 DESC, c2 ASC),
  INDEX idx4 (c1 DESC, c2 DESC)
);
```

The table definition results in four distinct indexes. The optimizer can perform a forward index scan for each of the `ORDER BY` clauses and need not use a `filesort` operation:

```
ORDER BY c1 ASC, c2 ASC    -- optimizer can use idx1
ORDER BY c1 DESC, c2 DESC  -- optimizer can use idx4
ORDER BY c1 ASC, c2 DESC   -- optimizer can use idx2
ORDER BY c1 DESC, c2 ASC   -- optimizer can use idx3
```

Use of descending indexes is subject to these conditions:

- Descending indexes are supported only for the `InnoDB` storage engine, with these limitations:
  - Change buffering is not supported for a secondary index if the index contains a descending index key column or if the primary key includes a descending index column.
  - The `InnoDB` SQL parser does not use descending indexes. For `InnoDB` full-text search, this means that the index required on the `FTS_DOC_ID` column of the indexed table cannot be defined as a descending index. For more information, see Section 15.6.2.4, “InnoDB Full-Text Indexes”.
- Descending indexes are supported for all data types for which ascending indexes are available.
- Descending indexes are supported for ordinary (nongenerated) and generated columns (both `VIRTUAL` and `STORED`).
- `DISTINCT` can use any index containing matching columns, including descending key parts.

- Indexes that have descending key parts are not used for `MIN()`/`MAX()` optimization of queries that invoke aggregate functions but do not have a `GROUP BY` clause.
- Descending indexes are supported for `BTREE` but not `HASH` indexes. Descending indexes are not supported for `FULLTEXT` or `SPATIAL` indexes.

  Explicitly specified `ASC` and `DESC` designators for `HASH`, `FULLTEXT`, and `SPATIAL` indexes results in an error.

You can see in the **`Extra`** column of the output of `EXPLAIN` that the optimizer is able to use a descending index, as shown here:

```
mysql> CREATE TABLE t1 (
    -> a INT,
    -> b INT,
    -> INDEX a_desc_b_asc (a DESC, b ASC)
    -> );

mysql> EXPLAIN SELECT * FROM t1 ORDER BY a ASC\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: t1
   partitions: NULL
         type: index
possible_keys: NULL
          key: a_desc_b_asc
      key_len: 10
          ref: NULL
         rows: 1
     filtered: 100.00
        Extra: Backward index scan; Using index
```

In `EXPLAIN FORMAT=TREE` output, use of a descending index is indicated by the addition of `(reverse)` following the name of the index, like this:

```
mysql> EXPLAIN FORMAT=TREE SELECT * FROM t1 ORDER BY a ASC\G
*************************** 1. row ***************************
EXPLAIN: -> Index scan on t1 using a_desc_b_asc (reverse)  (cost=0.35 rows=1)
```

See also EXPLAIN Extra Information.

8.3.14 Indexed Lookups from TIMESTAMP Columns

Temporal values are stored in `TIMESTAMP` columns as UTC values, and values inserted into and retrieved from `TIMESTAMP` columns are converted between the session time zone and UTC. (This is the same type of conversion performed by the `CONVERT_TZ()` function. If the session time zone is UTC, there is effectively no time zone conversion.)

Due to conventions for local time zone changes such as Daylight Saving Time (DST), conversions between UTC and non-UTC time zones are not one-to-one in both directions. UTC values that are distinct may not be distinct in another time zone. The following example shows distinct UTC values that become identical in a non-UTC time zone:

```
mysql> CREATE TABLE tstable (ts TIMESTAMP);
mysql> SET time_zone = 'UTC'; -- insert UTC values
mysql> INSERT INTO tstable VALUES
       ('2018-10-28 00:30:00'),
       ('2018-10-28 01:30:00');
mysql> SELECT ts FROM tstable;
+---------------------+
| ts                  |
+---------------------+
| 2018-10-28 00:30:00 |
| 2018-10-28 01:30:00 |
+---------------------+
mysql> SET time_zone = 'MET'; -- retrieve non-UTC values
```

```
mysql> SELECT ts FROM tstable;
+---------------------+
| ts                  |
+---------------------+
| 2018-10-28 02:30:00 |
| 2018-10-28 02:30:00 |
+---------------------+
```

Note

To use named time zones such as `'MET'` or `'Europe/Amsterdam'`, the time zone tables must be properly set up. For instructions, see Section 5.1.15, “MySQL Server Time Zone Support”.

You can see that the two distinct UTC values are the same when converted to the `'MET'` time zone. This phenomenon can lead to different results for a given `TIMESTAMP` column query, depending on whether the optimizer uses an index to execute the query.

Suppose that a query selects values from the table shown earlier using a `WHERE` clause to search the `ts` column for a single specific value such as a user-provided timestamp literal:

```
SELECT ts FROM tstable
WHERE ts = 'literal';
```

Suppose further that the query executes under these conditions:

- The session time zone is not UTC and has a DST shift. For example:

  ```
  SET time_zone = 'MET';
  ```

- Unique UTC values stored in the `TIMESTAMP` column are not unique in the session time zone due to DST shifts. (The example shown earlier illustrates how this can occur.)

- The query specifies a search value that is within the hour of entry into DST in the session time zone.

Under those conditions, the comparison in the `WHERE` clause occurs in different ways for nonindexed and indexed lookups and leads to different results:

- If there is no index or the optimizer cannot use it, comparisons occur in the session time zone. The optimizer performs a table scan in which it retrieves each `ts` column value, converts it from UTC to the session time zone, and compares it to the search value (also interpreted in the session time zone):

  ```
  mysql> SELECT ts FROM tstable
         WHERE ts = '2018-10-28 02:30:00';
  +---------------------+
  | ts                  |
  +---------------------+
  | 2018-10-28 02:30:00 |
  | 2018-10-28 02:30:00 |
  +---------------------+
  ```

  Because the stored `ts` values are converted to the session time zone, it is possible for the query to return two timestamp values that are distinct as UTC values but equal in the session time zone: One value that occurs before the DST shift when clocks are changed, and one value that was occurs after the DST shift.

- If there is a usable index, comparisons occur in UTC. The optimizer performs an index scan, first converting the search value from the session time zone to UTC, then comparing the result to the UTC index entries:

  ```
  mysql> ALTER TABLE tstable ADD INDEX (ts);
  mysql> SELECT ts FROM tstable
         WHERE ts = '2018-10-28 02:30:00';
  +---------------------+
  | ts                  |
  ```

```
+---------------------+
| 2018-10-28 02:30:00 |
+---------------------+
```

In this case, the (converted) search value is matched only to index entries, and because the index entries for the distinct stored UTC values are also distinct, the search value can match only one of them.

Due to different optimizer operation for nonindexed and indexed lookups, the query produces different results in each case. The result from the nonindexed lookup returns all values that match in the session time zone. The indexed lookup cannot do so:

- It is performed within the storage engine, which knows only about UTC values.
- For the two distinct session time zone values that map to the same UTC value, the indexed lookup matches only the corresponding UTC index entry and returns only a single row.

In the preceding discussion, the data set stored in `tstable` happens to consist of distinct UTC values. In such cases, all index-using queries of the form shown match at most one index entry.

If the index is not `UNIQUE`, it is possible for the table (and the index) to store multiple instances of a given UTC value. For example, the `ts` column might contain multiple instances of the UTC value `'2018-10-28 00:30:00'`. In this case, the index-using query would return each of them (converted to the MET value `'2018-10-28 02:30:00'` in the result set). It remains true that index-using queries match the converted search value to a single value in the UTC index entries, rather than matching multiple UTC values that convert to the search value in the session time zone.

If it is important to return all `ts` values that match in the session time zone, the workaround is to suppress use of the index with an `IGNORE INDEX` hint:

```
mysql> SELECT ts FROM tstable
       IGNORE INDEX (ts)
       WHERE ts = '2018-10-28 02:30:00';
+---------------------+
| ts                  |
+---------------------+
| 2018-10-28 02:30:00 |
| 2018-10-28 02:30:00 |
+---------------------+
```

The same lack of one-to-one mapping for time zone conversions in both directions occurs in other contexts as well, such as conversions performed with the `FROM_UNIXTIME()` and `UNIX_TIMESTAMP()` functions. See Section 12.7, “Date and Time Functions”.

8.4 Optimizing Database Structure

In your role as a database designer, look for the most efficient way to organize your schemas, tables, and columns. As when tuning application code, you minimize I/O, keep related items together, and plan ahead so that performance stays high as the data volume increases. Starting with an efficient database design makes it easier for team members to write high-performing application code, and makes the database likely to endure as applications evolve and are rewritten.

8.4.1 Optimizing Data Size

Design your tables to minimize their space on the disk. This can result in huge improvements by reducing the amount of data written to and read from disk. Smaller tables normally require less main memory while their contents are being actively processed during query execution. Any space reduction for table data also results in smaller indexes that can be processed faster.

MySQL supports many different storage engines (table types) and row formats. For each table, you can decide which storage and indexing method to use. Choosing the proper table format for your application can give you a big performance gain. See Chapter 15, *The InnoDB Storage Engine*, and Chapter 16, *Alternative Storage Engines*.

You can get better performance for a table and minimize storage space by using the techniques listed here:

- Table Columns
- Row Format
- Indexes
- Joins
- Normalization

Table Columns

- Use the most efficient (smallest) data types possible. MySQL has many specialized types that save disk space and memory. For example, use the smaller integer types if possible to get smaller tables. `MEDIUMINT` is often a better choice than `INT` because a `MEDIUMINT` column uses 25% less space.
- Declare columns to be `NOT NULL` if possible. It makes SQL operations faster, by enabling better use of indexes and eliminating overhead for testing whether each value is `NULL`. You also save some storage space, one bit per column. If you really need `NULL` values in your tables, use them. Just avoid the default setting that allows `NULL` values in every column.

Row Format

- `InnoDB` tables are created using the `DYNAMIC` row format by default. To use a row format other than `DYNAMIC`, configure `innodb_default_row_format`, or specify the `ROW_FORMAT` option explicitly in a `CREATE TABLE` or `ALTER TABLE` statement.

  The compact family of row formats, which includes `COMPACT`, `DYNAMIC`, and `COMPRESSED`, decreases row storage space at the cost of increasing CPU use for some operations. If your workload is a typical one that is limited by cache hit rates and disk speed it is likely to be faster. If it is a rare case that is limited by CPU speed, it might be slower.

  The compact family of row formats also optimizes `CHAR` column storage when using a variable-length character set such as `utf8mb3` or `utf8mb4`. With `ROW_FORMAT=REDUNDANT`, `CHAR(N)` occupies *`N`* × the maximum byte length of the character set. Many languages can be written primarily using single-byte `utf8mb3`or `utf8mb4` characters, so a fixed storage length often wastes space. With the compact family of rows formats, `InnoDB` allocates a variable amount of storage in the range of *`N`* to *`N`* × the maximum byte length of the character set for these columns by stripping trailing spaces. The minimum storage length is *`N`* bytes to facilitate in-place updates in typical cases. For more information, see Section 15.10, “InnoDB Row Formats”.

- To minimize space even further by storing table data in compressed form, specify `ROW_FORMAT=COMPRESSED` when creating `InnoDB` tables, or run the `myisampack` command on an existing `MyISAM` table. (`InnoDB` compressed tables are readable and writable, while `MyISAM` compressed tables are read-only.)
- For `MyISAM` tables, if you do not have any variable-length columns (`VARCHAR`, `TEXT`, or `BLOB` columns), a fixed-size row format is used. This is faster but may waste some space. See Section 16.2.3, “MyISAM Table Storage Formats”. You can hint that you want to have fixed length rows even if you have `VARCHAR` columns with the `CREATE TABLE` option `ROW_FORMAT=FIXED`.

Indexes

- The primary index of a table should be as short as possible. This makes identification of each row easy and efficient. For `InnoDB` tables, the primary key columns are duplicated in each secondary index entry, so a short primary key saves considerable space if you have many secondary indexes.
- Create only the indexes that you need to improve query performance. Indexes are good for retrieval, but slow down insert and update operations. If you access a table mostly by searching on a

combination of columns, create a single composite index on them rather than a separate index for each column. The first part of the index should be the column most used. If you *always* use many columns when selecting from the table, the first column in the index should be the one with the most duplicates, to obtain better compression of the index.

- If it is very likely that a long string column has a unique prefix on the first number of characters, it is better to index only this prefix, using MySQL's support for creating an index on the leftmost part of the column (see Section 13.1.15, “CREATE INDEX Statement”). Shorter indexes are faster, not only because they require less disk space, but because they also give you more hits in the index cache, and thus fewer disk seeks. See Section 5.1.1, “Configuring the Server”.

Joins

- In some circumstances, it can be beneficial to split into two a table that is scanned very often. This is especially true if it is a dynamic-format table and it is possible to use a smaller static format table that can be used to find the relevant rows when scanning the table.
- Declare columns with identical information in different tables with identical data types, to speed up joins based on the corresponding columns.
- Keep column names simple, so that you can use the same name across different tables and simplify join queries. For example, in a table named `customer`, use a column name of `name` instead of `customer_name`. To make your names portable to other SQL servers, consider keeping them shorter than 18 characters.

Normalization

- Normally, try to keep all data nonredundant (observing what is referred to in database theory as *third normal form*). Instead of repeating lengthy values such as names and addresses, assign them unique IDs, repeat these IDs as needed across multiple smaller tables, and join the tables in queries by referencing the IDs in the join clause.
- If speed is more important than disk space and the maintenance costs of keeping multiple copies of data, for example in a business intelligence scenario where you analyze all the data from large tables, you can relax the normalization rules, duplicating information or creating summary tables to gain more speed.

8.4.2 Optimizing MySQL Data Types

8.4.2.1 Optimizing for Numeric Data

- For unique IDs or other values that can be represented as either strings or numbers, prefer numeric columns to string columns. Since large numeric values can be stored in fewer bytes than the corresponding strings, it is faster and takes less memory to transfer and compare them.
- If you are using numeric data, it is faster in many cases to access information from a database (using a live connection) than to access a text file. Information in the database is likely to be stored in a more compact format than in the text file, so accessing it involves fewer disk accesses. You also save code in your application because you can avoid parsing the text file to find line and column boundaries.

8.4.2.2 Optimizing for Character and String Types

For character and string columns, follow these guidelines:

- Use binary collation order for fast comparison and sort operations, when you do not need language-specific collation features. You can use the `BINARY` operator to use binary collation within a particular query.
- When comparing values from different columns, declare those columns with the same character set and collation wherever possible, to avoid string conversions while running the query.

- For column values less than 8KB in size, use binary `VARCHAR` instead of `BLOB`. The `GROUP BY` and `ORDER BY` clauses can generate temporary tables, and these temporary tables can use the `MEMORY` storage engine if the original table does not contain any `BLOB` columns.
- If a table contains string columns such as name and address, but many queries do not retrieve those columns, consider splitting the string columns into a separate table and using join queries with a foreign key when necessary. When MySQL retrieves any value from a row, it reads a data block containing all the columns of that row (and possibly other adjacent rows). Keeping each row small, with only the most frequently used columns, allows more rows to fit in each data block. Such compact tables reduce disk I/O and memory usage for common queries.
- When you use a randomly generated value as a primary key in an `InnoDB` table, prefix it with an ascending value such as the current date and time if possible. When consecutive primary values are physically stored near each other, `InnoDB` can insert and retrieve them faster.
- See Section 8.4.2.1, “Optimizing for Numeric Data” for reasons why a numeric column is usually preferable to an equivalent string column.

8.4.2.3 Optimizing for BLOB Types

- When storing a large blob containing textual data, consider compressing it first. Do not use this technique when the entire table is compressed by `InnoDB` or `MyISAM`.
- For a table with several columns, to reduce memory requirements for queries that do not use the BLOB column, consider splitting the BLOB column into a separate table and referencing it with a join query when needed.
- Since the performance requirements to retrieve and display a BLOB value might be very different from other data types, you could put the BLOB-specific table on a different storage device or even a separate database instance. For example, to retrieve a BLOB might require a large sequential disk read that is better suited to a traditional hard drive than to an SSD device.
- See Section 8.4.2.2, “Optimizing for Character and String Types” for reasons why a binary `VARCHAR` column is sometimes preferable to an equivalent BLOB column.
- Rather than testing for equality against a very long text string, you can store a hash of the column value in a separate column, index that column, and test the hashed value in queries. (Use the `MD5()` or `CRC32()` function to produce the hash value.) Since hash functions can produce duplicate results for different inputs, you still include a clause `AND` *`blob_column = long_string_value`* in the query to guard against false matches; the performance benefit comes from the smaller, easily scanned index for the hashed values.

8.4.3 Optimizing for Many Tables

Some techniques for keeping individual queries fast involve splitting data across many tables. When the number of tables runs into the thousands or even millions, the overhead of dealing with all these tables becomes a new performance consideration.

8.4.3.1 How MySQL Opens and Closes Tables

When you execute a `mysqladmin status` command, you should see something like this:

```
Uptime: 426 Running threads: 1 Questions: 11082
Reloads: 1 Open tables: 12
```

The `Open tables` value of 12 can be somewhat puzzling if you have fewer than 12 tables.

MySQL is multithreaded, so there may be many clients issuing queries for a given table simultaneously. To minimize the problem with multiple client sessions having different states on the same table, the table is opened independently by each concurrent session. This uses additional memory but normally increases performance. With `MyISAM` tables, one extra file descriptor is required

for the data file for each client that has the table open. (By contrast, the index file descriptor is shared between all sessions.)

The `table_open_cache` and `max_connections` system variables affect the maximum number of files the server keeps open. If you increase one or both of these values, you may run up against a limit imposed by your operating system on the per-process number of open file descriptors. Many operating systems permit you to increase the open-files limit, although the method varies widely from system to system. Consult your operating system documentation to determine whether it is possible to increase the limit and how to do so.

`table_open_cache` is related to `max_connections`. For example, for 200 concurrent running connections, specify a table cache size of at least `200 * N`, where *`N`* is the maximum number of tables per join in any of the queries which you execute. You must also reserve some extra file descriptors for temporary tables and files.

Make sure that your operating system can handle the number of open file descriptors implied by the `table_open_cache` setting. If `table_open_cache` is set too high, MySQL may run out of file descriptors and exhibit symptoms such as refusing connections or failing to perform queries.

Also take into account that the `MyISAM` storage engine needs two file descriptors for each unique open table. To increase the number of file descriptors available to MySQL, set the `open_files_limit` system variable. See Section B.3.2.16, “File Not Found and Similar Errors”.

The cache of open tables is kept at a level of `table_open_cache` entries. The server autosizes the cache size at startup. To set the size explicitly, set the `table_open_cache` system variable at startup. MySQL may temporarily open more tables than this to execute queries, as described later in this section.

MySQL closes an unused table and removes it from the table cache under the following circumstances:

- When the cache is full and a thread tries to open a table that is not in the cache.
- When the cache contains more than `table_open_cache` entries and a table in the cache is no longer being used by any threads.
- When a table-flushing operation occurs. This happens when someone issues a `FLUSH TABLES` statement or executes a `mysqladmin flush-tables` or `mysqladmin refresh` command.

When the table cache fills up, the server uses the following procedure to locate a cache entry to use:

- Tables not currently in use are released, beginning with the table least recently used.
- If a new table must be opened, but the cache is full and no tables can be released, the cache is temporarily extended as necessary. When the cache is in a temporarily extended state and a table goes from a used to unused state, the table is closed and released from the cache.

A `MyISAM` table is opened for each concurrent access. This means the table needs to be opened twice if two threads access the same table or if a thread accesses the table twice in the same query (for example, by joining the table to itself). Each concurrent open requires an entry in the table cache. The first open of any `MyISAM` table takes two file descriptors: one for the data file and one for the index file. Each additional use of the table takes only one file descriptor for the data file. The index file descriptor is shared among all threads.

If you are opening a table with the `HANDLER tbl_name OPEN` statement, a dedicated table object is allocated for the thread. This table object is not shared by other threads and is not closed until the thread calls `HANDLER tbl_name CLOSE` or the thread terminates. When this happens, the table is put back in the table cache (if the cache is not full). See Section 13.2.5, “HANDLER Statement”.

To determine whether your table cache is too small, check the `Opened_tables` status variable, which indicates the number of table-opening operations since the server started:

```
mysql> SHOW GLOBAL STATUS LIKE 'Opened_tables';
+---------------+-------+
```

```
| Variable_name | Value |
+---------------+-------+
| Opened_tables | 2741  |
+---------------+-------+
```

If the value is very large or increases rapidly, even when you have not issued many `FLUSH TABLES` statements, increase the `table_open_cache` value at server startup.

8.4.3.2 Disadvantages of Creating Many Tables in the Same Database

If you have many `MyISAM` tables in the same database directory, open, close, and create operations are slow. If you execute `SELECT` statements on many different tables, there is a little overhead when the table cache is full, because for every table that has to be opened, another must be closed. You can reduce this overhead by increasing the number of entries permitted in the table cache.

8.4.4 Internal Temporary Table Use in MySQL

In some cases, the server creates internal temporary tables while processing statements. Users have no direct control over when this occurs.

The server creates temporary tables under conditions such as these:

- Evaluation of `UNION` statements, with some exceptions described later.
- Evaluation of some views, such those that use the `TEMPTABLE` algorithm, `UNION`, or aggregation.
- Evaluation of derived tables (see Section 13.2.15.8, “Derived Tables”).
- Evaluation of common table expressions (see Section 13.2.20, “WITH (Common Table Expressions)”).
- Tables created for subquery or semijoin materialization (see Section 8.2.2, “Optimizing Subqueries, Derived Tables, View References, and Common Table Expressions”).
- Evaluation of statements that contain an `ORDER BY` clause and a different `GROUP BY` clause, or for which the `ORDER BY` or `GROUP BY` contains columns from tables other than the first table in the join queue.
- Evaluation of `DISTINCT` combined with `ORDER BY` may require a temporary table.
- For queries that use the `SQL_SMALL_RESULT` modifier, MySQL uses an in-memory temporary table, unless the query also contains elements (described later) that require on-disk storage.
- To evaluate `INSERT ... SELECT` statements that select from and insert into the same table, MySQL creates an internal temporary table to hold the rows from the `SELECT`, then inserts those rows into the target table. See Section 13.2.7.1, “INSERT ... SELECT Statement”.
- Evaluation of multiple-table `UPDATE` statements.
- Evaluation of `GROUP_CONCAT()` or `COUNT(DISTINCT)` expressions.
- Evaluation of window functions (see Section 12.21, “Window Functions”) uses temporary tables as necessary.

To determine whether a statement requires a temporary table, use `EXPLAIN` and check the `Extra` column to see whether it says `Using temporary` (see Section 8.8.1, “Optimizing Queries with EXPLAIN”). `EXPLAIN` does not necessarily say `Using temporary` for derived or materialized temporary tables. For statements that use window functions, `EXPLAIN` with `FORMAT=JSON` always provides information about the windowing steps. If the windowing functions use temporary tables, it is indicated for each step.

Some query conditions prevent the use of an in-memory temporary table, in which case the server uses an on-disk table instead:

- Presence of a `BLOB` or `TEXT` column in the table. However, the `TempTable` storage engine, which is the default storage engine for in-memory internal temporary tables in MySQL 8.0, supports binary large object types as of MySQL 8.0.13. See Internal Temporary Table Storage Engine.
- Presence of any string column with a maximum length larger than 512 (bytes for binary strings, characters for nonbinary strings) in the `SELECT` list, if `UNION` or `UNION ALL` is used.
- The `SHOW COLUMNS` and `DESCRIBE` statements use `BLOB` as the type for some columns, thus the temporary table used for the results is an on-disk table.

The server does not use a temporary table for `UNION` statements that meet certain qualifications. Instead, it retains from temporary table creation only the data structures necessary to perform result column typecasting. The table is not fully instantiated and no rows are written to or read from it; rows are sent directly to the client. The result is reduced memory and disk requirements, and smaller delay before the first row is sent to the client because the server need not wait until the last query block is executed. `EXPLAIN` and optimizer trace output reflects this execution strategy: The `UNION RESULT` query block is not present because that block corresponds to the part that reads from the temporary table.

These conditions qualify a `UNION` for evaluation without a temporary table:

- The union is `UNION ALL`, not `UNION` or `UNION DISTINCT`.
- There is no global `ORDER BY` clause.
- The union is not the top-level query block of an `{INSERT | REPLACE} ... SELECT ...` statement.

Internal Temporary Table Storage Engine

An internal temporary table can be held in memory and processed by the `TempTable` or `MEMORY` storage engine, or stored on disk by the `InnoDB` storage engine.

Storage Engine for In-Memory Internal Temporary Tables

The `internal_tmp_mem_storage_engine` variable defines the storage engine used for in-memory internal temporary tables. Permitted values are `TempTable` (the default) and `MEMORY`.

Note

As of MySQL 8.0.27, configuring a session setting for `internal_tmp_mem_storage_engine` requires the `SESSION_VARIABLES_ADMIN` or `SYSTEM_VARIABLES_ADMIN` privilege.

The `TempTable` storage engine provides efficient storage for `VARCHAR` and `VARBINARY` columns, and other binary large object types as of MySQL 8.0.13.

The following variables control TempTable storage engine limits and behavior:

- `tmp_table_size`: From MySQL 8.0.28, `tmp_table_size` defines the maximum size of any individual in-memory internal temporary table created by the TempTable storage engine. When the `tmp_table_size` limit is reached, MySQL automatically converts the in-memory internal temporary table to an `InnoDB` on-disk internal temporary table. The default `tmp_table_size` setting is 16777216 bytes (16 MiB).

  The `tmp_table_size` limit is intended to prevent individual queries from consuming an inordinate amount global TempTable resources, which can affect the performance of concurrent queries that require TempTable resources. Global TempTable resources are controlled by the `temptable_max_ram` and `temptable_max_mmap` settings.

  If the `tmp_table_size` limit is less than the `temptable_max_ram` limit, it is not possible for an in-memory temporary table to contain more data than permitted by the `tmp_table_size` limit. If the `tmp_table_size` limit is greater than the sum of the `temptable_max_ram` and

`temptable_max_mmap` limits, it is not possible for an in-memory temporary table to contain more than the sum of the `temptable_max_ram` and `temptable_max_mmap` limits.

- `temptable_max_ram`: Defines the maximum amount of RAM that can be used by the `TempTable` storage engine before it starts allocating space from memory-mapped files or before MySQL starts using `InnoDB` on-disk internal temporary tables, depending on your configuration. The default `temptable_max_ram` setting is 1073741824 bytes (1GiB).

  

  Note

  The `temptable_max_ram` setting does not account for the thread-local memory block allocated to each thread that uses the `TempTable` storage engine. The size of the thread-local memory block depends on the size of the thread's first memory allocation request. If the request is less than 1MB, which it is in most cases, the thread-local memory block size is 1MB. If the request is greater than 1MB, the thread-local memory block is approximately the same size as the initial memory request. The thread-local memory block is held in thread-local storage until thread exit.

- `temptable_use_mmap`: Controls whether the `TempTable` storage engine allocates space from memory-mapped files or MySQL uses `InnoDB` on-disk internal temporary tables when the `temptable_max_ram` limit is exceeded. The default setting is `temptable_use_mmap=ON`.

  

  Note

  The `temptable_use_mmap` variable was introduced in MySQL 8.0.16 and deprecated in MySQL 8.0.26; expect support for it to be removed in a future version of MySQL. Setting `temptable_max_mmap=0` is equivalent to setting `temptable_use_mmap=OFF`.

- `temptable_max_mmap`: Introduced in MySQL 8.0.23. Defines the maximum amount of memory the TempTable storage engine is permitted to allocate from memory-mapped files before MySQL starts using `InnoDB` on-disk internal temporary tables. The default setting is 1073741824 bytes (1GiB). The limit is intended to address the risk of memory mapped files using too much space in the temporary directory (`tmpdir`). A `temptable_max_mmap=0` setting disables allocation from memory-mapped files, effectively disabling their use, regardless of the `temptable_use_mmap` setting.

Use of memory-mapped files by the `TempTable` storage engine is governed by these rules:

- Temporary files are created in the directory defined by the `tmpdir` variable.
- Temporary files are deleted immediately after they are created and opened, and therefore do not remain visible in the `tmpdir` directory. The space occupied by temporary files is held by the operating system while temporary files are open. The space is reclaimed when temporary files are closed by the `TempTable` storage engine, or when the `mysqld` process is shut down.
- Data is never moved between RAM and temporary files, within RAM, or between temporary files.
- New data is stored in RAM if space becomes available within the limit defined by `temptable_max_ram`. Otherwise, new data is stored in temporary files.
- If space becomes available in RAM after some of the data for a table is written to temporary files, it is possible for the remaining table data to be stored in RAM.

When using the `MEMORY` storage engine for in-memory temporary tables (`internal_tmp_mem_storage_engine=MEMORY`), MySQL automatically converts an in-memory temporary table to an on-disk table if it becomes too large. The maximum size of an in-memory temporary table is defined by the `tmp_table_size` or `max_heap_table_size` value, whichever is smaller. This differs from `MEMORY` tables explicitly created with `CREATE TABLE`. For such tables, only the `max_heap_table_size` variable determines how large a table can grow, and there is no conversion to on-disk format.

Storage Engine for On-Disk Internal Temporary Tables

In MySQL 8.0.15 and earlier, the `internal_tmp_disk_storage_engine` variable defined the storage engine used for on-disk internal temporary tables. Supported storage engines were `InnoDB` and `MyISAM`.

From MySQL 8.0.16, MySQL uses only the `InnoDB` storage engine for on-disk internal temporary tables. The `MYISAM` storage engine is no longer supported for this purpose.

`InnoDB` on-disk internal temporary tables are created in session temporary tablespaces that reside in the data directory by default. For more information, see Section 15.6.3.5, “Temporary Tablespaces”.

In MySQL 8.0.15 and earlier:

- For common table expressions (CTEs), the storage engine used for on-disk internal temporary tables cannot be `MyISAM`. If `internal_tmp_disk_storage_engine=MYISAM`, an error occurs for any attempt to materialize a CTE using an on-disk temporary table.
- When using `internal_tmp_disk_storage_engine=INNODB`, queries that generate on-disk internal temporary tables that exceed `InnoDB` row or column limits return `Row size too large` or `Too many columns` errors. The workaround is to set `internal_tmp_disk_storage_engine` to `MYISAM`.

Internal Temporary Table Storage Format

When in-memory internal temporary tables are managed by the `TempTable` storage engine, rows that include `VARCHAR` columns, `VARBINARY` columns, and other binary large object type columns (supported as of MySQL 8.0.13) are represented in memory by an array of cells, with each cell containing a NULL flag, the data length, and a data pointer. Column values are placed in consecutive order after the array, in a single region of memory, without padding. Each cell in the array uses 16 bytes of storage. The same storage format applies when the `TempTable` storage engine allocates space from memory-mapped files.

When in-memory internal temporary tables are managed by the `MEMORY` storage engine, fixed-length row format is used. `VARCHAR` and `VARBINARY` column values are padded to the maximum column length, in effect storing them as `CHAR` and `BINARY` columns.

Prior to MySQL 8.0.16, on-disk internal temporary tables were managed by the `InnoDB` or `MyISAM` storage engine (depending on the `internal_tmp_disk_storage_engine` setting). Both engines store internal temporary tables using dynamic-width row format. Columns take only as much storage as needed, which reduces disk I/O, space requirements, and processing time compared to on-disk tables that use fixed-length rows. Beginning with MySQL 8.0.16, `internal_tmp_disk_storage_engine` is not supported, and internal temporary tables on disk are always managed by `InnoDB`.

When using the `MEMORY` storage engine, statements can initially create an in-memory internal temporary table and then convert it to an on-disk table if the table becomes too large. In such cases, better performance might be achieved by skipping the conversion and creating the internal temporary table on disk to begin with. The `big_tables` variable can be used to force disk storage of internal temporary tables.

Monitoring Internal Temporary Table Creation

When an internal temporary table is created in memory or on disk, the server increments the `Created_tmp_tables` value. When an internal temporary table is created on disk, the server increments the `Created_tmp_disk_tables` value. If too many internal temporary tables are created on disk, consider adjusting the engine-specific limits described in Internal Temporary Table Storage Engine.

Note

Due to a known limitation, `Created_tmp_disk_tables` does not count on-disk temporary tables created in memory-mapped files. By default, the

> TempTable storage engine overflow mechanism creates internal temporary tables in memory-mapped files. See Internal Temporary Table Storage Engine.

The `memory/temptable/physical_ram` and `memory/temptable/physical_disk` Performance Schema instruments can be used to monitor `TempTable` space allocation from memory and disk. `memory/temptable/physical_ram` reports the amount of allocated RAM. `memory/temptable/physical_disk` reports the amount of space allocated from disk when memory-mapped files are used as the TempTable overflow mechanism. If the `physical_disk` instrument reports a value other than 0 and memory-mapped files are used as the TempTable overflow mechanism, a TempTable memory limit was reached at some point. Data can be queried in Performance Schema memory summary tables such as `memory_summary_global_by_event_name`. See Section 27.12.20.10, "Memory Summary Tables".

8.4.5 Limits on Number of Databases and Tables

MySQL has no limit on the number of databases. The underlying file system may have a limit on the number of directories.

MySQL has no limit on the number of tables. The underlying file system may have a limit on the number of files that represent tables. Individual storage engines may impose engine-specific constraints. `InnoDB` permits up to 4 billion tables.

8.4.6 Limits on Table Size

The effective maximum table size for MySQL databases is usually determined by operating system constraints on file sizes, not by MySQL internal limits. For up-to-date information operating system file size limits, refer to the documentation specific to your operating system.

Windows users, please note that FAT and VFAT (FAT32) are *not* considered suitable for production use with MySQL. Use NTFS instead.

If you encounter a full-table error, there are several reasons why it might have occurred:

- The disk might be full.
- You are using `InnoDB` tables and have run out of room in an `InnoDB` tablespace file. The maximum tablespace size is also the maximum size for a table. For tablespace size limits, see Section 15.22, "InnoDB Limits".

  Generally, partitioning of tables into multiple tablespace files is recommended for tables larger than 1TB in size.
- You have hit an operating system file size limit. For example, you are using `MyISAM` tables on an operating system that supports files only up to 2GB in size and you have hit this limit for the data file or index file.
- You are using a `MyISAM` table and the space required for the table exceeds what is permitted by the internal pointer size. `MyISAM` permits data and index files to grow up to 256TB by default, but this limit can be changed up to the maximum permissible size of 65,536TB ($256^7 - 1$ bytes).

  If you need a `MyISAM` table that is larger than the default limit and your operating system supports large files, the `CREATE TABLE` statement supports `AVG_ROW_LENGTH` and `MAX_ROWS` options. See Section 13.1.20, "CREATE TABLE Statement". The server uses these options to determine how large a table to permit.

  If the pointer size is too small for an existing table, you can change the options with `ALTER TABLE` to increase a table's maximum permissible size. See Section 13.1.9, "ALTER TABLE Statement".

  ```
  ALTER TABLE tbl_name MAX_ROWS=1000000000 AVG_ROW_LENGTH=nnn;
  ```

  You have to specify `AVG_ROW_LENGTH` only for tables with `BLOB` or `TEXT` columns; in this case, MySQL cannot optimize the space required based only on the number of rows.

To change the default size limit for `MyISAM` tables, set the `myisam_data_pointer_size`, which sets the number of bytes used for internal row pointers. The value is used to set the pointer size for new tables if you do not specify the `MAX_ROWS` option. The value of `myisam_data_pointer_size` can be from 2 to 7. For example, for tables that use the dynamic storage format, a value of 4 permits tables up to 4GB; a value of 6 permits tables up to 256TB. Tables that use the fixed storage format have a larger maximum data length. For storage format characteristics, see Section 16.2.3, "MyISAM Table Storage Formats".

You can check the maximum data and index sizes by using this statement:

```
SHOW TABLE STATUS FROM db_name LIKE 'tbl_name';
```

You also can use `myisamchk -dv /path/to/table-index-file`. See Section 13.7.7, "SHOW Statements", or Section 4.6.4, "myisamchk — MyISAM Table-Maintenance Utility".

Other ways to work around file-size limits for `MyISAM` tables are as follows:

- If your large table is read only, you can use `myisampack` to compress it. `myisampack` usually compresses a table by at least 50%, so you can have, in effect, much bigger tables. `myisampack` also can merge multiple tables into a single table. See Section 4.6.6, "myisampack — Generate Compressed, Read-Only MyISAM Tables".
- MySQL includes a `MERGE` library that enables you to handle a collection of `MyISAM` tables that have identical structure as a single `MERGE` table. See Section 16.7, "The MERGE Storage Engine".

- You are using the `MEMORY` (`HEAP`) storage engine; in this case you need to increase the value of the `max_heap_table_size` system variable. See Section 5.1.8, "Server System Variables".

8.4.7 Limits on Table Column Count and Row Size

This section describes limits on the number of columns in tables and the size of individual rows.

- Column Count Limits
- Row Size Limits

Column Count Limits

MySQL has hard limit of 4096 columns per table, but the effective maximum may be less for a given table. The exact column limit depends on several factors:

- The maximum row size for a table constrains the number (and possibly size) of columns because the total length of all columns cannot exceed this size. See Row Size Limits.
- The storage requirements of individual columns constrain the number of columns that fit within a given maximum row size. Storage requirements for some data types depend on factors such as storage engine, storage format, and character set. See Section 11.7, "Data Type Storage Requirements".
- Storage engines may impose additional restrictions that limit table column count. For example, `InnoDB` has a limit of 1017 columns per table. See Section 15.22, "InnoDB Limits". For information about other storage engines, see Chapter 16, *Alternative Storage Engines*.
- Functional key parts (see Section 13.1.15, "CREATE INDEX Statement") are implemented as hidden virtual generated stored columns, so each functional key part in a table index counts against the table total column limit.

Row Size Limits

The maximum row size for a given table is determined by several factors:

- The internal representation of a MySQL table has a maximum row size limit of 65,535 bytes, even if the storage engine is capable of supporting larger rows. `BLOB` and `TEXT` columns only contribute 9 to 12 bytes toward the row size limit because their contents are stored separately from the rest of the row.
- The maximum row size for an `InnoDB` table, which applies to data stored locally within a database page, is slightly less than half a page for 4KB, 8KB, 16KB, and 32KB `innodb_page_size` settings. For example, the maximum row size is slightly less than 8KB for the default 16KB `InnoDB` page size. For 64KB pages, the maximum row size is slightly less than 16KB. See Section 15.22, “InnoDB Limits”.

  If a row containing variable-length columns exceeds the `InnoDB` maximum row size, `InnoDB` selects variable-length columns for external off-page storage until the row fits within the `InnoDB` row size limit. The amount of data stored locally for variable-length columns that are stored off-page differs by row format. For more information, see Section 15.10, “InnoDB Row Formats”.
- Different storage formats use different amounts of page header and trailer data, which affects the amount of storage available for rows.
  - For information about `InnoDB` row formats, see Section 15.10, “InnoDB Row Formats”.
  - For information about `MyISAM` storage formats, see Section 16.2.3, “MyISAM Table Storage Formats”.

Row Size Limit Examples

- The MySQL maximum row size limit of 65,535 bytes is demonstrated in the following `InnoDB` and `MyISAM` examples. The limit is enforced regardless of storage engine, even though the storage engine may be capable of supporting larger rows.

```
mysql> CREATE TABLE t (a VARCHAR(10000), b VARCHAR(10000),
       c VARCHAR(10000), d VARCHAR(10000), e VARCHAR(10000),
       f VARCHAR(10000), g VARCHAR(6000)) ENGINE=InnoDB CHARACTER SET latin1;
ERROR 1118 (42000): Row size too large. The maximum row size for the used
table type, not counting BLOBs, is 65535. This includes storage overhead,
check the manual. You have to change some columns to TEXT or BLOBs
```

```
mysql> CREATE TABLE t (a VARCHAR(10000), b VARCHAR(10000),
       c VARCHAR(10000), d VARCHAR(10000), e VARCHAR(10000),
       f VARCHAR(10000), g VARCHAR(6000)) ENGINE=MyISAM CHARACTER SET latin1;
ERROR 1118 (42000): Row size too large. The maximum row size for the used
table type, not counting BLOBs, is 65535. This includes storage overhead,
check the manual. You have to change some columns to TEXT or BLOBs
```

  In the following `MyISAM` example, changing a column to `TEXT` avoids the 65,535-byte row size limit and permits the operation to succeed because `BLOB` and `TEXT` columns only contribute 9 to 12 bytes toward the row size.

```
mysql> CREATE TABLE t (a VARCHAR(10000), b VARCHAR(10000),
       c VARCHAR(10000), d VARCHAR(10000), e VARCHAR(10000),
       f VARCHAR(10000), g TEXT(6000)) ENGINE=MyISAM CHARACTER SET latin1;
Query OK, 0 rows affected (0.02 sec)
```

  The operation succeeds for an `InnoDB` table because changing a column to `TEXT` avoids the MySQL 65,535-byte row size limit, and `InnoDB` off-page storage of variable-length columns avoids the `InnoDB` row size limit.

```
mysql> CREATE TABLE t (a VARCHAR(10000), b VARCHAR(10000),
       c VARCHAR(10000), d VARCHAR(10000), e VARCHAR(10000),
       f VARCHAR(10000), g TEXT(6000)) ENGINE=InnoDB CHARACTER SET latin1;
Query OK, 0 rows affected (0.02 sec)
```

- Storage for variable-length columns includes length bytes, which are counted toward the row size. For example, a `VARCHAR(255) CHARACTER SET utf8mb3` column takes two bytes to store the length of the value, so each value can take up to 767 bytes.

The statement to create table `t1` succeeds because the columns require 32,765 + 2 bytes and 32,766 + 2 bytes, which falls within the maximum row size of 65,535 bytes:

```
mysql> CREATE TABLE t1
       (c1 VARCHAR(32765) NOT NULL, c2 VARCHAR(32766) NOT NULL)
       ENGINE = InnoDB CHARACTER SET latin1;
Query OK, 0 rows affected (0.02 sec)
```

The statement to create table `t2` fails because, although the column length is within the maximum length of 65,535 bytes, two additional bytes are required to record the length, which causes the row size to exceed 65,535 bytes:

```
mysql> CREATE TABLE t2
       (c1 VARCHAR(65535) NOT NULL)
       ENGINE = InnoDB CHARACTER SET latin1;
ERROR 1118 (42000): Row size too large. The maximum row size for the used
table type, not counting BLOBs, is 65535. This includes storage overhead,
check the manual. You have to change some columns to TEXT or BLOBs
```

Reducing the column length to 65,533 or less permits the statement to succeed.

```
mysql> CREATE TABLE t2
       (c1 VARCHAR(65533) NOT NULL)
       ENGINE = InnoDB CHARACTER SET latin1;
Query OK, 0 rows affected (0.01 sec)
```

- For `MyISAM` tables, `NULL` columns require additional space in the row to record whether their values are `NULL`. Each `NULL` column takes one bit extra, rounded up to the nearest byte.

  The statement to create table `t3` fails because `MyISAM` requires space for `NULL` columns in addition to the space required for variable-length column length bytes, causing the row size to exceed 65,535 bytes:

  ```
  mysql> CREATE TABLE t3
         (c1 VARCHAR(32765) NULL, c2 VARCHAR(32766) NULL)
         ENGINE = MyISAM CHARACTER SET latin1;
  ERROR 1118 (42000): Row size too large. The maximum row size for the used
  table type, not counting BLOBs, is 65535. This includes storage overhead,
  check the manual. You have to change some columns to TEXT or BLOBs
  ```

  For information about `InnoDB` `NULL` column storage, see Section 15.10, “InnoDB Row Formats”.

- `InnoDB` restricts row size (for data stored locally within the database page) to slightly less than half a database page for 4KB, 8KB, 16KB, and 32KB `innodb_page_size` settings, and to slightly less than 16KB for 64KB pages.

  The statement to create table `t4` fails because the defined columns exceed the row size limit for a 16KB `InnoDB` page.

  ```
  mysql> CREATE TABLE t4 (
         c1 CHAR(255),c2 CHAR(255),c3 CHAR(255),
         c4 CHAR(255),c5 CHAR(255),c6 CHAR(255),
         c7 CHAR(255),c8 CHAR(255),c9 CHAR(255),
         c10 CHAR(255),c11 CHAR(255),c12 CHAR(255),
         c13 CHAR(255),c14 CHAR(255),c15 CHAR(255),
         c16 CHAR(255),c17 CHAR(255),c18 CHAR(255),
         c19 CHAR(255),c20 CHAR(255),c21 CHAR(255),
         c22 CHAR(255),c23 CHAR(255),c24 CHAR(255),
         c25 CHAR(255),c26 CHAR(255),c27 CHAR(255),
         c28 CHAR(255),c29 CHAR(255),c30 CHAR(255),
         c31 CHAR(255),c32 CHAR(255),c33 CHAR(255)
         ) ENGINE=InnoDB ROW_FORMAT=DYNAMIC DEFAULT CHARSET latin1;
  ERROR 1118 (42000): Row size too large (> 8126). Changing some columns to TEXT or BLOB may help.
  In current row format, BLOB prefix of 0 bytes is stored inline.
  ```

8.5 Optimizing for InnoDB Tables

`InnoDB` is the storage engine that MySQL customers typically use in production databases where reliability and concurrency are important. `InnoDB` is the default storage engine in MySQL. This section explains how to optimize database operations for `InnoDB` tables.

8.5.1 Optimizing Storage Layout for InnoDB Tables

- Once your data reaches a stable size, or a growing table has increased by tens or some hundreds of megabytes, consider using the `OPTIMIZE TABLE` statement to reorganize the table and compact any wasted space. The reorganized tables require less disk I/O to perform full table scans. This is a straightforward technique that can improve performance when other techniques such as improving index usage or tuning application code are not practical.

  `OPTIMIZE TABLE` copies the data part of the table and rebuilds the indexes. The benefits come from improved packing of data within indexes, and reduced fragmentation within the tablespaces and on disk. The benefits vary depending on the data in each table. You may find that there are significant gains for some and not for others, or that the gains decrease over time until you next optimize the table. This operation can be slow if the table is large or if the indexes being rebuilt do not fit into the buffer pool. The first run after adding a lot of data to a table is often much slower than later runs.

- In `InnoDB`, having a long `PRIMARY KEY` (either a single column with a lengthy value, or several columns that form a long composite value) wastes a lot of disk space. The primary key value for a row is duplicated in all the secondary index records that point to the same row. (See Section 15.6.2.1, “Clustered and Secondary Indexes”.) Create an `AUTO_INCREMENT` column as the primary key if your primary key is long, or index a prefix of a long `VARCHAR` column instead of the entire column.

- Use the `VARCHAR` data type instead of `CHAR` to store variable-length strings or for columns with many `NULL` values. A `CHAR(N)` column always takes *`N`* characters to store data, even if the string is shorter or its value is `NULL`. Smaller tables fit better in the buffer pool and reduce disk I/O.

  When using `COMPACT` row format (the default `InnoDB` format) and variable-length character sets, such as `utf8mb4` or `sjis`, `CHAR(N)` columns occupy a variable amount of space, but still at least *`N`* bytes.

- For tables that are big, or contain lots of repetitive text or numeric data, consider using `COMPRESSED` row format. Less disk I/O is required to bring data into the buffer pool, or to perform full table scans. Before making a permanent decision, measure the amount of compression you can achieve by using `COMPRESSED` versus `COMPACT` row format.

8.5.2 Optimizing InnoDB Transaction Management

To optimize `InnoDB` transaction processing, find the ideal balance between the performance overhead of transactional features and the workload of your server. For example, an application might encounter performance issues if it commits thousands of times per second, and different performance issues if it commits only every 2-3 hours.

- The default MySQL setting `AUTOCOMMIT=1` can impose performance limitations on a busy database server. Where practical, wrap several related data change operations into a single transaction, by issuing `SET AUTOCOMMIT=0` or a `START TRANSACTION` statement, followed by a `COMMIT` statement after making all the changes.

  `InnoDB` must flush the log to disk at each transaction commit if that transaction made modifications to the database. When each change is followed by a commit (as with the default autocommit setting), the I/O throughput of the storage device puts a cap on the number of potential operations per second.

- Alternatively, for transactions that consist only of a single `SELECT` statement, turning on `AUTOCOMMIT` helps `InnoDB` to recognize read-only transactions and optimize them. See Section 8.5.3, “Optimizing InnoDB Read-Only Transactions” for requirements.

- Avoid performing rollbacks after inserting, updating, or deleting huge numbers of rows. If a big transaction is slowing down server performance, rolling it back can make the problem worse, potentially taking several times as long to perform as the original data change operations. Killing the database process does not help, because the rollback starts again on server startup.

  To minimize the chance of this issue occurring:

  - Increase the size of the buffer pool so that all the data change changes can be cached rather than immediately written to disk.
  - Set `innodb_change_buffering=all` so that update and delete operations are buffered in addition to inserts.
  - Consider issuing `COMMIT` statements periodically during the big data change operation, possibly breaking a single delete or update into multiple statements that operate on smaller numbers of rows.

  To get rid of a runaway rollback once it occurs, increase the buffer pool so that the rollback becomes CPU-bound and runs fast, or kill the server and restart with `innodb_force_recovery=3`, as explained in Section 15.18.2, “InnoDB Recovery”.

  This issue is expected to be infrequent with the default setting `innodb_change_buffering=all`, which allows update and delete operations to be cached in memory, making them faster to perform in the first place, and also faster to roll back if needed. Make sure to use this parameter setting on servers that process long-running transactions with many inserts, updates, or deletes.

- If you can afford the loss of some of the latest committed transactions if an unexpected exit occurs, you can set the `innodb_flush_log_at_trx_commit` parameter to 0. `InnoDB` tries to flush the log once per second anyway, although the flush is not guaranteed.

- When rows are modified or deleted, the rows and associated undo logs are not physically removed immediately, or even immediately after the transaction commits. The old data is preserved until transactions that started earlier or concurrently are finished, so that those transactions can access the previous state of modified or deleted rows. Thus, a long-running transaction can prevent `InnoDB` from purging data that was changed by a different transaction.

- When rows are modified or deleted within a long-running transaction, other transactions using the `READ COMMITTED` and `REPEATABLE READ` isolation levels have to do more work to reconstruct the older data if they read those same rows.

- When a long-running transaction modifies a table, queries against that table from other transactions do not make use of the covering index technique. Queries that normally could retrieve all the result columns from a secondary index, instead look up the appropriate values from the table data.

  If secondary index pages are found to have a `PAGE_MAX_TRX_ID` that is too new, or if records in the secondary index are delete-marked, `InnoDB` may need to look up records using a clustered index.

8.5.3 Optimizing InnoDB Read-Only Transactions

`InnoDB` can avoid the overhead associated with setting up the transaction ID (`TRX_ID` field) for transactions that are known to be read-only. A transaction ID is only needed for a transaction that might perform write operations or locking reads such as `SELECT ... FOR UPDATE`. Eliminating unnecessary transaction IDs reduces the size of internal data structures that are consulted each time a query or data change statement constructs a read view.

`InnoDB` detects read-only transactions when:

- The transaction is started with the `START TRANSACTION READ ONLY` statement. In this case, attempting to make changes to the database (for `InnoDB`, `MyISAM`, or other types of tables) causes an error, and the transaction continues in read-only state:

```
ERROR 1792 (25006): Cannot execute statement in a READ ONLY transaction.
```

You can still make changes to session-specific temporary tables in a read-only transaction, or issue locking queries for them, because those changes and locks are not visible to any other transaction.

- The `autocommit` setting is turned on, so that the transaction is guaranteed to be a single statement, and the single statement making up the transaction is a "non-locking" `SELECT` statement. That is, a `SELECT` that does not use a `FOR UPDATE` or `LOCK IN SHARED MODE` clause.
- The transaction is started without the `READ ONLY` option, but no updates or statements that explicitly lock rows have been executed yet. Until updates or explicit locks are required, a transaction stays in read-only mode.

Thus, for a read-intensive application such as a report generator, you can tune a sequence of `InnoDB` queries by grouping them inside `START TRANSACTION READ ONLY` and `COMMIT`, or by turning on the `autocommit` setting before running the `SELECT` statements, or simply by avoiding any data change statements interspersed with the queries.

For information about `START TRANSACTION` and `autocommit`, see Section 13.3.1, "START TRANSACTION, COMMIT, and ROLLBACK Statements".

Note

Transactions that qualify as auto-commit, non-locking, and read-only (AC-NL-RO) are kept out of certain internal `InnoDB` data structures and are therefore not listed in `SHOW ENGINE INNODB STATUS` output.

8.5.4 Optimizing InnoDB Redo Logging

Consider the following guidelines for optimizing redo logging:

- Increase the size of your redo log files. When `InnoDB` has written redo log files full, it must write the modified contents of the buffer pool to disk in a checkpoint. Small redo log files cause many unnecessary disk writes.

  From MySQL 8.0.30, the redo log file size is determined by the `innodb_redo_log_capacity` setting. `InnoDB` tries to maintain 32 redo log files of the same size, with each file equal to 1/32 * `innodb_redo_log_capacity`. Therefore, changing the `innodb_redo_log_capacity` setting changes the size of the redo log files.

  Before MySQL 8.0.30, the size and number of redo log files are configured using the `innodb_log_file_size` and `innodb_log_files_in_group` variables.

  For information about modifying your redo log file configuration, see Section 15.6.5, "Redo Log".

- Consider increasing the size of the log buffer. A large log buffer enables large transactions to run without a need to write the log to disk before the transactions commit. Thus, if you have transactions that update, insert, or delete many rows, making the log buffer larger saves disk I/O. Log buffer size is configured using the `innodb_log_buffer_size` configuration option, which can be configured dynamically in MySQL 8.0.
- Configure the `innodb_log_write_ahead_size` configuration option to avoid "read-on-write". This option defines the write-ahead block size for the redo log. Set `innodb_log_write_ahead_size` to match the operating system or file system cache block size. Read-on-write occurs when redo log blocks are not entirely cached to the operating system or file system due to a mismatch between write-ahead block size for the redo log and operating system or file system cache block size.

  Valid values for `innodb_log_write_ahead_size` are multiples of the `InnoDB` log file block size (2^n). The minimum value is the `InnoDB` log file block size (512). Write-ahead does not occur

when the minimum value is specified. The maximum value is equal to the `innodb_page_size` value. If you specify a value for `innodb_log_write_ahead_size` that is larger than the `innodb_page_size` value, the `innodb_log_write_ahead_size` setting is truncated to the `innodb_page_size` value.

Setting the `innodb_log_write_ahead_size` value too low in relation to the operating system or file system cache block size results in read-on-write. Setting the value too high may have a slight impact on `fsync` performance for log file writes due to several blocks being written at once.

- MySQL 8.0.11 introduced dedicated log writer threads for writing redo log records from the log buffer to the system buffers and flushing the system buffers to the redo log files. Previously, individual user threads were responsible those tasks. As of MySQL 8.0.22, you can enable or disable log writer threads using the `innodb_log_writer_threads` variable. Dedicated log writer threads can improve performance on high-concurrency systems, but for low-concurrency systems, disabling dedicated log writer threads provides better performance.

- Optimize the use of spin delay by user threads waiting for flushed redo. Spin delay helps reduce latency. During periods of low concurrency, reducing latency may be less of a priority, and avoiding the use of spin delay during these periods may reduce energy consumption. During periods of high concurrency, you may want to avoid expending processing power on spin delay so that it can be used for other work. The following system variables permit setting high and low watermark values that define boundaries for the use of spin delay.

  - `innodb_log_wait_for_flush_spin_hwm`: Defines the maximum average log flush time beyond which user threads no longer spin while waiting for flushed redo. The default value is 400 microseconds.

  - `innodb_log_spin_cpu_abs_lwm`: Defines the minimum amount of CPU usage below which user threads no longer spin while waiting for flushed redo. The value is expressed as a sum of CPU core usage. For example, The default value of 80 is 80% of a single CPU core. On a system with a multi-core processor, a value of 150 represents 100% usage of one CPU core plus 50% usage of a second CPU core.

  - `innodb_log_spin_cpu_pct_hwm`: Defines the maximum amount of CPU usage above which user threads no longer spin while waiting for flushed redo. The value is expressed as a percentage of the combined total processing power of all CPU cores. The default value is 50%. For example, 100% usage of two CPU cores is 50% of the combined CPU processing power on a server with four CPU cores.

    The `innodb_log_spin_cpu_pct_hwm` configuration option respects processor affinity. For example, if a server has 48 cores but the `mysqld` process is pinned to only four CPU cores, the other 44 CPU cores are ignored.

8.5.5 Bulk Data Loading for InnoDB Tables

These performance tips supplement the general guidelines for fast inserts in Section 8.2.5.1, “Optimizing INSERT Statements”.

- When importing data into `InnoDB`, turn off autocommit mode, because it performs a log flush to disk for every insert. To disable autocommit during your import operation, surround it with `SET autocommit` and `COMMIT` statements:

```
SET autocommit=0;
... SQL import statements ...
COMMIT;
```

  The `mysqldump` option `--opt` creates dump files that are fast to import into an `InnoDB` table, even without wrapping them with the `SET autocommit` and `COMMIT` statements.

- If you have `UNIQUE` constraints on secondary keys, you can speed up table imports by temporarily turning off the uniqueness checks during the import session:

```
SET unique_checks=0;
... SQL import statements ...
SET unique_checks=1;
```

For big tables, this saves a lot of disk I/O because `InnoDB` can use its change buffer to write secondary index records in a batch. Be certain that the data contains no duplicate keys.

- If you have `FOREIGN KEY` constraints in your tables, you can speed up table imports by turning off the foreign key checks for the duration of the import session:

```
SET foreign_key_checks=0;
... SQL import statements ...
SET foreign_key_checks=1;
```

For big tables, this can save a lot of disk I/O.

- Use the multiple-row `INSERT` syntax to reduce communication overhead between the client and the server if you need to insert many rows:

```
INSERT INTO yourtable VALUES (1,2), (5,5), ...;
```

This tip is valid for inserts into any table, not just `InnoDB` tables.

- When doing bulk inserts into tables with auto-increment columns, set `innodb_autoinc_lock_mode` to 2 (interleaved) instead of 1 (consecutive). See Section 15.6.1.6, "AUTO_INCREMENT Handling in InnoDB" for details.

- When performing bulk inserts, it is faster to insert rows in `PRIMARY KEY` order. `InnoDB` tables use a clustered index, which makes it relatively fast to use data in the order of the `PRIMARY KEY`. Performing bulk inserts in `PRIMARY KEY` order is particularly important for tables that do not fit entirely within the buffer pool.

- For optimal performance when loading data into an `InnoDB` `FULLTEXT` index, follow this set of steps:

  1. Define a column `FTS_DOC_ID` at table creation time, of type `BIGINT UNSIGNED NOT NULL`, with a unique index named `FTS_DOC_ID_INDEX`. For example:

     ```
     CREATE TABLE t1 (
     FTS_DOC_ID BIGINT unsigned NOT NULL AUTO_INCREMENT,
     title varchar(255) NOT NULL DEFAULT '',
     text mediumtext NOT NULL,
     PRIMARY KEY (`FTS_DOC_ID`)
     ) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;
     CREATE UNIQUE INDEX FTS_DOC_ID_INDEX on t1(FTS_DOC_ID);
     ```

  2. Load the data into the table.

  3. Create the `FULLTEXT` index after the data is loaded.

Note

When adding `FTS_DOC_ID` column at table creation time, ensure that the `FTS_DOC_ID` column is updated when the `FULLTEXT` indexed column is updated, as the `FTS_DOC_ID` must increase monotonically with each `INSERT` or `UPDATE`. If you choose not to add the `FTS_DOC_ID` at table creation time and have `InnoDB` manage DOC IDs for you, `InnoDB` adds the `FTS_DOC_ID` as a hidden column with the next `CREATE FULLTEXT INDEX` call. This approach, however, requires a table rebuild which can impact performance.

- If loading data into a *new* MySQL instance, consider disabling redo logging using `ALTER INSTANCE {ENABLE|DISABLE} INNODB REDO_LOG` syntax. Disabling redo logging helps speed up data loading by avoiding redo log writes. For more information, see Disabling Redo Logging.

Warning

This feature is intended only for loading data into a new MySQL instance. *Do not disable redo logging on a production system.* It is permitted to shutdown and restart the server while redo logging is disabled, but an unexpected server stoppage while redo logging is disabled can cause data loss and instance corruption.

- Use MySQL Shell to import data. MySQL Shell's parallel table import utility `util.importTable()` provides rapid data import to a MySQL relational table for large data files. MySQL Shell's dump loading utility `util.loadDump()` also offers parallel load capabilities. See MySQL Shell Utilities.

8.5.6 Optimizing InnoDB Queries

To tune queries for `InnoDB` tables, create an appropriate set of indexes on each table. See Section 8.3.1, “How MySQL Uses Indexes” for details. Follow these guidelines for `InnoDB` indexes:

- Because each `InnoDB` table has a primary key (whether you request one or not), specify a set of primary key columns for each table, columns that are used in the most important and time-critical queries.
- Do not specify too many or too long columns in the primary key, because these column values are duplicated in each secondary index. When an index contains unnecessary data, the I/O to read this data and memory to cache it reduce the performance and scalability of the server.
- Do not create a separate secondary index for each column, because each query can only make use of one index. Indexes on rarely tested columns or columns with only a few different values might not be helpful for any queries. If you have many queries for the same table, testing different combinations of columns, try to create a small number of concatenated indexes rather than a large number of single-column indexes. If an index contains all the columns needed for the result set (known as a covering index), the query might be able to avoid reading the table data at all.
- If an indexed column cannot contain any `NULL` values, declare it as `NOT NULL` when you create the table. The optimizer can better determine which index is most effective to use for a query, when it knows whether each column contains `NULL` values.
- You can optimize single-query transactions for `InnoDB` tables, using the technique in Section 8.5.3, “Optimizing InnoDB Read-Only Transactions”.

8.5.7 Optimizing InnoDB DDL Operations

- Many DDL operations on tables and indexes (`CREATE`, `ALTER`, and `DROP` statements) can be performed online. See Section 15.12, “InnoDB and Online DDL” for details.
- Online DDL support for adding secondary indexes means that you can generally speed up the process of creating and loading a table and associated indexes by creating the table without secondary indexes, then adding secondary indexes after the data is loaded.
- Use `TRUNCATE TABLE` to empty a table, not `DELETE FROM` *`tbl_name`*. Foreign key constraints can make a `TRUNCATE` statement work like a regular `DELETE` statement, in which case a sequence of commands like `DROP TABLE` and `CREATE TABLE` might be fastest.
- Because the primary key is integral to the storage layout of each `InnoDB` table, and changing the definition of the primary key involves reorganizing the whole table, always set up the primary key as part of the `CREATE TABLE` statement, and plan ahead so that you do not need to `ALTER` or `DROP` the primary key afterward.

8.5.8 Optimizing InnoDB Disk I/O

If you follow best practices for database design and tuning techniques for SQL operations, but your database is still slow due to heavy disk I/O activity, consider these disk I/O optimizations. If the Unix

`top` tool or the Windows Task Manager shows that the CPU usage percentage with your workload is less than 70%, your workload is probably disk-bound.

- Increase buffer pool size

  When table data is cached in the `InnoDB` buffer pool, it can be accessed repeatedly by queries without requiring any disk I/O. Specify the size of the buffer pool with the `innodb_buffer_pool_size` option. This memory area is important enough that it is typically recommended that `innodb_buffer_pool_size` is configured to 50 to 75 percent of system memory. For more information see, Section 8.12.3.1, "How MySQL Uses Memory".

- Adjust the flush method

  In some versions of GNU/Linux and Unix, flushing files to disk with the Unix `fsync()` call (which `InnoDB` uses by default) and similar methods is surprisingly slow. If database write performance is an issue, conduct benchmarks with the `innodb_flush_method` parameter set to `O_DSYNC`.

- Configure a threshold for operating system flushes

  By default, when `InnoDB` creates a new data file, such as a new log file or tablespace file, the file is fully written to the operating system cache before it is flushed to disk, which can cause a large amount of disk write activity to occur at once. To force smaller, periodic flushes of data from the operating system cache, you can use the `innodb_fsync_threshold` variable to define a threshold value, in bytes. When the byte threshold is reached, the contents of the operating system cache are flushed to disk. The default value of 0 forces the default behavior, which is to flush data to disk only after a file is fully written to the cache.

  Specifying a threshold to force smaller, periodic flushes may be beneficial in cases where multiple MySQL instances use the same storage devices. For example, creating a new MySQL instance and its associated data files could cause large surges of disk write activity, impeding the performance of other MySQL instances that use the same storage devices. Configuring a threshold helps avoid such surges in write activity.

- Use fdatasync() instead of fsync()

  On platforms that support `fdatasync()` system calls, the `innodb_use_fdatasync` variable, introduced in MySQL 8.0.26, permits using `fdatasync()` instead of `fsync()` for operating system flushes. An `fdatasync()` system call does not flush changes to file metadata unless required for subsequent data retrieval, providing a potential performance benefit.

  A subset of `innodb_flush_method` settings such as `fsync`, `O_DSYNC`, and `O_DIRECT` use `fsync()` system calls. The `innodb_use_fdatasync` variable is applicable when using those settings.

- Use a noop or deadline I/O scheduler with native AIO on Linux

  `InnoDB` uses the asynchronous I/O subsystem (native AIO) on Linux to perform read-ahead and write requests for data file pages. This behavior is controlled by the `innodb_use_native_aio` configuration option, which is enabled by default. With native AIO, the type of I/O scheduler has greater influence on I/O performance. Generally, noop and deadline I/O schedulers are recommended. Conduct benchmarks to determine which I/O scheduler provides the best results for your workload and environment. For more information, see Section 15.8.6, "Using Asynchronous I/O on Linux".

- Use direct I/O on Solaris 10 for x86_64 architecture

  When using the `InnoDB` storage engine on Solaris 10 for x86_64 architecture (AMD Opteron), use direct I/O for `InnoDB`-related files to avoid degradation of `InnoDB` performance. To use direct I/O for an entire UFS file system used for storing `InnoDB`-related files, mount it with the `forcedirectio` option; see `mount_ufs(1M)`. (The default on Solaris 10/x86_64 is *not* to use this option.) To apply direct I/O only to `InnoDB` file operations rather than the whole file system, set

`innodb_flush_method = O_DIRECT`. With this setting, `InnoDB` calls `directio()` instead of `fcntl()` for I/O to data files (not for I/O to log files).

- Use raw storage for data and log files with Solaris 2.6 or later

  When using the `InnoDB` storage engine with a large `innodb_buffer_pool_size` value on any release of Solaris 2.6 and up and any platform (sparc/x86/x64/amd64), conduct benchmarks with `InnoDB` data files and log files on raw devices or on a separate direct I/O UFS file system, using the `forcedirectio` mount option as described previously. (It is necessary to use the mount option rather than setting `innodb_flush_method` if you want direct I/O for the log files.) Users of the Veritas file system VxFS should use the `convosync=direct` mount option.

  Do not place other MySQL data files, such as those for `MyISAM` tables, on a direct I/O file system. Executables or libraries *must not* be placed on a direct I/O file system.

- Use additional storage devices

  Additional storage devices could be used to set up a RAID configuration. For related information, see Section 8.12.1, “Optimizing Disk I/O”.

  Alternatively, `InnoDB` tablespace data files and log files can be placed on different physical disks. For more information, refer to the following sections:

  - Section 15.8.1, “InnoDB Startup Configuration”
  - Section 15.6.1.2, “Creating Tables Externally”
  - Creating a General Tablespace
  - Section 15.6.1.4, “Moving or Copying InnoDB Tables”

- Consider non-rotational storage

  Non-rotational storage generally provides better performance for random I/O operations; and rotational storage for sequential I/O operations. When distributing data and log files across rotational and non-rotational storage devices, consider the type of I/O operations that are predominantly performed on each file.

  Random I/O-oriented files typically include file-per-table and general tablespace data files, undo tablespace files, and temporary tablespace files. Sequential I/O-oriented files include `InnoDB`

system tablespace files (due to doublewrite buffering prior to MySQL 8.0.20 and change buffering), doublewrite files introduced in MySQL 8.0.20, and log files such as binary log files and redo log files.

Review settings for the following configuration options when using non-rotational storage:

- `innodb_checksum_algorithm`

  The `crc32` option uses a faster checksum algorithm and is recommended for fast storage systems.

- `innodb_flush_neighbors`

  Optimizes I/O for rotational storage devices. Disable it for non-rotational storage or a mix of rotational and non-rotational storage. It is disabled by default.

- `innodb_idle_flush_pct`

  Permits placing a limit on page flushing during idle periods, which can help extend the life of non-rotational storage devices. Introduced in MySQL 8.0.18.

- `innodb_io_capacity`

  The default setting of 200 is generally sufficient for a lower-end non-rotational storage device. For higher-end, bus-attached devices, consider a higher setting such as 1000.

- `innodb_io_capacity_max`

  The default value of 2000 is intended for workloads that use non-rotational storage. For a high-end, bus-attached non-rotational storage device, consider a higher setting such as 2500.

- `innodb_log_compressed_pages`

  If redo logs are on non-rotational storage, consider disabling this option to reduce logging. See Disable logging of compressed pages.

- `innodb_log_file_size` (deprecated in MySQL 8.0.30)

  If redo logs are on non-rotational storage, configure this option to maximize caching and write combining.

- `innodb_redo_log_capacity`

  If redo logs are on non-rotational storage, configure this option to maximize caching and write combining.

- `innodb_page_size`

  Consider using a page size that matches the internal sector size of the disk. Early-generation SSD devices often have a 4KB sector size. Some newer devices have a 16KB sector size. The default `InnoDB` page size is 16KB. Keeping the page size close to the storage device block size minimizes the amount of unchanged data that is rewritten to disk.

- `binlog_row_image`

  If binary logs are on non-rotational storage and all tables have primary keys, consider setting this option to `minimal` to reduce logging.

Ensure that TRIM support is enabled for your operating system. It is typically enabled by default.

- Increase I/O capacity to avoid backlogs

  If throughput drops periodically because of `InnoDB` checkpoint operations, consider increasing the value of the `innodb_io_capacity` configuration option. Higher values cause more frequent flushing, avoiding the backlog of work that can cause dips in throughput.

- Lower I/O capacity if flushing does not fall behind

  If the system is not falling behind with `InnoDB` flushing operations, consider lowering the value of the `innodb_io_capacity` configuration option. Typically, you keep this option value as low as practical, but not so low that it causes periodic drops in throughput as mentioned in the preceding bullet. In a typical scenario where you could lower the option value, you might see a combination like this in the output from `SHOW ENGINE INNODB STATUS`:

  - History list length low, below a few thousand.
  - Insert buffer merges close to rows inserted.
  - Modified pages in buffer pool consistently well below `innodb_max_dirty_pages_pct` of the buffer pool. (Measure at a time when the server is not doing bulk inserts; it is normal during bulk inserts for the modified pages percentage to rise significantly.)
  - `Log sequence number - Last checkpoint` is at less than 7/8 or ideally less than 6/8 of the total size of the `InnoDB` log files.

- Store system tablespace files on Fusion-io devices

  You can take advantage of a doublewrite buffer-related I/O optimization by storing the files that contain the doublewrite storage area on Fusion-io devices that support atomic writes. (Prior to MySQL 8.0.20, the doublewrite buffer storage are resides in the system tablespace data files. As of MySQL 8.0.20, the storage area resides in doublewrite files. See Section 15.6.4, “Doublewrite Buffer”.) When doublewrite storage area files are placed on Fusion-io devices that support atomic writes, the doublewrite buffer is automatically disabled and Fusion-io atomic writes are used for all data files. This feature is only supported on Fusion-io hardware and is only enabled for Fusion-io NVMFS on Linux. To take full advantage of this feature, an `innodb_flush_method` setting of `O_DIRECT` is recommended.

  

  Note

  Because the doublewrite buffer setting is global, the doublewrite buffer is also disabled for data files that do not reside on Fusion-io hardware.

- Disable logging of compressed pages

  When using the `InnoDB` table compression feature, images of re-compressed pages are written to the redo log when changes are made to compressed data. This behavior is controlled by `innodb_log_compressed_pages`, which is enabled by default to prevent corruption that can occur if a different version of the `zlib` compression algorithm is used during recovery. If you are certain that the `zlib` version is not subject to change, disable `innodb_log_compressed_pages` to reduce redo log generation for workloads that modify compressed data.

8.5.9 Optimizing InnoDB Configuration Variables

Different settings work best for servers with light, predictable loads, versus servers that are running near full capacity all the time, or that experience spikes of high activity.

Because the `InnoDB` storage engine performs many of its optimizations automatically, many performance-tuning tasks involve monitoring to ensure that the database is performing well, and changing configuration options when performance drops. See Section 15.16, “InnoDB Integration with MySQL Performance Schema” for information about detailed `InnoDB` performance monitoring.

The main configuration steps you can perform include:

- Controlling the types of data change operations for which `InnoDB` buffers the changed data, to avoid frequent small disk writes. See Configuring Change Buffering. Because the default is to buffer all types of data change operations, only change this setting if you need to reduce the amount of buffering.
- Turning the adaptive hash indexing feature on and off using the `innodb_adaptive_hash_index` option. See Section 15.5.3, “Adaptive Hash Index” for more information. You might change this setting during periods of unusual activity, then restore it to its original setting.
- Setting a limit on the number of concurrent threads that `InnoDB` processes, if context switching is a bottleneck. See Section 15.8.4, “Configuring Thread Concurrency for InnoDB”.
- Controlling the amount of prefetching that `InnoDB` does with its read-ahead operations. When the system has unused I/O capacity, more read-ahead can improve the performance of queries. Too much read-ahead can cause periodic drops in performance on a heavily loaded system. See Section 15.8.3.4, “Configuring InnoDB Buffer Pool Prefetching (Read-Ahead)”.
- Increasing the number of background threads for read or write operations, if you have a high-end I/O subsystem that is not fully utilized by the default values. See Section 15.8.5, “Configuring the Number of Background InnoDB I/O Threads”.
- Controlling how much I/O `InnoDB` performs in the background. See Section 15.8.7, “Configuring InnoDB I/O Capacity”. You might scale back this setting if you observe periodic drops in performance.
- Controlling the algorithm that determines when `InnoDB` performs certain types of background writes. See Section 15.8.3.5, “Configuring Buffer Pool Flushing”. The algorithm works for some types of workloads but not others, so you might disable this feature if you observe periodic drops in performance.
- Taking advantage of multicore processors and their cache memory configuration, to minimize delays in context switching. See Section 15.8.8, “Configuring Spin Lock Polling”.
- Preventing one-time operations such as table scans from interfering with the frequently accessed data stored in the `InnoDB` buffer cache. See Section 15.8.3.3, “Making the Buffer Pool Scan Resistant”.
- Adjusting log files to a size that makes sense for reliability and crash recovery. `InnoDB` log files have often been kept small to avoid long startup times after a crash. Optimizations introduced in MySQL 5.5 speed up certain steps of the crash recovery process. In particular, scanning the redo log and applying the redo log are faster due to improved algorithms for memory management. If you have kept your log files artificially small to avoid long startup times, you can now consider increasing log file size to reduce the I/O that occurs due recycling of redo log records.
- Configuring the size and number of instances for the `InnoDB` buffer pool, especially important for systems with multi-gigabyte buffer pools. See Section 15.8.3.2, “Configuring Multiple Buffer Pool Instances”.
- Increasing the maximum number of concurrent transactions, which dramatically improves scalability for the busiest databases. See Section 15.6.6, “Undo Logs”.
- Moving purge operations (a type of garbage collection) into a background thread. See Section 15.8.9, “Purge Configuration”. To effectively measure the results of this setting, tune the other I/O-related and thread-related configuration settings first.
- Reducing the amount of switching that `InnoDB` does between concurrent threads, so that SQL operations on a busy server do not queue up and form a “traffic jam”. Set a value for the `innodb_thread_concurrency` option, up to approximately 32 for a high-powered modern system. Increase the value for the `innodb_concurrency_tickets` option, typically to 5000 or so. This combination of options sets a cap on the number of threads that `InnoDB` processes at any one time,

and allows each thread to do substantial work before being swapped out, so that the number of waiting threads stays low and operations can complete without excessive context switching.

8.5.10 Optimizing InnoDB for Systems with Many Tables

- If you have configured non-persistent optimizer statistics (a non-default configuration), `InnoDB` computes index cardinality values for a table the first time that table is accessed after startup, instead of storing such values in the table. This step can take significant time on systems that partition the data into many tables. Since this overhead only applies to the initial table open operation, to "warm up" a table for later use, access it immediately after startup by issuing a statement such as `SELECT 1 FROM tbl_name LIMIT 1`.

  Optimizer statistics are persisted to disk by default, enabled by the `innodb_stats_persistent` configuration option. For information about persistent optimizer statistics, see Section 15.8.10.1, "Configuring Persistent Optimizer Statistics Parameters".

8.6 Optimizing for MyISAM Tables

The `MyISAM` storage engine performs best with read-mostly data or with low-concurrency operations, because table locks limit the ability to perform simultaneous updates. In MySQL, `InnoDB` is the default storage engine rather than `MyISAM`.

8.6.1 Optimizing MyISAM Queries

Some general tips for speeding up queries on `MyISAM` tables:

- To help MySQL better optimize queries, use `ANALYZE TABLE` or run `myisamchk --analyze` on a table after it has been loaded with data. This updates a value for each index part that indicates the average number of rows that have the same value. (For unique indexes, this is always 1.) MySQL uses this to decide which index to choose when you join two tables based on a nonconstant expression. You can check the result from the table analysis by using `SHOW INDEX FROM tbl_name` and examining the `Cardinality` value. `myisamchk --description --verbose` shows index distribution information.

- To sort an index and data according to an index, use `myisamchk --sort-index --sort-records=1` (assuming that you want to sort on index 1). This is a good way to make queries faster if you have a unique index from which you want to read all rows in order according to the index. The first time you sort a large table this way, it may take a long time.

- Try to avoid complex `SELECT` queries on `MyISAM` tables that are updated frequently, to avoid problems with table locking that occur due to contention between readers and writers.

- `MyISAM` supports concurrent inserts: If a table has no free blocks in the middle of the data file, you can `INSERT` new rows into it at the same time that other threads are reading from the table. If it is important to be able to do this, consider using the table in ways that avoid deleting rows. Another possibility is to run `OPTIMIZE TABLE` to defragment the table after you have deleted a lot of rows from it. This behavior is altered by setting the `concurrent_insert` variable. You can force new rows to be appended (and therefore permit concurrent inserts), even in tables that have deleted rows. See Section 8.11.3, "Concurrent Inserts".

- For `MyISAM` tables that change frequently, try to avoid all variable-length columns (`VARCHAR`, `BLOB`, and `TEXT`). The table uses dynamic row format if it includes even a single variable-length column. See Chapter 16, *Alternative Storage Engines*.

- It is normally not useful to split a table into different tables just because the rows become large. In accessing a row, the biggest performance hit is the disk seek needed to find the first byte of the row. After finding the data, most modern disks can read the entire row fast enough for most applications. The only cases where splitting up a table makes an appreciable difference is if it is a `MyISAM` table using dynamic row format that you can change to a fixed row size, or if you very often need to scan the table but do not need most of the columns. See Chapter 16, *Alternative Storage Engines*.

- Use `ALTER TABLE ... ORDER BY expr1, expr2, ...` if you usually retrieve rows in *`expr1, expr2, ...`* order. By using this option after extensive changes to the table, you may be able to get higher performance.
- If you often need to calculate results such as counts based on information from a lot of rows, it may be preferable to introduce a new table and update the counter in real time. An update of the following form is very fast:

  ```
  UPDATE tbl_name SET count_col=count_col+1 WHERE key_col=constant;
  ```

  This is very important when you use MySQL storage engines such as `MyISAM` that has only table-level locking (multiple readers with single writers). This also gives better performance with most database systems, because the row locking manager in this case has less to do.
- Use `OPTIMIZE TABLE` periodically to avoid fragmentation with dynamic-format `MyISAM` tables. See Section 16.2.3, “MyISAM Table Storage Formats”.
- Declaring a `MyISAM` table with the `DELAY_KEY_WRITE=1` table option makes index updates faster because they are not flushed to disk until the table is closed. The downside is that if something kills the server while such a table is open, you must ensure that the table is okay by running the server with the `myisam_recover_options` system variable set, or by running `myisamchk` before restarting the server. (However, even in this case, you should not lose anything by using `DELAY_KEY_WRITE`, because the key information can always be generated from the data rows.)
- Strings are automatically prefix- and end-space compressed in `MyISAM` indexes. See Section 13.1.15, “CREATE INDEX Statement”.
- You can increase performance by caching queries or answers in your application and then executing many inserts or updates together. Locking the table during this operation ensures that the index cache is only flushed once after all updates.

8.6.2 Bulk Data Loading for MyISAM Tables

These performance tips supplement the general guidelines for fast inserts in Section 8.2.5.1, “Optimizing INSERT Statements”.

- For a `MyISAM` table, you can use concurrent inserts to add rows at the same time that `SELECT` statements are running, if there are no deleted rows in middle of the data file. See Section 8.11.3, “Concurrent Inserts”.
- With some extra work, it is possible to make `LOAD DATA` run even faster for a `MyISAM` table when the table has many indexes. Use the following procedure:

  1. Execute a `FLUSH TABLES` statement or a `mysqladmin flush-tables` command.
  2. Use `myisamchk --keys-used=0 -rq /path/to/db/tbl_name` to remove all use of indexes for the table.
  3. Insert data into the table with `LOAD DATA`. This does not update any indexes and therefore is very fast.
  4. If you intend only to read from the table in the future, use `myisampack` to compress it. See Section 16.2.3.3, “Compressed Table Characteristics”.
  5. Re-create the indexes with `myisamchk -rq /path/to/db/tbl_name`. This creates the index tree in memory before writing it to disk, which is much faster than updating the index during `LOAD DATA` because it avoids lots of disk seeks. The resulting index tree is also perfectly balanced.
  6. Execute a `FLUSH TABLES` statement or a `mysqladmin flush-tables` command.

  `LOAD DATA` performs the preceding optimization automatically if the `MyISAM` table into which you insert data is empty. The main difference between automatic optimization and using the procedure

explicitly is that you can let `myisamchk` allocate much more temporary memory for the index creation than you might want the server to allocate for index re-creation when it executes the `LOAD DATA` statement.

You can also disable or enable the nonunique indexes for a `MyISAM` table by using the following statements rather than `myisamchk`. If you use these statements, you can skip the `FLUSH TABLES` operations:

```
ALTER TABLE tbl_name DISABLE KEYS;
ALTER TABLE tbl_name ENABLE KEYS;
```

- To speed up `INSERT` operations that are performed with multiple statements for nontransactional tables, lock your tables:

```
LOCK TABLES a WRITE;
INSERT INTO a VALUES (1,23),(2,34),(4,33);
INSERT INTO a VALUES (8,26),(6,29);
...
UNLOCK TABLES;
```

This benefits performance because the index buffer is flushed to disk only once, after all `INSERT` statements have completed. Normally, there would be as many index buffer flushes as there are `INSERT` statements. Explicit locking statements are not needed if you can insert all rows with a single `INSERT`.

Locking also lowers the total time for multiple-connection tests, although the maximum wait time for individual connections might go up because they wait for locks. Suppose that five clients attempt to perform inserts simultaneously as follows:

- Connection 1 does 1000 inserts
- Connections 2, 3, and 4 do 1 insert
- Connection 5 does 1000 inserts

If you do not use locking, connections 2, 3, and 4 finish before 1 and 5. If you use locking, connections 2, 3, and 4 probably do not finish before 1 or 5, but the total time should be about 40% faster.

`INSERT`, `UPDATE`, and `DELETE` operations are very fast in MySQL, but you can obtain better overall performance by adding locks around everything that does more than about five successive inserts or updates. If you do very many successive inserts, you could do a `LOCK TABLES` followed by an `UNLOCK TABLES` once in a while (each 1,000 rows or so) to permit other threads to access table. This would still result in a nice performance gain.

`INSERT` is still much slower for loading data than `LOAD DATA`, even when using the strategies just outlined.

- To increase performance for `MyISAM` tables, for both `LOAD DATA` and `INSERT`, enlarge the key cache by increasing the `key_buffer_size` system variable. See Section 5.1.1, “Configuring the Server”.

8.6.3 Optimizing REPAIR TABLE Statements

`REPAIR TABLE` for `MyISAM` tables is similar to using `myisamchk` for repair operations, and some of the same performance optimizations apply:

- `myisamchk` has variables that control memory allocation. You may be able to its improve performance by setting these variables, as described in Section 4.6.4.6, “myisamchk Memory Usage”.

- For `REPAIR TABLE`, the same principle applies, but because the repair is done by the server, you set server system variables instead of `myisamchk` variables. Also, in addition to setting memory-allocation variables, increasing the `myisam_max_sort_file_size` system variable increases the likelihood that the repair uses the faster filesort method and avoids the slower repair by key cache method. Set the variable to the maximum file size for your system, after checking to be sure that there is enough free space to hold a copy of the table files. The free space must be available in the file system containing the original table files.

Suppose that a `myisamchk` table-repair operation is done using the following options to set its memory-allocation variables:

```
--key_buffer_size=128M --myisam_sort_buffer_size=256M
--read_buffer_size=64M --write_buffer_size=64M
```

Some of those `myisamchk` variables correspond to server system variables:

`myisamchk` Variable	**System Variable**
`key_buffer_size`	`key_buffer_size`
`myisam_sort_buffer_size`	`myisam_sort_buffer_size`
`read_buffer_size`	`read_buffer_size`
`write_buffer_size`	none

Each of the server system variables can be set at runtime, and some of them (`myisam_sort_buffer_size`, `read_buffer_size`) have a session value in addition to a global value. Setting a session value limits the effect of the change to your current session and does not affect other users. Changing a global-only variable (`key_buffer_size`, `myisam_max_sort_file_size`) affects other users as well. For `key_buffer_size`, you must take into account that the buffer is shared with those users. For example, if you set the `myisamchk key_buffer_size` variable to 128MB, you could set the corresponding `key_buffer_size` system variable larger than that (if it is not already set larger), to permit key buffer use by activity in other sessions. However, changing the global key buffer size invalidates the buffer, causing increased disk I/O and slowdown for other sessions. An alternative that avoids this problem is to use a separate key cache, assign to it the indexes from the table to be repaired, and deallocate it when the repair is complete. See Section 8.10.2.2, “Multiple Key Caches”.

Based on the preceding remarks, a `REPAIR TABLE` operation can be done as follows to use settings similar to the `myisamchk` command. Here a separate 128MB key buffer is allocated and the file system is assumed to permit a file size of at least 100GB.

```
SET SESSION myisam_sort_buffer_size = 256*1024*1024;
SET SESSION read_buffer_size = 64*1024*1024;
SET GLOBAL myisam_max_sort_file_size = 100*1024*1024*1024;
SET GLOBAL repair_cache.key_buffer_size = 128*1024*1024;
CACHE INDEX tbl_name IN repair_cache;
LOAD INDEX INTO CACHE tbl_name;
REPAIR TABLE tbl_name ;
SET GLOBAL repair_cache.key_buffer_size = 0;
```

If you intend to change a global variable but want to do so only for the duration of a `REPAIR TABLE` operation to minimally affect other users, save its value in a user variable and restore it afterward. For example:

```
SET @old_myisam_sort_buffer_size = @@GLOBAL.myisam_max_sort_file_size;
SET GLOBAL myisam_max_sort_file_size = 100*1024*1024*1024;
REPAIR TABLE tbl_name ;
SET GLOBAL myisam_max_sort_file_size = @old_myisam_max_sort_file_size;
```

The system variables that affect `REPAIR TABLE` can be set globally at server startup if you want the values to be in effect by default. For example, add these lines to the server `my.cnf` file:

```
[mysqld]
myisam_sort_buffer_size=256M
key_buffer_size=1G
```

```
myisam_max_sort_file_size=100G
```

These settings do not include `read_buffer_size`. Setting `read_buffer_size` globally to a large value does so for all sessions and can cause performance to suffer due to excessive memory allocation for a server with many simultaneous sessions.

8.7 Optimizing for MEMORY Tables

Consider using `MEMORY` tables for noncritical data that is accessed often, and is read-only or rarely updated. Benchmark your application against equivalent `InnoDB` or `MyISAM` tables under a realistic workload, to confirm that any additional performance is worth the risk of losing data, or the overhead of copying data from a disk-based table at application start.

For best performance with `MEMORY` tables, examine the kinds of queries against each table, and specify the type to use for each associated index, either a B-tree index or a hash index. On the `CREATE INDEX` statement, use the clause `USING BTREE` or `USING HASH`. B-tree indexes are fast for queries that do greater-than or less-than comparisons through operators such as `>` or `BETWEEN`. Hash indexes are only fast for queries that look up single values through the `=` operator, or a restricted set of values through the `IN` operator. For why `USING BTREE` is often a better choice than the default `USING HASH`, see Section 8.2.1.23, “Avoiding Full Table Scans”. For implementation details of the different types of `MEMORY` indexes, see Section 8.3.9, “Comparison of B-Tree and Hash Indexes”.

8.8 Understanding the Query Execution Plan

Depending on the details of your tables, columns, indexes, and the conditions in your `WHERE` clause, the MySQL optimizer considers many techniques to efficiently perform the lookups involved in an SQL query. A query on a huge table can be performed without reading all the rows; a join involving several tables can be performed without comparing every combination of rows. The set of operations that the optimizer chooses to perform the most efficient query is called the “query execution plan”, also known as the `EXPLAIN` plan. Your goals are to recognize the aspects of the `EXPLAIN` plan that indicate a query is optimized well, and to learn the SQL syntax and indexing techniques to improve the plan if you see some inefficient operations.

8.8.1 Optimizing Queries with EXPLAIN

The `EXPLAIN` statement provides information about how MySQL executes statements:

- `EXPLAIN` works with `SELECT`, `DELETE`, `INSERT`, `REPLACE`, and `UPDATE` statements.
- When `EXPLAIN` is used with an explainable statement, MySQL displays information from the optimizer about the statement execution plan. That is, MySQL explains how it would process the statement, including information about how tables are joined and in which order. For information about using `EXPLAIN` to obtain execution plan information, see Section 8.8.2, “EXPLAIN Output Format”.
- When `EXPLAIN` is used with `FOR CONNECTION` *`connection_id`* rather than an explainable statement, it displays the execution plan for the statement executing in the named connection. See Section 8.8.4, “Obtaining Execution Plan Information for a Named Connection”.
- For `SELECT` statements, `EXPLAIN` produces additional execution plan information that can be displayed using `SHOW WARNINGS`. See Section 8.8.3, “Extended EXPLAIN Output Format”.
- `EXPLAIN` is useful for examining queries involving partitioned tables. See Section 24.3.5, “Obtaining Information About Partitions”.
- The `FORMAT` option can be used to select the output format. `TRADITIONAL` presents the output in tabular format. This is the default if no `FORMAT` option is present. `JSON` format displays the information in JSON format.

With the help of `EXPLAIN`, you can see where you should add indexes to tables so that the statement executes faster by using indexes to find rows. You can also use `EXPLAIN` to check whether the

optimizer joins the tables in an optimal order. To give a hint to the optimizer to use a join order corresponding to the order in which the tables are named in a `SELECT` statement, begin the statement with `SELECT STRAIGHT_JOIN` rather than just `SELECT`. (See Section 13.2.13, “SELECT Statement”.) However, `STRAIGHT_JOIN` may prevent indexes from being used because it disables semijoin transformations. See Section 8.2.2.1, “Optimizing IN and EXISTS Subquery Predicates with Semijoin Transformations”.

The optimizer trace may sometimes provide information complementary to that of `EXPLAIN`. However, the optimizer trace format and content are subject to change between versions. For details, see MySQL Internals: Tracing the Optimizer.

If you have a problem with indexes not being used when you believe that they should be, run `ANALYZE TABLE` to update table statistics, such as cardinality of keys, that can affect the choices the optimizer makes. See Section 13.7.3.1, “ANALYZE TABLE Statement”.

Note

`EXPLAIN` can also be used to obtain information about the columns in a table. `EXPLAIN *tbl_name*` is synonymous with `DESCRIBE *tbl_name*` and `SHOW COLUMNS FROM *tbl_name*`. For more information, see Section 13.8.1, “DESCRIBE Statement”, and Section 13.7.7.5, “SHOW COLUMNS Statement”.

8.8.2 EXPLAIN Output Format

The `EXPLAIN` statement provides information about how MySQL executes statements. `EXPLAIN` works with `SELECT`, `DELETE`, `INSERT`, `REPLACE`, and `UPDATE` statements.

`EXPLAIN` returns a row of information for each table used in the `SELECT` statement. It lists the tables in the output in the order that MySQL would read them while processing the statement. This means that MySQL reads a row from the first table, then finds a matching row in the second table, and then in the third table, and so on. When all tables are processed, MySQL outputs the selected columns and backtracks through the table list until a table is found for which there are more matching rows. The next row is read from this table and the process continues with the next table.

Note

MySQL Workbench has a Visual Explain capability that provides a visual representation of `EXPLAIN` output. See Tutorial: Using Explain to Improve Query Performance.

- EXPLAIN Output Columns
- EXPLAIN Join Types
- EXPLAIN Extra Information
- EXPLAIN Output Interpretation

EXPLAIN Output Columns

This section describes the output columns produced by `EXPLAIN`. Later sections provide additional information about the `type` and `Extra` columns.

Each output row from `EXPLAIN` provides information about one table. Each row contains the values summarized in Table 8.1, “EXPLAIN Output Columns”, and described in more detail following the table. Column names are shown in the table's first column; the second column provides the equivalent property name shown in the output when `FORMAT=JSON` is used.

Table 8.1 EXPLAIN Output Columns

Column	JSON Name	Meaning
`id`	`select_id`	The `SELECT` identifier

Column	JSON Name	Meaning
`select_type`	None	The `SELECT` type
`table`	`table_name`	The table for the output row
`partitions`	`partitions`	The matching partitions
`type`	`access_type`	The join type
`possible_keys`	`possible_keys`	The possible indexes to choose
`key`	`key`	The index actually chosen
`key_len`	`key_length`	The length of the chosen key
`ref`	`ref`	The columns compared to the index
`rows`	`rows`	Estimate of rows to be examined
`filtered`	`filtered`	Percentage of rows filtered by table condition
`Extra`	None	Additional information

Note

JSON properties which are `NULL` are not displayed in JSON-formatted `EXPLAIN` output.

- `id` (JSON name: `select_id`)

  The `SELECT` identifier. This is the sequential number of the `SELECT` within the query. The value can be `NULL` if the row refers to the union result of other rows. In this case, the `table` column shows a value like `<unionM,N>` to indicate that the row refers to the union of the rows with `id` values of *`M`* and *`N`*.

- `select_type` (JSON name: none)

  The type of `SELECT`, which can be any of those shown in the following table. A JSON-formatted `EXPLAIN` exposes the `SELECT` type as a property of a `query_block`, unless it is `SIMPLE` or `PRIMARY`. The JSON names (where applicable) are also shown in the table.

`select_type` Value	JSON Name	Meaning
`SIMPLE`	None	Simple `SELECT` (not using `UNION` or subqueries)
`PRIMARY`	None	Outermost `SELECT`
`UNION`	None	Second or later `SELECT` statement in a `UNION`
`DEPENDENT UNION`	`dependent` (`true`)	Second or later `SELECT` statement in a `UNION`, dependent on outer query
`UNION RESULT`	`union_result`	Result of a `UNION`.
`SUBQUERY`	None	First `SELECT` in subquery
`DEPENDENT SUBQUERY`	`dependent` (`true`)	First `SELECT` in subquery, dependent on outer query
`DERIVED`	None	Derived table
`DEPENDENT DERIVED`	`dependent` (`true`)	Derived table dependent on another table
`MATERIALIZED`	`materialized_from_subquer`	Materialized subquery

`select_type` Value	JSON Name	Meaning
`UNCACHEABLE SUBQUERY`	`cacheable` (`false`)	A subquery for which the result cannot be cached and must be re-evaluated for each row of the outer query
`UNCACHEABLE UNION`	`cacheable` (`false`)	The second or later select in a `UNION` that belongs to an uncacheable subquery (see `UNCACHEABLE SUBQUERY`)

`DEPENDENT` typically signifies the use of a correlated subquery. See Section 13.2.15.7, "Correlated Subqueries".

`DEPENDENT SUBQUERY` evaluation differs from `UNCACHEABLE SUBQUERY` evaluation. For `DEPENDENT SUBQUERY`, the subquery is re-evaluated only once for each set of different values of the variables from its outer context. For `UNCACHEABLE SUBQUERY`, the subquery is re-evaluated for each row of the outer context.

When you specify `FORMAT=JSON` with `EXPLAIN`, the output has no single property directly equivalent to `select_type`; the `query_block` property corresponds to a given `SELECT`. Properties equivalent to most of the `SELECT` subquery types just shown are available (an example being `materialized_from_subquery` for `MATERIALIZED`), and are displayed when appropriate. There are no JSON equivalents for `SIMPLE` or `PRIMARY`.

The `select_type` value for non-`SELECT` statements displays the statement type for affected tables. For example, `select_type` is `DELETE` for `DELETE` statements.

- `table` (JSON name: `table_name`)

  The name of the table to which the row of output refers. This can also be one of the following values:

  - `<union*M*,*N*>`: The row refers to the union of the rows with `id` values of *`M`* and *`N`*.
  - `<derived*N*>`: The row refers to the derived table result for the row with an `id` value of *`N`*. A derived table may result, for example, from a subquery in the `FROM` clause.
  - `<subquery*N*>`: The row refers to the result of a materialized subquery for the row with an `id` value of *`N`*. See Section 8.2.2.2, "Optimizing Subqueries with Materialization".

- `partitions` (JSON name: `partitions`)

  The partitions from which records would be matched by the query. The value is `NULL` for nonpartitioned tables. See Section 24.3.5, "Obtaining Information About Partitions".

- `type` (JSON name: `access_type`)

  The join type. For descriptions of the different types, see `EXPLAIN` Join Types.

- `possible_keys` (JSON name: `possible_keys`)

  The `possible_keys` column indicates the indexes from which MySQL can choose to find the rows in this table. Note that this column is totally independent of the order of the tables as displayed in the output from `EXPLAIN`. That means that some of the keys in `possible_keys` might not be usable in practice with the generated table order.

  If this column is `NULL` (or undefined in JSON-formatted output), there are no relevant indexes. In this case, you may be able to improve the performance of your query by examining the `WHERE` clause to check whether it refers to some column or columns that would be suitable for indexing. If so, create an appropriate index and check the query with `EXPLAIN` again. See Section 13.1.9, "ALTER TABLE Statement".

To see what indexes a table has, use `SHOW INDEX FROM tbl_name`.

- `key` (JSON name: `key`)

  The `key` column indicates the key (index) that MySQL actually decided to use. If MySQL decides to use one of the `possible_keys` indexes to look up rows, that index is listed as the key value.

  It is possible that `key` may name an index that is not present in the `possible_keys` value. This can happen if none of the `possible_keys` indexes are suitable for looking up rows, but all the columns selected by the query are columns of some other index. That is, the named index covers the selected columns, so although it is not used to determine which rows to retrieve, an index scan is more efficient than a data row scan.

  For `InnoDB`, a secondary index might cover the selected columns even if the query also selects the primary key because `InnoDB` stores the primary key value with each secondary index. If `key` is `NULL`, MySQL found no index to use for executing the query more efficiently.

  To force MySQL to use or ignore an index listed in the `possible_keys` column, use `FORCE INDEX`, `USE INDEX`, or `IGNORE INDEX` in your query. See Section 8.9.4, “Index Hints”.

  For `MyISAM` tables, running `ANALYZE TABLE` helps the optimizer choose better indexes. For `MyISAM` tables, `myisamchk --analyze` does the same. See Section 13.7.3.1, “ANALYZE TABLE Statement”, and Section 7.6, “MyISAM Table Maintenance and Crash Recovery”.

- `key_len` (JSON name: `key_length`)

  The `key_len` column indicates the length of the key that MySQL decided to use. The value of `key_len` enables you to determine how many parts of a multiple-part key MySQL actually uses. If the `key` column says `NULL`, the `key_len` column also says `NULL`.

  Due to the key storage format, the key length is one greater for a column that can be `NULL` than for a `NOT NULL` column.

- `ref` (JSON name: `ref`)

  The `ref` column shows which columns or constants are compared to the index named in the `key` column to select rows from the table.

  If the value is `func`, the value used is the result of some function. To see which function, use `SHOW WARNINGS` following `EXPLAIN` to see the extended `EXPLAIN` output. The function might actually be an operator such as an arithmetic operator.

- `rows` (JSON name: `rows`)

  The `rows` column indicates the number of rows MySQL believes it must examine to execute the query.

  For `InnoDB` tables, this number is an estimate, and may not always be exact.

- `filtered` (JSON name: `filtered`)

  The `filtered` column indicates an estimated percentage of table rows that are filtered by the table condition. The maximum value is 100, which means no filtering of rows occurred. Values decreasing from 100 indicate increasing amounts of filtering. `rows` shows the estimated number of rows examined and `rows` × `filtered` shows the number of rows that are joined with the following table. For example, if `rows` is 1000 and `filtered` is 50.00 (50%), the number of rows to be joined with the following table is 1000 × 50% = 500.

- `Extra` (JSON name: none)

  This column contains additional information about how MySQL resolves the query. For descriptions of the different values, see `EXPLAIN` Extra Information.

  There is no single JSON property corresponding to the `Extra` column; however, values that can occur in this column are exposed as JSON properties, or as the text of the `message` property.

EXPLAIN Join Types

The `type` column of `EXPLAIN` output describes how tables are joined. In JSON-formatted output, these are found as values of the `access_type` property. The following list describes the join types, ordered from the best type to the worst:

- `system`

  The table has only one row (= system table). This is a special case of the `const` join type.

- `const`

  The table has at most one matching row, which is read at the start of the query. Because there is only one row, values from the column in this row can be regarded as constants by the rest of the optimizer. `const` tables are very fast because they are read only once.

  `const` is used when you compare all parts of a `PRIMARY KEY` or `UNIQUE` index to constant values. In the following queries, *`tbl_name`* can be used as a `const` table:

```
SELECT * FROM tbl_name WHERE primary_key=1;

SELECT * FROM tbl_name
  WHERE primary_key_part1=1 AND primary_key_part2=2;
```

- `eq_ref`

  One row is read from this table for each combination of rows from the previous tables. Other than the `system` and `const` types, this is the best possible join type. It is used when all parts of an index are used by the join and the index is a `PRIMARY KEY` or `UNIQUE NOT NULL` index.

  `eq_ref` can be used for indexed columns that are compared using the `=` operator. The comparison value can be a constant or an expression that uses columns from tables that are read before this table. In the following examples, MySQL can use an `eq_ref` join to process *`ref_table`*:

```
SELECT * FROM ref_table,other_table
  WHERE ref_table.key_column=other_table.column;

SELECT * FROM ref_table,other_table
  WHERE ref_table.key_column_part1=other_table.column
  AND ref_table.key_column_part2=1;
```

- `ref`

  All rows with matching index values are read from this table for each combination of rows from the previous tables. `ref` is used if the join uses only a leftmost prefix of the key or if the key is not a `PRIMARY KEY` or `UNIQUE` index (in other words, if the join cannot select a single row based on the key value). If the key that is used matches only a few rows, this is a good join type.

  `ref` can be used for indexed columns that are compared using the `=` or `<=>` operator. In the following examples, MySQL can use a `ref` join to process *`ref_table`*:

```
SELECT * FROM ref_table WHERE key_column=expr;

SELECT * FROM ref_table,other_table
  WHERE ref_table.key_column=other_table.column;
```

```
SELECT * FROM ref_table,other_table
  WHERE ref_table.key_column_part1=other_table.column
  AND ref_table.key_column_part2=1;
```

- `fulltext`

  The join is performed using a `FULLTEXT` index.

- `ref_or_null`

  This join type is like `ref`, but with the addition that MySQL does an extra search for rows that contain `NULL` values. This join type optimization is used most often in resolving subqueries. In the following examples, MySQL can use a `ref_or_null` join to process *`ref_table`*:

  ```
  SELECT * FROM ref_table
    WHERE key_column=expr OR key_column IS NULL;
  ```

  See Section 8.2.1.15, “IS NULL Optimization”.

- `index_merge`

  This join type indicates that the Index Merge optimization is used. In this case, the `key` column in the output row contains a list of indexes used, and `key_len` contains a list of the longest key parts for the indexes used. For more information, see Section 8.2.1.3, “Index Merge Optimization”.

- `unique_subquery`

  This type replaces `eq_ref` for some `IN` subqueries of the following form:

  ```
  value IN (SELECT primary_key FROM single_table WHERE some_expr)
  ```

  `unique_subquery` is just an index lookup function that replaces the subquery completely for better efficiency.

- `index_subquery`

  This join type is similar to `unique_subquery`. It replaces `IN` subqueries, but it works for nonunique indexes in subqueries of the following form:

  ```
  value IN (SELECT key_column FROM single_table WHERE some_expr)
  ```

- `range`

  Only rows that are in a given range are retrieved, using an index to select the rows. The `key` column in the output row indicates which index is used. The `key_len` contains the longest key part that was used. The `ref` column is `NULL` for this type.

  `range` can be used when a key column is compared to a constant using any of the `=`, `<>`, `>`, `>=`, `<`, `<=`, `IS NULL`, `<=>`, `BETWEEN`, `LIKE`, or `IN()` operators:

  ```
  SELECT * FROM tbl_name
    WHERE key_column = 10;

  SELECT * FROM tbl_name
    WHERE key_column BETWEEN 10 and 20;

  SELECT * FROM tbl_name
    WHERE key_column IN (10,20,30);

  SELECT * FROM tbl_name
    WHERE key_part1 = 10 AND key_part2 IN (10,20,30);
  ```

- `index`

  The `index` join type is the same as `ALL`, except that the index tree is scanned. This occurs two ways:

  - If the index is a covering index for the queries and can be used to satisfy all data required from the table, only the index tree is scanned. In this case, the `Extra` column says `Using index`. An index-only scan usually is faster than `ALL` because the size of the index usually is smaller than the table data.
  - A full table scan is performed using reads from the index to look up data rows in index order. `Uses index` does not appear in the `Extra` column.

  MySQL can use this join type when the query uses only columns that are part of a single index.

- `ALL`

  A full table scan is done for each combination of rows from the previous tables. This is normally not good if the table is the first table not marked `const`, and usually *very* bad in all other cases. Normally, you can avoid `ALL` by adding indexes that enable row retrieval from the table based on constant values or column values from earlier tables.

EXPLAIN Extra Information

The `Extra` column of `EXPLAIN` output contains additional information about how MySQL resolves the query. The following list explains the values that can appear in this column. Each item also indicates for JSON-formatted output which property displays the `Extra` value. For some of these, there is a specific property. The others display as the text of the `message` property.

If you want to make your queries as fast as possible, look out for `Extra` column values of `Using filesort` and `Using temporary`, or, in JSON-formatted `EXPLAIN` output, for `using_filesort` and `using_temporary_table` properties equal to `true`.

- `Backward index scan` (JSON: `backward_index_scan`)

  The optimizer is able to use a descending index on an `InnoDB` table. Shown together with `Using index`. For more information, see Section 8.3.13, “Descending Indexes”.

- `Child of '`*`table`*`' pushed join@1` (JSON: `message` text)

  This table is referenced as the child of *`table`* in a join that can be pushed down to the NDB kernel. Applies only in NDB Cluster, when pushed-down joins are enabled. See the description of the `ndb_join_pushdown` server system variable for more information and examples.

- `const row not found` (JSON property: `const_row_not_found`)

  For a query such as `SELECT ... FROM` *`tbl_name`*, the table was empty.

- `Deleting all rows` (JSON property: `message`)

  For `DELETE`, some storage engines (such as `MyISAM`) support a handler method that removes all table rows in a simple and fast way. This `Extra` value is displayed if the engine uses this optimization.

- `Distinct` (JSON property: `distinct`)

  MySQL is looking for distinct values, so it stops searching for more rows for the current row combination after it has found the first matching row.

- `FirstMatch(`*`tbl_name`*`)` (JSON property: `first_match`)

  The semijoin FirstMatch join shortcutting strategy is used for *`tbl_name`*.

- `Full scan on NULL key` (JSON property: `message`)

  This occurs for subquery optimization as a fallback strategy when the optimizer cannot use an index-lookup access method.

- `Impossible HAVING` (JSON property: `message`)

  The `HAVING` clause is always false and cannot select any rows.

- `Impossible WHERE` (JSON property: `message`)

  The `WHERE` clause is always false and cannot select any rows.

- `Impossible WHERE noticed after reading const tables` (JSON property: `message`)

  MySQL has read all `const` (and `system`) tables and notice that the `WHERE` clause is always false.

- `LooseScan(`*`m`*`..`*`n`*`)` (JSON property: `message`)

  The semijoin LooseScan strategy is used. *`m`* and *`n`* are key part numbers.

- `No matching min/max row` (JSON property: `message`)

  No row satisfies the condition for a query such as `SELECT MIN(...) FROM ... WHERE` *`condition`*.

- `no matching row in const table` (JSON property: `message`)

  For a query with a join, there was an empty table or a table with no rows satisfying a unique index condition.

- `No matching rows after partition pruning` (JSON property: `message`)

  For `DELETE` or `UPDATE`, the optimizer found nothing to delete or update after partition pruning. It is similar in meaning to `Impossible WHERE` for `SELECT` statements.

- `No tables used` (JSON property: `message`)

  The query has no `FROM` clause, or has a `FROM DUAL` clause.

  For `INSERT` or `REPLACE` statements, `EXPLAIN` displays this value when there is no `SELECT` part. For example, it appears for `EXPLAIN INSERT INTO t VALUES(10)` because that is equivalent to `EXPLAIN INSERT INTO t SELECT 10 FROM DUAL`.

- `Not exists` (JSON property: `message`)

  MySQL was able to do a `LEFT JOIN` optimization on the query and does not examine more rows in this table for the previous row combination after it finds one row that matches the `LEFT JOIN` criteria. Here is an example of the type of query that can be optimized this way:

  ```
  SELECT * FROM t1 LEFT JOIN t2 ON t1.id=t2.id
    WHERE t2.id IS NULL;
  ```

  Assume that `t2.id` is defined as `NOT NULL`. In this case, MySQL scans `t1` and looks up the rows in `t2` using the values of `t1.id`. If MySQL finds a matching row in `t2`, it knows that `t2.id` can never be `NULL`, and does not scan through the rest of the rows in `t2` that have the same `id` value. In other words, for each row in `t1`, MySQL needs to do only a single lookup in `t2`, regardless of how many rows actually match in `t2`.

  In MySQL 8.0.17 and later, this can also indicate that a `WHERE` condition of the form `NOT IN (`*`subquery`*`)` or `NOT EXISTS (`*`subquery`*`)` has been transformed internally into an antijoin. This removes the subquery and brings its tables into the plan for the topmost query, providing improved cost planning. By merging semijoins and antijoins, the optimizer can reorder tables in the execution plan more freely, in some cases resulting in a faster plan.

  You can see when an antijoin transformation is performed for a given query by checking the `Message` column from `SHOW WARNINGS` following execution of `EXPLAIN`, or in the output of `EXPLAIN FORMAT=TREE`.

Note

An antijoin is the complement of a semijoin *`table_a JOIN table_b ON condition`*. The antijoin returns all rows from *`table_a`* for which there is *no* row in *`table_b`* which matches *`condition`*.

- `Plan isn't ready yet` (JSON property: none)

  This value occurs with `EXPLAIN FOR CONNECTION` when the optimizer has not finished creating the execution plan for the statement executing in the named connection. If execution plan output comprises multiple lines, any or all of them could have this `Extra` value, depending on the progress of the optimizer in determining the full execution plan.

- `Range checked for each record (index map: N)` (JSON property: `message`)

  MySQL found no good index to use, but found that some of indexes might be used after column values from preceding tables are known. For each row combination in the preceding tables, MySQL checks whether it is possible to use a `range` or `index_merge` access method to retrieve rows. This is not very fast, but is faster than performing a join with no index at all. The applicability criteria are as described in Section 8.2.1.2, “Range Optimization”, and Section 8.2.1.3, “Index Merge Optimization”, with the exception that all column values for the preceding table are known and considered to be constants.

  Indexes are numbered beginning with 1, in the same order as shown by `SHOW INDEX` for the table. The index map value *`N`* is a bitmask value that indicates which indexes are candidates. For example, a value of `0x19` (binary 11001) means that indexes 1, 4, and 5 are considered.

- `Recursive` (JSON property: `recursive`)

  This indicates that the row applies to the recursive `SELECT` part of a recursive common table expression. See Section 13.2.20, “WITH (Common Table Expressions)”.

- `Rematerialize` (JSON property: `rematerialize`)

  `Rematerialize (X,...)` is displayed in the `EXPLAIN` row for table `T`, where `X` is any lateral derived table whose rematerialization is triggered when a new row of `T` is read. For example:

  ```
  SELECT
    ...
  FROM
    t,
    LATERAL (derived table that refers to t) AS dt
  ...
  ```

  The content of the derived table is rematerialized to bring it up to date each time a new row of `t` is processed by the top query.

- `Scanned N databases` (JSON property: `message`)

  This indicates how many directory scans the server performs when processing a query for `INFORMATION_SCHEMA` tables, as described in Section 8.2.3, “Optimizing INFORMATION_SCHEMA Queries”. The value of *`N`* can be 0, 1, or `all`.

- `Select tables optimized away` (JSON property: `message`)

  The optimizer determined 1) that at most one row should be returned, and 2) that to produce this row, a deterministic set of rows must be read. When the rows to be read can be read during the

optimization phase (for example, by reading index rows), there is no need to read any tables during query execution.

The first condition is fulfilled when the query is implicitly grouped (contains an aggregate function but no `GROUP BY` clause). The second condition is fulfilled when one row lookup is performed per index used. The number of indexes read determines the number of rows to read.

Consider the following implicitly grouped query:

```
SELECT MIN(c1), MIN(c2) FROM t1;
```

Suppose that `MIN(c1)` can be retrieved by reading one index row and `MIN(c2)` can be retrieved by reading one row from a different index. That is, for each column `c1` and `c2`, there exists an index where the column is the first column of the index. In this case, one row is returned, produced by reading two deterministic rows.

This `Extra` value does not occur if the rows to read are not deterministic. Consider this query:

```
SELECT MIN(c2) FROM t1 WHERE c1 <= 10;
```

Suppose that `(c1, c2)` is a covering index. Using this index, all rows with `c1 <= 10` must be scanned to find the minimum `c2` value. By contrast, consider this query:

```
SELECT MIN(c2) FROM t1 WHERE c1 = 10;
```

In this case, the first index row with `c1 = 10` contains the minimum `c2` value. Only one row must be read to produce the returned row.

For storage engines that maintain an exact row count per table (such as `MyISAM`, but not `InnoDB`), this `Extra` value can occur for `COUNT(*)` queries for which the `WHERE` clause is missing or always true and there is no `GROUP BY` clause. (This is an instance of an implicitly grouped query where the storage engine influences whether a deterministic number of rows can be read.)

- `Skip_open_table`, `Open_frm_only`, `Open_full_table` (JSON property: `message`)

  These values indicate file-opening optimizations that apply to queries for `INFORMATION_SCHEMA` tables.

  - `Skip_open_table`: Table files do not need to be opened. The information is already available from the data dictionary.
  - `Open_frm_only`: Only the data dictionary need be read for table information.
  - `Open_full_table`: Unoptimized information lookup. Table information must be read from the data dictionary and by reading table files.

- `Start temporary`, `End temporary` (JSON property: `message`)

  This indicates temporary table use for the semijoin Duplicate Weedout strategy.

- `unique row not found` (JSON property: `message`)

  For a query such as `SELECT ... FROM tbl_name`, no rows satisfy the condition for a `UNIQUE` index or `PRIMARY KEY` on the table.

- `Using filesort` (JSON property: `using_filesort`)

  MySQL must do an extra pass to find out how to retrieve the rows in sorted order. The sort is done by going through all rows according to the join type and storing the sort key and pointer to the row for all rows that match the `WHERE` clause. The keys then are sorted and the rows are retrieved in sorted order. See Section 8.2.1.16, “ORDER BY Optimization”.

- `Using index` (JSON property: `using_index`)

The column information is retrieved from the table using only information in the index tree without having to do an additional seek to read the actual row. This strategy can be used when the query uses only columns that are part of a single index.

For `InnoDB` tables that have a user-defined clustered index, that index can be used even when `Using index` is absent from the `Extra` column. This is the case if `type` is `index` and `key` is `PRIMARY`.

Information about any covering indexes used is shown for `EXPLAIN FORMAT=TRADITIONAL` and `EXPLAIN FORMAT=JSON`. Beginning with MySQL 8.0.27, it is also shown for `EXPLAIN FORMAT=TREE`.

- `Using index condition` (JSON property: `using_index_condition`)

  Tables are read by accessing index tuples and testing them first to determine whether to read full table rows. In this way, index information is used to defer ("push down") reading full table rows unless it is necessary. See Section 8.2.1.6, "Index Condition Pushdown Optimization".

- `Using index for group-by` (JSON property: `using_index_for_group_by`)

  Similar to the `Using index` table access method, `Using index for group-by` indicates that MySQL found an index that can be used to retrieve all columns of a `GROUP BY` or `DISTINCT` query without any extra disk access to the actual table. Additionally, the index is used in the most efficient way so that for each group, only a few index entries are read. For details, see Section 8.2.1.17, "GROUP BY Optimization".

- `Using index for skip scan` (JSON property: `using_index_for_skip_scan`)

  Indicates that the Skip Scan access method is used. See Skip Scan Range Access Method.

- `Using join buffer (Block Nested Loop)`, `Using join buffer (Batched Key Access)`, `Using join buffer (hash join)` (JSON property: `using_join_buffer`)

  Tables from earlier joins are read in portions into the join buffer, and then their rows are used from the buffer to perform the join with the current table. `(Block Nested Loop)` indicates use of the Block Nested-Loop algorithm, `(Batched Key Access)` indicates use of the Batched Key Access algorithm, and `(hash join)` indicates use of a hash join. That is, the keys from the table on the preceding line of the `EXPLAIN` output are buffered, and the matching rows are fetched in batches from the table represented by the line in which `Using join buffer` appears.

  In JSON-formatted output, the value of `using_join_buffer` is always one of `Block Nested Loop`, `Batched Key Access`, or `hash join`.

  Hash joins are available beginning with MySQL 8.0.18; the Block Nested-Loop algorithm is not used in MySQL 8.0.20 or later MySQL releases. For more information about these optimizations, see Section 8.2.1.4, "Hash Join Optimization", and Block Nested-Loop Join Algorithm.

  See Batched Key Access Joins, for information about the Batched Key Access algorithm.

- `Using MRR` (JSON property: `message`)

  Tables are read using the Multi-Range Read optimization strategy. See Section 8.2.1.11, "Multi-Range Read Optimization".

- `Using sort_union(...)`, `Using union(...)`, `Using intersect(...)` (JSON property: `message`)

  These indicate the particular algorithm showing how index scans are merged for the `index_merge` join type. See Section 8.2.1.3, "Index Merge Optimization".

- `Using temporary` (JSON property: `using_temporary_table`)

To resolve the query, MySQL needs to create a temporary table to hold the result. This typically happens if the query contains `GROUP BY` and `ORDER BY` clauses that list columns differently.

- `Using where` (JSON property: `attached_condition`)

  A `WHERE` clause is used to restrict which rows to match against the next table or send to the client. Unless you specifically intend to fetch or examine all rows from the table, you may have something wrong in your query if the `Extra` value is not `Using where` and the table join type is `ALL` or `index`.

  `Using where` has no direct counterpart in JSON-formatted output; the `attached_condition` property contains any `WHERE` condition used.

- `Using where with pushed condition` (JSON property: `message`)

  This item applies to `NDB` tables *only*. It means that NDB Cluster is using the Condition Pushdown optimization to improve the efficiency of a direct comparison between a nonindexed column and a constant. In such cases, the condition is "pushed down" to the cluster's data nodes and is evaluated on all data nodes simultaneously. This eliminates the need to send nonmatching rows over the network, and can speed up such queries by a factor of 5 to 10 times over cases where Condition Pushdown could be but is not used. For more information, see Section 8.2.1.5, "Engine Condition Pushdown Optimization".

- `Zero limit` (JSON property: `message`)

  The query had a `LIMIT 0` clause and cannot select any rows.

EXPLAIN Output Interpretation

You can get a good indication of how good a join is by taking the product of the values in the `rows` column of the `EXPLAIN` output. This should tell you roughly how many rows MySQL must examine to execute the query. If you restrict queries with the `max_join_size` system variable, this row product also is used to determine which multiple-table `SELECT` statements to execute and which to abort. See Section 5.1.1, "Configuring the Server".

The following example shows how a multiple-table join can be optimized progressively based on the information provided by `EXPLAIN`.

Suppose that you have the `SELECT` statement shown here and that you plan to examine it using `EXPLAIN`:

```
EXPLAIN SELECT tt.TicketNumber, tt.TimeIn,
               tt.ProjectReference, tt.EstimatedShipDate,
               tt.ActualShipDate, tt.ClientID,
               tt.ServiceCodes, tt.RepetitiveID,
               tt.CurrentProcess, tt.CurrentDPPerson,
               tt.RecordVolume, tt.DPPrinted, et.COUNTRY,
               et_1.COUNTRY, do.CUSTNAME
        FROM tt, et, et AS et_1, do
        WHERE tt.SubmitTime IS NULL
          AND tt.ActualPC = et.EMPLOYID
          AND tt.AssignedPC = et_1.EMPLOYID
          AND tt.ClientID = do.CUSTNMBR;
```

For this example, make the following assumptions:

- The columns being compared have been declared as follows.

Table	Column	Data Type
`tt`	`ActualPC`	`CHAR(10)`
`tt`	`AssignedPC`	`CHAR(10)`
`tt`	`ClientID`	`CHAR(10)`

Table	Column	Data Type
et	EMPLOYID	CHAR(15)
do	CUSTNMBR	CHAR(15)

- The tables have the following indexes.

Table	Index
tt	ActualPC
tt	AssignedPC
tt	ClientID
et	EMPLOYID (primary key)
do	CUSTNMBR (primary key)

- The tt.ActualPC values are not evenly distributed.

Initially, before any optimizations have been performed, the EXPLAIN statement produces the following information:

```
table type possible_keys key  key_len ref  rows  Extra
et    ALL  PRIMARY       NULL NULL    NULL 74
do    ALL  PRIMARY       NULL NULL    NULL 2135
et_1  ALL  PRIMARY       NULL NULL    NULL 74
tt    ALL  AssignedPC,   NULL NULL    NULL 3872
           ClientID,
           ActualPC
      Range checked for each record (index map: 0x23)
```

Because type is ALL for each table, this output indicates that MySQL is generating a Cartesian product of all the tables; that is, every combination of rows. This takes quite a long time, because the product of the number of rows in each table must be examined. For the case at hand, this product is 74 × 2135 × 74 × 3872 = 45,268,558,720 rows. If the tables were bigger, you can only imagine how long it would take.

One problem here is that MySQL can use indexes on columns more efficiently if they are declared as the same type and size. In this context, VARCHAR and CHAR are considered the same if they are declared as the same size. tt.ActualPC is declared as CHAR(10) and et.EMPLOYID is CHAR(15), so there is a length mismatch.

To fix this disparity between column lengths, use ALTER TABLE to lengthen ActualPC from 10 characters to 15 characters:

```
mysql> ALTER TABLE tt MODIFY ActualPC VARCHAR(15);
```

Now tt.ActualPC and et.EMPLOYID are both VARCHAR(15). Executing the EXPLAIN statement again produces this result:

```
table type   possible_keys key     key_len ref         rows    Extra
tt    ALL    AssignedPC,   NULL    NULL    NULL        3872    Using
             ClientID,                                        where
             ActualPC
do    ALL    PRIMARY       NULL    NULL    NULL        2135
      Range checked for each record (index map: 0x1)
et_1  ALL    PRIMARY       NULL    NULL    NULL        74
      Range checked for each record (index map: 0x1)
et    eq_ref PRIMARY       PRIMARY 15      tt.ActualPC 1
```

This is not perfect, but is much better: The product of the rows values is less by a factor of 74. This version executes in a couple of seconds.

A second alteration can be made to eliminate the column length mismatches for the tt.AssignedPC = et_1.EMPLOYID and tt.ClientID = do.CUSTNMBR comparisons:

```
mysql> ALTER TABLE tt MODIFY AssignedPC VARCHAR(15),
                      MODIFY ClientID   VARCHAR(15);
```

After that modification, EXPLAIN produces the output shown here:

```
table type   possible_keys key      key_len ref           rows Extra
et    ALL    PRIMARY       NULL     NULL    NULL          74
tt    ref    AssignedPC,   ActualPC 15      et.EMPLOYID   52   Using
             ClientID,                                        where
             ActualPC
et_1  eq_ref PRIMARY       PRIMARY  15      tt.AssignedPC 1
do    eq_ref PRIMARY       PRIMARY  15      tt.ClientID   1
```

At this point, the query is optimized almost as well as possible. The remaining problem is that, by default, MySQL assumes that values in the tt.ActualPC column are evenly distributed, and that is not the case for the tt table. Fortunately, it is easy to tell MySQL to analyze the key distribution:

```
mysql> ANALYZE TABLE tt;
```

With the additional index information, the join is perfect and EXPLAIN produces this result:

```
table type   possible_keys key     key_len ref           rows Extra
tt    ALL    AssignedPC    NULL    NULL    NULL          3872 Using
             ClientID,                                        where
             ActualPC
et    eq_ref PRIMARY       PRIMARY 15      tt.ActualPC   1
et_1  eq_ref PRIMARY       PRIMARY 15      tt.AssignedPC 1
do    eq_ref PRIMARY       PRIMARY 15      tt.ClientID   1
```

The rows column in the output from EXPLAIN is an educated guess from the MySQL join optimizer. Check whether the numbers are even close to the truth by comparing the rows product with the actual number of rows that the query returns. If the numbers are quite different, you might get better performance by using STRAIGHT_JOIN in your SELECT statement and trying to list the tables in a different order in the FROM clause. (However, STRAIGHT_JOIN may prevent indexes from being used because it disables semijoin transformations. See Section 8.2.2.1, “Optimizing IN and EXISTS Subquery Predicates with Semijoin Transformations”.)

It is possible in some cases to execute statements that modify data when EXPLAIN SELECT is used with a subquery; for more information, see Section 13.2.15.8, “Derived Tables”.

8.8.3 Extended EXPLAIN Output Format

The EXPLAIN statement produces extra (“extended”) information that is not part of EXPLAIN output but can be viewed by issuing a SHOW WARNINGS statement following EXPLAIN. As of MySQL 8.0.12, extended information is available for SELECT, DELETE, INSERT, REPLACE, and UPDATE statements. Prior to 8.0.12, extended information is available only for SELECT statements.

The Message value in SHOW WARNINGS output displays how the optimizer qualifies table and column names in the SELECT statement, what the SELECT looks like after the application of rewriting and optimization rules, and possibly other notes about the optimization process.

The extended information displayable with a SHOW WARNINGS statement following EXPLAIN is produced only for SELECT statements. SHOW WARNINGS displays an empty result for other explainable statements (DELETE, INSERT, REPLACE, and UPDATE).

Here is an example of extended EXPLAIN output:

```
mysql> EXPLAIN
       SELECT t1.a, t1.a IN (SELECT t2.a FROM t2) FROM t1\G
*************************** 1. row ***************************
           id: 1
  select_type: PRIMARY
        table: t1
         type: index
possible_keys: NULL
          key: PRIMARY
```

```
      key_len: 4
          ref: NULL
         rows: 4
     filtered: 100.00
        Extra: Using index
*************************** 2. row ***************************
           id: 2
  select_type: SUBQUERY
        table: t2
         type: index
possible_keys: a
          key: a
      key_len: 5
          ref: NULL
         rows: 3
     filtered: 100.00
        Extra: Using index
2 rows in set, 1 warning (0.00 sec)

mysql> SHOW WARNINGS\G
*************************** 1. row ***************************
  Level: Note
   Code: 1003
Message: /* select#1 */ select `test`.`t1`.`a` AS `a`,
         <in_optimizer>(`test`.`t1`.`a`,`test`.`t1`.`a` in
         ( <materialize> (/* select#2 */ select `test`.`t2`.`a`
         from `test`.`t2` where 1 having 1 ),
         <primary_index_lookup>(`test`.`t1`.`a` in
         <temporary table> on <auto_key>
         where ((`test`.`t1`.`a` = `materialized-subquery`.`a`))))) AS `t1.a
         IN (SELECT t2.a FROM t2)` from `test`.`t1`
1 row in set (0.00 sec)
```

Because the statement displayed by `SHOW WARNINGS` may contain special markers to provide information about query rewriting or optimizer actions, the statement is not necessarily valid SQL and is not intended to be executed. The output may also include rows with `Message` values that provide additional non-SQL explanatory notes about actions taken by the optimizer.

The following list describes special markers that can appear in the extended output displayed by `SHOW WARNINGS`:

- `<auto_key>`

  An automatically generated key for a temporary table.

- `<cache>(`*`expr`*`)`

  The expression (such as a scalar subquery) is executed once and the resulting value is saved in memory for later use. For results consisting of multiple values, a temporary table may be created and `<temporary table>` is shown instead.

- `<exists>(`*`query fragment`*`)`

  The subquery predicate is converted to an `EXISTS` predicate and the subquery is transformed so that it can be used together with the `EXISTS` predicate.

- `<in_optimizer>(`*`query fragment`*`)`

  This is an internal optimizer object with no user significance.

- `<index_lookup>(`*`query fragment`*`)`

  The query fragment is processed using an index lookup to find qualifying rows.

- `<if>(`*`condition, expr1, expr2`*`)`

  If the condition is true, evaluate to *`expr1`*, otherwise *`expr2`*.

- `<is_not_null_test>(expr)`

  A test to verify that the expression does not evaluate to `NULL`.

- `<materialize>(query fragment)`

  Subquery materialization is used.

- `` `materialized-subquery`.col_name ``

  A reference to the column `col_name` in an internal temporary table materialized to hold the result from evaluating a subquery.

- `<primary_index_lookup>(query fragment)`

  The query fragment is processed using a primary key lookup to find qualifying rows.

- `<ref_null_helper>(expr)`

  This is an internal optimizer object with no user significance.

- `/* select#N */ select_stmt`

  The `SELECT` is associated with the row in non-extended `EXPLAIN` output that has an `id` value of `N`.

- `outer_tables semi join (inner_tables)`

  A semijoin operation. `inner_tables` shows the tables that were not pulled out. See Section 8.2.2.1, “Optimizing IN and EXISTS Subquery Predicates with Semijoin Transformations”.

- `<temporary table>`

  This represents an internal temporary table created to cache an intermediate result.

When some tables are of `const` or `system` type, expressions involving columns from these tables are evaluated early by the optimizer and are not part of the displayed statement. However, with `FORMAT=JSON`, some `const` table accesses are displayed as a `ref` access that uses a const value.

8.8.4 Obtaining Execution Plan Information for a Named Connection

To obtain the execution plan for an explainable statement executing in a named connection, use this statement:

```
EXPLAIN [options] FOR CONNECTION connection_id;
```

`EXPLAIN FOR CONNECTION` returns the `EXPLAIN` information that is currently being used to execute a query in a given connection. Because of changes to data (and supporting statistics) it may produce a different result from running `EXPLAIN` on the equivalent query text. This difference in behavior can be useful in diagnosing more transient performance problems. For example, if you are running a statement in one session that is taking a long time to complete, using `EXPLAIN FOR CONNECTION` in another session may yield useful information about the cause of the delay.

`connection_id` is the connection identifier, as obtained from the `INFORMATION_SCHEMA PROCESSLIST` table or the `SHOW PROCESSLIST` statement. If you have the `PROCESS` privilege, you can specify the identifier for any connection. Otherwise, you can specify the identifier only for your own connections. In all cases, you must have sufficient privileges to explain the query on the specified connection.

If the named connection is not executing a statement, the result is empty. Otherwise, `EXPLAIN FOR CONNECTION` applies only if the statement being executed in the named connection is explainable. This includes `SELECT`, `DELETE`, `INSERT`, `REPLACE`, and `UPDATE`. (However, `EXPLAIN FOR CONNECTION` does not work for prepared statements, even prepared statements of those types.)

If the named connection is executing an explainable statement, the output is what you would obtain by using `EXPLAIN` on the statement itself.

If the named connection is executing a statement that is not explainable, an error occurs. For example, you cannot name the connection identifier for your current session because `EXPLAIN` is not explainable:

```
mysql> SELECT CONNECTION_ID();
+-----------------+
| CONNECTION_ID() |
+-----------------+
|             373 |
+-----------------+
1 row in set (0.00 sec)

mysql> EXPLAIN FOR CONNECTION 373;
ERROR 1889 (HY000): EXPLAIN FOR CONNECTION command is supported
only for SELECT/UPDATE/INSERT/DELETE/REPLACE
```

The `Com_explain_other` status variable indicates the number of `EXPLAIN FOR CONNECTION` statements executed.

8.8.5 Estimating Query Performance

In most cases, you can estimate query performance by counting disk seeks. For small tables, you can usually find a row in one disk seek (because the index is probably cached). For bigger tables, you can estimate that, using B-tree indexes, you need this many seeks to find a row: `log(row_count) / log(index_block_length / 3 * 2 / (index_length + data_pointer_length)) + 1`.

In MySQL, an index block is usually 1,024 bytes and the data pointer is usually four bytes. For a 500,000-row table with a key value length of three bytes (the size of `MEDIUMINT`), the formula indicates `log(500,000)/log(1024/3*2/(3+4)) + 1` = `4` seeks.

This index would require storage of about 500,000 * 7 * 3/2 = 5.2MB (assuming a typical index buffer fill ratio of 2/3), so you probably have much of the index in memory and so need only one or two calls to read data to find the row.

For writes, however, you need four seek requests to find where to place a new index value and normally two seeks to update the index and write the row.

The preceding discussion does not mean that your application performance slowly degenerates by log *N*. As long as everything is cached by the OS or the MySQL server, things become only marginally slower as the table gets bigger. After the data gets too big to be cached, things start to go much slower until your applications are bound only by disk seeks (which increase by log *N*). To avoid this, increase the key cache size as the data grows. For `MyISAM` tables, the key cache size is controlled by the `key_buffer_size` system variable. See Section 5.1.1, “Configuring the Server”.

8.9 Controlling the Query Optimizer

MySQL provides optimizer control through system variables that affect how query plans are evaluated, switchable optimizations, optimizer and index hints, and the optimizer cost model.

The server maintains histogram statistics about column values in the `column_statistics` data dictionary table (see Section 8.9.6, “Optimizer Statistics”). Like other data dictionary tables, this table is not directly accessible by users. Instead, you can obtain histogram information by querying `INFORMATION_SCHEMA.COLUMN_STATISTICS`, which is implemented as a view on the data dictionary table. You can also perform histogram management using the `ANALYZE TABLE` statement.

8.9.1 Controlling Query Plan Evaluation

The task of the query optimizer is to find an optimal plan for executing an SQL query. Because the difference in performance between “good” and “bad” plans can be orders of magnitude (that is, seconds versus hours or even days), most query optimizers, including that of MySQL, perform a more

or less exhaustive search for an optimal plan among all possible query evaluation plans. For join queries, the number of possible plans investigated by the MySQL optimizer grows exponentially with the number of tables referenced in a query. For small numbers of tables (typically less than 7 to 10) this is not a problem. However, when larger queries are submitted, the time spent in query optimization may easily become the major bottleneck in the server's performance.

A more flexible method for query optimization enables the user to control how exhaustive the optimizer is in its search for an optimal query evaluation plan. The general idea is that the fewer plans that are investigated by the optimizer, the less time it spends in compiling a query. On the other hand, because the optimizer skips some plans, it may miss finding an optimal plan.

The behavior of the optimizer with respect to the number of plans it evaluates can be controlled using two system variables:

- The `optimizer_prune_level` variable tells the optimizer to skip certain plans based on estimates of the number of rows accessed for each table. Our experience shows that this kind of "educated guess" rarely misses optimal plans, and may dramatically reduce query compilation times. That is why this option is on (`optimizer_prune_level=1`) by default. However, if you believe that the optimizer missed a better query plan, this option can be switched off (`optimizer_prune_level=0`) with the risk that query compilation may take much longer. Note that, even with the use of this heuristic, the optimizer still explores a roughly exponential number of plans.
- The `optimizer_search_depth` variable tells how far into the "future" of each incomplete plan the optimizer should look to evaluate whether it should be expanded further. Smaller values of `optimizer_search_depth` may result in orders of magnitude smaller query compilation times. For example, queries with 12, 13, or more tables may easily require hours and even days to compile if `optimizer_search_depth` is close to the number of tables in the query. At the same time, if compiled with `optimizer_search_depth` equal to 3 or 4, the optimizer may compile in less than a minute for the same query. If you are unsure of what a reasonable value is for `optimizer_search_depth`, this variable can be set to 0 to tell the optimizer to determine the value automatically.

8.9.2 Switchable Optimizations

The `optimizer_switch` system variable enables control over optimizer behavior. Its value is a set of flags, each of which has a value of `on` or `off` to indicate whether the corresponding optimizer behavior is enabled or disabled. This variable has global and session values and can be changed at runtime. The global default can be set at server startup.

To see the current set of optimizer flags, select the variable value:

```
mysql> SELECT @@optimizer_switch\G
*************************** 1. row ***************************
@@optimizer_switch: index_merge=on,index_merge_union=on,
                    index_merge_sort_union=on,index_merge_intersection=on,
                    engine_condition_pushdown=on,index_condition_pushdown=on,
                    mrr=on,mrr_cost_based=on,block_nested_loop=on,
                    batched_key_access=off,materialization=on,semijoin=on,
                    loosescan=on,firstmatch=on,duplicateweedout=on,
                    subquery_materialization_cost_based=on,
                    use_index_extensions=on,condition_fanout_filter=on,
                    derived_merge=on,use_invisible_indexes=off,skip_scan=on,
                    hash_join=on,subquery_to_derived=off,
                    prefer_ordering_index=on,hypergraph_optimizer=off,
                    derived_condition_pushdown=on
1 row in set (0.00 sec)
```

To change the value of `optimizer_switch`, assign a value consisting of a comma-separated list of one or more commands:

```
SET [GLOBAL|SESSION] optimizer_switch='command[,command]...';
```

Each *`command`* value should have one of the forms shown in the following table.

Command Syntax	Meaning
`default`	Reset every optimization to its default value
`opt_name``=default`	Set the named optimization to its default value
`opt_name``=off`	Disable the named optimization
`opt_name``=on`	Enable the named optimization

The order of the commands in the value does not matter, although the `default` command is executed first if present. Setting an *`opt_name`* flag to `default` sets it to whichever of `on` or `off` is its default value. Specifying any given *`opt_name`* more than once in the value is not permitted and causes an error. Any errors in the value cause the assignment to fail with an error, leaving the value of `optimizer_switch` unchanged.

The following list describes the permissible *`opt_name`* flag names, grouped by optimization strategy:

- Batched Key Access Flags
  - `batched_key_access` (default `off`)

    Controls use of BKA join algorithm.

  For `batched_key_access` to have any effect when set to `on`, the `mrr` flag must also be `on`. Currently, the cost estimation for MRR is too pessimistic. Hence, it is also necessary for `mrr_cost_based` to be `off` for BKA to be used.

  For more information, see Section 8.2.1.12, “Block Nested-Loop and Batched Key Access Joins”.
- Block Nested-Loop Flags
  - `block_nested_loop` (default `on`)

    Controls use of BNL join algorithm. In MySQL 8.0.18 and later, this also controls use of hash joins, as do the `BNL` and `NO_BNL` optimizer hints. In MySQL 8.0.20 and later, block nested loop support is removed from the MySQL server, and this flag controls the use of hash joins only, as do the referenced optimizer hints.

  For more information, see Section 8.2.1.12, “Block Nested-Loop and Batched Key Access Joins”.
- Condition Filtering Flags
  - `condition_fanout_filter` (default `on`)

    Controls use of condition filtering.

  For more information, see Section 8.2.1.13, “Condition Filtering”.
- Derived Condition Pushdown Flags
  - `derived_condition_pushdown` (default `on`)

    Controls derived condition pushdown.

  For more information, see Section 8.2.2.5, “Derived Condition Pushdown Optimization”
- Derived Table Merging Flags
  - `derived_merge` (default `on`)

    Controls merging of derived tables and views into outer query block.

  The `derived_merge` flag controls whether the optimizer attempts to merge derived tables, view references, and common table expressions into the outer query block, assuming that no other rule

prevents merging; for example, an `ALGORITHM` directive for a view takes precedence over the `derived_merge` setting. By default, the flag is `on` to enable merging.

For more information, see Section 8.2.2.4, “Optimizing Derived Tables, View References, and Common Table Expressions with Merging or Materialization”.

- Engine Condition Pushdown Flags
  - `engine_condition_pushdown` (default `on`)

    Controls engine condition pushdown.

  For more information, see Section 8.2.1.5, “Engine Condition Pushdown Optimization”.
- Hash Join Flags
  - `hash_join` (default `on`)

    Controls hash joins in MySQL 8.0.18 only, and has no effect in any subsequent version. In MySQL 8.0.19 and later, to control hash join usage, use the `block_nested_loop` flag, instead.

  For more information, see Section 8.2.1.4, “Hash Join Optimization”.
- Index Condition Pushdown Flags
  - `index_condition_pushdown` (default `on`)

    Controls index condition pushdown.

  For more information, see Section 8.2.1.6, “Index Condition Pushdown Optimization”.
- Index Extensions Flags
  - `use_index_extensions` (default `on`)

    Controls use of index extensions.

  For more information, see Section 8.3.10, “Use of Index Extensions”.
- Index Merge Flags
  - `index_merge` (default `on`)

    Controls all Index Merge optimizations.
  - `index_merge_intersection` (default `on`)

    Controls the Index Merge Intersection Access optimization.
  - `index_merge_sort_union` (default `on`)

    Controls the Index Merge Sort-Union Access optimization.
  - `index_merge_union` (default `on`)

    Controls the Index Merge Union Access optimization.

  For more information, see Section 8.2.1.3, “Index Merge Optimization”.
- Index Visibility Flags
  - `use_invisible_indexes` (default `off`)

    Controls use of invisible indexes.

For more information, see Section 8.3.12, “Invisible Indexes”.

- Limit Optimization Flags

  - `prefer_ordering_index` (default `on`)

    Controls whether, in the case of a query having an `ORDER BY` or `GROUP BY` with a `LIMIT` clause, the optimizer tries to use an ordered index instead of an unordered index, a filesort, or some other optimization. This optimization is performed by default whenever the optimizer determines that using it would allow for faster execution of the query.

    Because the algorithm that makes this determination cannot handle every conceivable case (due in part to the assumption that the distribution of data is always more or less uniform), there are cases in which this optimization may not be desirable. Prior to MySQL 8.0.21, it was not possible to disable this optimization, but in MySQL 8.0.21 and later, while it remains the default behavior, it can be disabled by setting the `prefer_ordering_index` flag to `off`.

  For more information and examples, see Section 8.2.1.19, “LIMIT Query Optimization”.

- Multi-Range Read Flags

  - `mrr` (default `on`)

    Controls the Multi-Range Read strategy.

  - `mrr_cost_based` (default `on`)

    Controls use of cost-based MRR if `mrr=on`.

  For more information, see Section 8.2.1.11, “Multi-Range Read Optimization”.

- Semijoin Flags
  - `duplicateweedout` (default `on`)

    Controls the semijoin Duplicate Weedout strategy.
  - `firstmatch` (default `on`)

    Controls the semijoin FirstMatch strategy.
  - `loosescan` (default `on`)

    Controls the semijoin LooseScan strategy (not to be confused with Loose Index Scan for `GROUP BY`).
  - `semijoin` (default `on`)

    Controls all semijoin strategies.

    In MySQL 8.0.17 and later, this also applies to the antijoin optimization.

  The `semijoin`, `firstmatch`, `loosescan`, and `duplicateweedout` flags enable control over semijoin strategies. The `semijoin` flag controls whether semijoins are used. If it is set to `on`, the `firstmatch` and `loosescan` flags enable finer control over the permitted semijoin strategies.

  If the `duplicateweedout` semijoin strategy is disabled, it is not used unless all other applicable strategies are also disabled.

  If `semijoin` and `materialization` are both `on`, semijoins also use materialization where applicable. These flags are `on` by default.

  For more information, see Section 8.2.2.1, “Optimizing IN and EXISTS Subquery Predicates with Semijoin Transformations”.
- Skip Scan Flags
  - `skip_scan` (default `on`)

    Controls use of Skip Scan access method.

  For more information, see Skip Scan Range Access Method.
- Subquery Materialization Flags
  - `materialization` (default `on`)

    Controls materialization (including semijoin materialization).
  - `subquery_materialization_cost_based` (default `on`)

    Use cost-based materialization choice.

  The `materialization` flag controls whether subquery materialization is used. If `semijoin` and `materialization` are both `on`, semijoins also use materialization where applicable. These flags are `on` by default.

  The `subquery_materialization_cost_based` flag enables control over the choice between subquery materialization and `IN`-to-`EXISTS` subquery transformation. If the flag is `on` (the default), the optimizer performs a cost-based choice between subquery materialization and `IN`-to-`EXISTS`

subquery transformation if either method could be used. If the flag is `off`, the optimizer chooses subquery materialization over `IN`-to-`EXISTS` subquery transformation.

For more information, see Section 8.2.2, “Optimizing Subqueries, Derived Tables, View References, and Common Table Expressions”.

- Subquery Transformation Flags

  - `subquery_to_derived` (default `off`)

    Beginning with MySQL 8.0.21, the optimizer is able in many cases to transform a scalar subquery in a `SELECT`, `WHERE`, `JOIN`, or `HAVING` clause into a left outer joins on a derived table. (Depending on the nullability of the derived table, this can sometimes be simplified further to an inner join.) This can be done for a subquery which meets the following conditions:

    - The subquery does not make use of any nondeterministic functions, such as `RAND()`.
    - The subquery is not an `ANY` or `ALL` subquery which can be rewritten to use `MIN()` or `MAX()`.
    - The parent query does not set a user variable, since rewriting it may affect the order of execution, which could lead to unexpected results if the variable is accessed more than once in the same query.
    - The subquery should not be correlated, that is, it should not reference a column from a table in the outer query, or contain an aggregate that is evaluated in the outer query.

    Prior to MySQL 8.0.22, the subquery could not contain a `GROUP BY` clause.

    This optimization can also be applied to a table subquery which is the argument to `IN`, `NOT IN`, `EXISTS`, or `NOT EXISTS`, that does not contain a `GROUP BY`.

    The default value for this flag is `off`, since, in most cases, enabling this optimization does not produce any noticeable improvement in performance (and in many cases can even make queries run more slowly), but you can enable the optimization by setting the `subquery_to_derived` flag to `on`. It is primarily intended for use in testing.

    Example, using a scalar subquery:

```
d
mysql> CREATE TABLE t1(a INT);

mysql> CREATE TABLE t2(a INT);

mysql> INSERT INTO t1 VALUES ROW(1), ROW(2), ROW(3), ROW(4);

mysql> INSERT INTO t2 VALUES ROW(1), ROW(2);

mysql> SELECT * FROM t1
    ->     WHERE t1.a > (SELECT COUNT(a) FROM t2);
+------+
| a    |
+------+
|    3 |
|    4 |
+------+

mysql> SELECT @@optimizer_switch LIKE '%subquery_to_derived=off%';
+-----------------------------------------------------+
| @@optimizer_switch LIKE '%subquery_to_derived=off%' |
+-----------------------------------------------------+
|                                                   1 |
+-----------------------------------------------------+

mysql> EXPLAIN SELECT * FROM t1 WHERE t1.a > (SELECT COUNT(a) FROM t2)\G
*************************** 1. row ***************************
           id: 1
```

```
  select_type: PRIMARY
        table: t1
   partitions: NULL
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 4
     filtered: 33.33
        Extra: Using where
*************************** 2. row ***************************
           id: 2
  select_type: SUBQUERY
        table: t2
   partitions: NULL
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 2
     filtered: 100.00
        Extra: NULL

mysql> SET @@optimizer_switch='subquery_to_derived=on';


mysql> SELECT @@optimizer_switch LIKE '%subquery_to_derived=off%';
+-----------------------------------------------------+
| @@optimizer_switch LIKE '%subquery_to_derived=off%' |
+-----------------------------------------------------+
|                                                   0 |
+-----------------------------------------------------+

mysql> SELECT @@optimizer_switch LIKE '%subquery_to_derived=on%';
+----------------------------------------------------+
| @@optimizer_switch LIKE '%subquery_to_derived=on%' |
+----------------------------------------------------+
|                                                  1 |
+----------------------------------------------------+

mysql> EXPLAIN SELECT * FROM t1 WHERE t1.a > (SELECT COUNT(a) FROM t2)\G
*************************** 1. row ***************************
           id: 1
  select_type: PRIMARY
        table: <derived2>
   partitions: NULL
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 1
     filtered: 100.00
        Extra: NULL
*************************** 2. row ***************************
           id: 1
  select_type: PRIMARY
        table: t1
   partitions: NULL
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 4
     filtered: 33.33
        Extra: Using where; Using join buffer (hash join)
*************************** 3. row ***************************
           id: 2
  select_type: DERIVED
```

```
        table: t2
   partitions: NULL
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 2
     filtered: 100.00
        Extra: NULL
```

As can be seen from executing `SHOW WARNINGS` immediately following the second `EXPLAIN` statement, with the optimization enabled, the query `SELECT * FROM t1 WHERE t1.a > (SELECT COUNT(a) FROM t2)` is rewritten in a form similar to what is shown here:

```
SELECT t1.a FROM t1
    JOIN  ( SELECT COUNT(t2.a) AS c FROM t2 ) AS d
            WHERE t1.a > d.c;
```

Example, using a query with `IN (subquery)`:

```
mysql> DROP TABLE IF EXISTS t1, t2;

mysql> CREATE TABLE t1 (a INT, b INT);
mysql> CREATE TABLE t2 (a INT, b INT);

mysql> INSERT INTO t1 VALUES ROW(1,10), ROW(2,20), ROW(3,30);
mysql> INSERT INTO t2
    ->    VALUES ROW(1,10), ROW(2,20), ROW(3,30), ROW(1,110), ROW(2,120), ROW(3,130);

mysql> SELECT * FROM t1
    ->     WHERE   t1.b < 0
    ->             OR
    ->             t1.a IN (SELECT t2.a + 1 FROM t2);
+------+------+
| a    | b    |
+------+------+
|    2 |   20 |
|    3 |   30 |
+------+------+

mysql> SET @@optimizer_switch="subquery_to_derived=off";

mysql> EXPLAIN SELECT * FROM t1
    ->             WHERE   t1.b < 0
    ->                     OR
    ->                     t1.a IN (SELECT t2.a + 1 FROM t2)\G
*************************** 1. row ***************************
           id: 1
  select_type: PRIMARY
        table: t1
   partitions: NULL
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 3
     filtered: 100.00
        Extra: Using where
*************************** 2. row ***************************
           id: 2
  select_type: DEPENDENT SUBQUERY
        table: t2
   partitions: NULL
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 6
```

```
     filtered: 100.00
        Extra: Using where

mysql> SET @@optimizer_switch="subquery_to_derived=on";

mysql> EXPLAIN SELECT * FROM t1
    ->             WHERE   t1.b < 0
    ->                     OR
    ->                     t1.a IN (SELECT t2.a + 1 FROM t2)\G
*************************** 1. row ***************************
           id: 1
  select_type: PRIMARY
        table: t1
   partitions: NULL
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 3
     filtered: 100.00
        Extra: NULL
*************************** 2. row ***************************
           id: 1
  select_type: PRIMARY
        table: <derived2>
   partitions: NULL
         type: ref
possible_keys: <auto_key0>
          key: <auto_key0>
      key_len: 9
          ref: std2.t1.a
         rows: 2
     filtered: 100.00
        Extra: Using where; Using index
*************************** 3. row ***************************
           id: 2
  select_type: DERIVED
        table: t2
   partitions: NULL
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 6
     filtered: 100.00
        Extra: Using temporary
```

Checking and simplifying the result of `SHOW WARNINGS` after executing `EXPLAIN` on this query shows that, when the `subquery_to_derived` flag enabled, `SELECT * FROM t1 WHERE t1.b < 0 OR t1.a IN (SELECT t2.a + 1 FROM t2)` is rewritten in a form similar to what is shown here:

```
SELECT a, b FROM t1
    LEFT JOIN (SELECT DISTINCT a + 1 AS e FROM t2) d
    ON t1.a = d.e
    WHERE   t1.b < 0
            OR
            d.e IS NOT NULL;
```

Example, using a query with `EXISTS (subquery)` and the same tables and data as in the previous example:

```
mysql> SELECT * FROM t1
    ->     WHERE   t1.b < 0
    ->             OR
    ->             EXISTS(SELECT * FROM t2 WHERE t2.a = t1.a + 1);
+------+------+
| a    | b    |
+------+------+
```

```
|    1 |   10 |
|    2 |   20 |
+------+------+

mysql> SET @@optimizer_switch="subquery_to_derived=off";

mysql> EXPLAIN SELECT * FROM t1
    ->             WHERE   t1.b < 0
    ->                     OR
    ->                     EXISTS(SELECT * FROM t2 WHERE t2.a = t1.a + 1)\G
*************************** 1. row ***************************
           id: 1
  select_type: PRIMARY
        table: t1
   partitions: NULL
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 3
     filtered: 100.00
        Extra: Using where
*************************** 2. row ***************************
           id: 2
  select_type: DEPENDENT SUBQUERY
        table: t2
   partitions: NULL
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 6
     filtered: 16.67
        Extra: Using where

mysql> SET @@optimizer_switch="subquery_to_derived=on";

mysql> EXPLAIN SELECT * FROM t1
    ->             WHERE   t1.b < 0
    ->                     OR
    ->                     EXISTS(SELECT * FROM t2 WHERE t2.a = t1.a + 1)\G
*************************** 1. row ***************************
           id: 1
  select_type: PRIMARY
        table: t1
   partitions: NULL
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 3
     filtered: 100.00
        Extra: NULL
*************************** 2. row ***************************
           id: 1
  select_type: PRIMARY
        table: <derived2>
   partitions: NULL
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 6
     filtered: 100.00
        Extra: Using where; Using join buffer (hash join)
*************************** 3. row ***************************
           id: 2
  select_type: DERIVED
```

```
        table: t2
   partitions: NULL
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 6
     filtered: 100.00
        Extra: Using temporary
```

If we execute `SHOW WARNINGS` after running `EXPLAIN` on the query `SELECT * FROM t1 WHERE t1.b < 0 OR EXISTS(SELECT * FROM t2 WHERE t2.a = t1.a + 1)` when `subquery_to_derived` has been enabled, and simplify the second row of the result, we see that it has been rewritten in a form which resembles this:

```
SELECT a, b FROM t1
LEFT JOIN (SELECT DISTINCT 1 AS e1, t2.a AS e2 FROM t2) d
ON t1.a + 1 = d.e2
WHERE   t1.b < 0
        OR
        d.e1 IS NOT NULL;
```

For more information, see Section 8.2.2.4, “Optimizing Derived Tables, View References, and Common Table Expressions with Merging or Materialization”, as well as Section 8.2.1.19, “LIMIT Query Optimization”, and Section 8.2.2.1, “Optimizing IN and EXISTS Subquery Predicates with Semijoin Transformations”.

When you assign a value to `optimizer_switch`, flags that are not mentioned keep their current values. This makes it possible to enable or disable specific optimizer behaviors in a single statement without affecting other behaviors. The statement does not depend on what other optimizer flags exist and what their values are. Suppose that all Index Merge optimizations are enabled:

```
mysql> SELECT @@optimizer_switch\G
*************************** 1. row ***************************
@@optimizer_switch: index_merge=on,index_merge_union=on,
                    index_merge_sort_union=on,index_merge_intersection=on,
                    engine_condition_pushdown=on,index_condition_pushdown=on,
                    mrr=on,mrr_cost_based=on,block_nested_loop=on,
                    batched_key_access=off,materialization=on,semijoin=on,
                    loosescan=on, firstmatch=on,
                    subquery_materialization_cost_based=on,
                    use_index_extensions=on,condition_fanout_filter=on,
                    derived_merge=on,use_invisible_indexes=off,skip_scan=on,
                    hash_join=on,subquery_to_derived=off,
                    prefer_ordering_index=on
```

If the server is using the Index Merge Union or Index Merge Sort-Union access methods for certain queries and you want to check whether the optimizer can perform better without them, set the variable value like this:

```
mysql> SET optimizer_switch='index_merge_union=off,index_merge_sort_union=off';

mysql> SELECT @@optimizer_switch\G
*************************** 1. row ***************************
@@optimizer_switch: index_merge=on,index_merge_union=off,
                    index_merge_sort_union=off,index_merge_intersection=on,
                    engine_condition_pushdown=on,index_condition_pushdown=on,
                    mrr=on,mrr_cost_based=on,block_nested_loop=on,
                    batched_key_access=off,materialization=on,semijoin=on,
                    loosescan=on, firstmatch=on,
                    subquery_materialization_cost_based=on,
                    use_index_extensions=on,condition_fanout_filter=on,
                    derived_merge=on,use_invisible_indexes=off,skip_scan=on,
                    hash_join=on,subquery_to_derived=off,
                    prefer_ordering_index=on
```

8.9.3 Optimizer Hints

One means of control over optimizer strategies is to set the `optimizer_switch` system variable (see Section 8.9.2, “Switchable Optimizations”). Changes to this variable affect execution of all subsequent queries; to affect one query differently from another, it is necessary to change `optimizer_switch` before each one.

Another way to control the optimizer is by using optimizer hints, which can be specified within individual statements. Because optimizer hints apply on a per-statement basis, they provide finer control over statement execution plans than can be achieved using `optimizer_switch`. For example, you can enable an optimization for one table in a statement and disable the optimization for a different table. Hints within a statement take precedence over `optimizer_switch` flags.

Examples:

```
SELECT /*+ NO_RANGE_OPTIMIZATION(t3 PRIMARY, f2_idx) */ f1
  FROM t3 WHERE f1 > 30 AND f1 < 33;
SELECT /*+ BKA(t1) NO_BKA(t2) */ * FROM t1 INNER JOIN t2 WHERE ...;
SELECT /*+ NO_ICP(t1, t2) */ * FROM t1 INNER JOIN t2 WHERE ...;
SELECT /*+ SEMIJOIN(FIRSTMATCH, LOOSESCAN) */ * FROM t1 ...;
EXPLAIN SELECT /*+ NO_ICP(t1) */ * FROM t1 WHERE ...;
SELECT /*+ MERGE(dt) */ * FROM (SELECT * FROM t1) AS dt;
INSERT /*+ SET_VAR(foreign_key_checks=OFF) */ INTO t2 VALUES(2);
```

Optimizer hints, described here, differ from index hints, described in Section 8.9.4, “Index Hints”. Optimizer and index hints may be used separately or together.

- Optimizer Hint Overview
- Optimizer Hint Syntax
- Join-Order Optimizer Hints
- Table-Level Optimizer Hints
- Index-Level Optimizer Hints
- Subquery Optimizer Hints
- Statement Execution Time Optimizer Hints
- Variable-Setting Hint Syntax
- Resource Group Hint Syntax
- Optimizer Hints for Naming Query Blocks

Optimizer Hint Overview

Optimizer hints apply at different scope levels:

- Global: The hint affects the entire statement
- Query block: The hint affects a particular query block within a statement
- Table-level: The hint affects a particular table within a query block
- Index-level: The hint affects a particular index within a table

The following table summarizes the available optimizer hints, the optimizer strategies they affect, and the scope or scopes at which they apply. More details are given later.

Table 8.2 Optimizer Hints Available

Hint Name	Description	Applicable Scopes
`BKA`, `NO_BKA`	Affects Batched Key Access join processing	Query block, table

Hint Name	Description	Applicable Scopes
BNL, NO_BNL	Prior to MySQL 8.0.20: affects Block Nested-Loop join processing; MySQL 8.0.18 and later: also affects hash join optimization; MySQL 8.0.20 and later: affects hash join optimization only	Query block, table
DERIVED_CONDITION_PUSHDOWN, NO_DERIVED_CONDITION_PUSHDOWN	Use or ignore the derived condition pushdown optimization for materialized derived tables (Added in MySQL 8.0.22)	Query block, table
GROUP_INDEX, NO_GROUP_INDEX	Use or ignore the specified index or indexes for index scans in GROUP BY operations (Added in MySQL 8.0.20)	Index
HASH_JOIN, NO_HASH_JOIN	Affects Hash Join optimization (MySQL 8.0.18 only	Query block, table
INDEX, NO_INDEX	Acts as the combination of JOIN_INDEX, GROUP_INDEX, and ORDER_INDEX, or as the combination of NO_JOIN_INDEX, NO_GROUP_INDEX, and NO_ORDER_INDEX (Added in MySQL 8.0.20)	Index
INDEX_MERGE, NO_INDEX_MERGE	Affects Index Merge optimization	Table, index
JOIN_FIXED_ORDER	Use table order specified in FROM clause for join order	Query block
JOIN_INDEX, NO_JOIN_INDEX	Use or ignore the specified index or indexes for any access method (Added in MySQL 8.0.20)	Index
JOIN_ORDER	Use table order specified in hint for join order	Query block
JOIN_PREFIX	Use table order specified in hint for first tables of join order	Query block
JOIN_SUFFIX	Use table order specified in hint for last tables of join order	Query block
MAX_EXECUTION_TIME	Limits statement execution time	Global
MERGE, NO_MERGE	Affects derived table/view merging into outer query block	Table
MRR, NO_MRR	Affects Multi-Range Read optimization	Table, index
NO_ICP	Affects Index Condition Pushdown optimization	Table, index
NO_RANGE_OPTIMIZATION	Affects range optimization	Table, index
ORDER_INDEX, NO_ORDER_INDEX	Use or ignore the specified index or indexes for sorting rows (Added in MySQL 8.0.20)	Index

Hint Name	Description	Applicable Scopes
QB_NAME	Assigns name to query block	Query block
RESOURCE_GROUP	Set resource group during statement execution	Global
SEMIJOIN, NO_SEMIJOIN	Affects semijoin strategies; beginning with MySQL 8.0.17, this also applies to antijoins	Query block
SKIP_SCAN, NO_SKIP_SCAN	Affects Skip Scan optimization	Table, index
SET_VAR	Set variable during statement execution	Global
SUBQUERY	Affects materialization, IN-to-EXISTS subquery strategies	Query block

Disabling an optimization prevents the optimizer from using it. Enabling an optimization means the optimizer is free to use the strategy if it applies to statement execution, not that the optimizer necessarily uses it.

Optimizer Hint Syntax

MySQL supports comments in SQL statements as described in Section 9.7, “Comments”. Optimizer hints must be specified within `/*+ ... */` comments. That is, optimizer hints use a variant of `/* ... */` C-style comment syntax, with a `+` character following the `/*` comment opening sequence. Examples:

```
/*+ BKA(t1) */
/*+ BNL(t1, t2) */
/*+ NO_RANGE_OPTIMIZATION(t4 PRIMARY) */
/*+ QB_NAME(qb2) */
```

Whitespace is permitted after the `+` character.

The parser recognizes optimizer hint comments after the initial keyword of `SELECT`, `UPDATE`, `INSERT`, `REPLACE`, and `DELETE` statements. Hints are permitted in these contexts:

- At the beginning of query and data change statements:

  ```
  SELECT /*+ ... */ ...
  INSERT /*+ ... */ ...
  REPLACE /*+ ... */ ...
  UPDATE /*+ ... */ ...
  DELETE /*+ ... */ ...
  ```

- At the beginning of query blocks:

  ```
  (SELECT /*+ ... */ ... )
  (SELECT ... ) UNION (SELECT /*+ ... */ ... )
  (SELECT /*+ ... */ ... ) UNION (SELECT /*+ ... */ ... )
  UPDATE ... WHERE x IN (SELECT /*+ ... */ ...)
  INSERT ... SELECT /*+ ... */ ...
  ```

- In hintable statements prefaced by `EXPLAIN`. For example:

  ```
  EXPLAIN SELECT /*+ ... */ ...
  EXPLAIN UPDATE ... WHERE x IN (SELECT /*+ ... */ ...)
  ```

  The implication is that you can use `EXPLAIN` to see how optimizer hints affect execution plans. Use `SHOW WARNINGS` immediately after `EXPLAIN` to see how hints are used. The extended `EXPLAIN` output displayed by a following `SHOW WARNINGS` indicates which hints were used. Ignored hints are not displayed.

A hint comment may contain multiple hints, but a query block cannot contain multiple hint comments. This is valid:

```
SELECT /*+ BNL(t1) BKA(t2) */ ...
```

But this is invalid:

```
SELECT /*+ BNL(t1) */ /* BKA(t2) */ ...
```

When a hint comment contains multiple hints, the possibility of duplicates and conflicts exists. The following general guidelines apply. For specific hint types, additional rules may apply, as indicated in the hint descriptions.

- Duplicate hints: For a hint such as `/*+ MRR(idx1) MRR(idx1) */`, MySQL uses the first hint and issues a warning about the duplicate hint.
- Conflicting hints: For a hint such as `/*+ MRR(idx1) NO_MRR(idx1) */`, MySQL uses the first hint and issues a warning about the second conflicting hint.

Query block names are identifiers and follow the usual rules about what names are valid and how to quote them (see Section 9.2, “Schema Object Names”).

Hint names, query block names, and strategy names are not case-sensitive. References to table and index names follow the usual identifier case-sensitivity rules (see Section 9.2.3, “Identifier Case Sensitivity”).

Join-Order Optimizer Hints

Join-order hints affect the order in which the optimizer joins tables.

Syntax of the `JOIN_FIXED_ORDER` hint:

```
hint_name([@query_block_name])
```

Syntax of other join-order hints:

```
hint_name([@query_block_name] tbl_name [, tbl_name] ...)
hint_name(tbl_name[@query_block_name] [, tbl_name[@query_block_name]] ...)
```

The syntax refers to these terms:

- *`hint_name`*: These hint names are permitted:
  - `JOIN_FIXED_ORDER`: Force the optimizer to join tables using the order in which they appear in the `FROM` clause. This is the same as specifying `SELECT STRAIGHT_JOIN`.
  - `JOIN_ORDER`: Instruct the optimizer to join tables using the specified table order. The hint applies to the named tables. The optimizer may place tables that are not named anywhere in the join order, including between specified tables.
  - `JOIN_PREFIX`: Instruct the optimizer to join tables using the specified table order for the first tables of the join execution plan. The hint applies to the named tables. The optimizer places all other tables after the named tables.
  - `JOIN_SUFFIX`: Instruct the optimizer to join tables using the specified table order for the last tables of the join execution plan. The hint applies to the named tables. The optimizer places all other tables before the named tables.
- *`tbl_name`*: The name of a table used in the statement. A hint that names tables applies to all tables that it names. The `JOIN_FIXED_ORDER` hint names no tables and applies to all tables in the `FROM` clause of the query block in which it occurs.

  If a table has an alias, hints must refer to the alias, not the table name.

  Table names in hints cannot be qualified with schema names.
- *`query_block_name`*: The query block to which the hint applies. If the hint includes no leading *`@query_block_name`*, the hint applies to the query block in which it occurs. For

tbl_name@query_block_name syntax, the hint applies to the named table in the named query block. To assign a name to a query block, see Optimizer Hints for Naming Query Blocks.

Example:

```
SELECT
/*+ JOIN_PREFIX(t2, t5@subq2, t4@subq1)
    JOIN_ORDER(t4@subq1, t3)
    JOIN_SUFFIX(t1) */
COUNT(*) FROM t1 JOIN t2 JOIN t3
           WHERE t1.f1 IN (SELECT /*+ QB_NAME(subq1) */ f1 FROM t4)
             AND t2.f1 IN (SELECT /*+ QB_NAME(subq2) */ f1 FROM t5);
```

Hints control the behavior of semijoin tables that are merged to the outer query block. If subqueries `subq1` and `subq2` are converted to semijoins, tables `t4@subq1` and `t5@subq2` are merged to the outer query block. In this case, the hint specified in the outer query block controls the behavior of `t4@subq1`, `t5@subq2` tables.

The optimizer resolves join-order hints according to these principles:

- Multiple hint instances

  Only one `JOIN_PREFIX` and `JOIN_SUFFIX` hint of each type are applied. Any later hints of the same type are ignored with a warning. `JOIN_ORDER` can be specified several times.

  Examples:

  ```
  /*+ JOIN_PREFIX(t1) JOIN_PREFIX(t2) */
  ```

  The second `JOIN_PREFIX` hint is ignored with a warning.

  ```
  /*+ JOIN_PREFIX(t1) JOIN_SUFFIX(t2) */
  ```

  Both hints are applicable. No warning occurs.

  ```
  /*+ JOIN_ORDER(t1, t2) JOIN_ORDER(t2, t3) */
  ```

  Both hints are applicable. No warning occurs.

- Conflicting hints

  In some cases hints can conflict, such as when `JOIN_ORDER` and `JOIN_PREFIX` have table orders that are impossible to apply at the same time:

  ```
  SELECT /*+ JOIN_ORDER(t1, t2) JOIN_PREFIX(t2, t1) */ ... FROM t1, t2;
  ```

  In this case, the first specified hint is applied and subsequent conflicting hints are ignored with no warning. A valid hint that is impossible to apply is silently ignored with no warning.

- Ignored hints

  A hint is ignored if a table specified in the hint has a circular dependency.

  Example:

  ```
  /*+ JOIN_ORDER(t1, t2) JOIN_PREFIX(t2, t1) */
  ```

  The `JOIN_ORDER` hint sets table `t2` dependent on `t1`. The `JOIN_PREFIX` hint is ignored because table `t1` cannot be dependent on `t2`. Ignored hints are not displayed in extended `EXPLAIN` output.

- Interaction with `const` tables

  The MySQL optimizer places `const` tables first in the join order, and the position of a `const` table cannot be affected by hints. References to `const` tables in join-order hints are ignored, although the hint is still applicable. For example, these are equivalent:

```
JOIN_ORDER(t1, const_tbl, t2)
JOIN_ORDER(t1, t2)
```

Accepted hints shown in extended `EXPLAIN` output include `const` tables as they were specified.

- Interaction with types of join operations

  MySQL supports several type of joins: `LEFT`, `RIGHT`, `INNER`, `CROSS`, `STRAIGHT_JOIN`. A hint that conflicts with the specified type of join is ignored with no warning.

  Example:

```
SELECT /*+ JOIN_PREFIX(t1, t2) */FROM t2 LEFT JOIN t1;
```

  Here a conflict occurs between the requested join order in the hint and the order required by the `LEFT JOIN`. The hint is ignored with no warning.

Table-Level Optimizer Hints

Table-level hints affect:

- Use of the Block Nested-Loop (BNL) and Batched Key Access (BKA) join-processing algorithms (see Section 8.2.1.12, “Block Nested-Loop and Batched Key Access Joins”).
- Whether derived tables, view references, or common table expressions should be merged into the outer query block, or materialized using an internal temporary table.
- Use of the derived table condition pushdown optimization (added in MySQL 8.0.22). See Section 8.2.2.5, “Derived Condition Pushdown Optimization”.

These hint types apply to specific tables, or all tables in a query block.

Syntax of table-level hints:

```
hint_name([@query_block_name] [tbl_name [, tbl_name] ...])
hint_name([tbl_name@query_block_name [, tbl_name@query_block_name] ...])
```

The syntax refers to these terms:

- *`hint_name`*: These hint names are permitted:
  - `BKA`, `NO_BKA`: Enable or disable batched key access for the specified tables.
  - `BNL`, `NO_BNL`: Enable or disable block nested loop for the specified tables. In MySQL 8.0.18 and later, these hints also enable and disable the hash join optimization.

    

    Note

    The block-nested loop optimization is removed in MySQL 8.0.20 and later releases, but `BNL` and `NO_BNL` continue to be supported for enabling and disabling hash joins.

  - `DERIVED_CONDITION_PUSHDOWN`, `NO_DERIVED_CONDITION_PUSHDOWN`: Enable or disable use of derived table condition pushdown for the specified tables (added in MySQL 8.0.22). For more information, see Section 8.2.2.5, “Derived Condition Pushdown Optimization”.
  - `HASH_JOIN`, `NO_HASH_JOIN`: In MySQL 8.0.18 only, enable or disable use of a hash join for the specified tables. These hints have no effect in MySQL 8.0.19 or later, where you should use `BNL` or `NO_BNL` instead.
  - `MERGE`, `NO_MERGE`: Enable merging for the specified tables, view references or common table expressions; or disable merging and use materialization instead.

Note

To use a block nested loop or batched key access hint to enable join buffering for any inner table of an outer join, join buffering must be enabled for all inner tables of the outer join.

- *`tbl_name`*: The name of a table used in the statement. The hint applies to all tables that it names. If the hint names no tables, it applies to all tables of the query block in which it occurs.

  If a table has an alias, hints must refer to the alias, not the table name.

  Table names in hints cannot be qualified with schema names.

- *`query_block_name`*: The query block to which the hint applies. If the hint includes no leading *`@query_block_name`*, the hint applies to the query block in which it occurs. For *`tbl_name@query_block_name`* syntax, the hint applies to the named table in the named query block. To assign a name to a query block, see Optimizer Hints for Naming Query Blocks.

Examples:

```
SELECT /*+ NO_BKA(t1, t2) */ t1.* FROM t1 INNER JOIN t2 INNER JOIN t3;
SELECT /*+ NO_BNL() BKA(t1) */ t1.* FROM t1 INNER JOIN t2 INNER JOIN t3;
SELECT /*+ NO_MERGE(dt) */ * FROM (SELECT * FROM t1) AS dt;
```

A table-level hint applies to tables that receive records from previous tables, not sender tables. Consider this statement:

```
SELECT /*+ BNL(t2) */ FROM t1, t2;
```

If the optimizer chooses to process `t1` first, it applies a Block Nested-Loop join to `t2` by buffering the rows from `t1` before starting to read from `t2`. If the optimizer instead chooses to process `t2` first, the hint has no effect because `t2` is a sender table.

For the `MERGE` and `NO_MERGE` hints, these precedence rules apply:

- A hint takes precedence over any optimizer heuristic that is not a technical constraint. (If providing a hint as a suggestion has no effect, the optimizer has a reason for ignoring it.)
- A hint takes precedence over the `derived_merge` flag of the `optimizer_switch` system variable.
- For view references, an `ALGORITHM={MERGE|TEMPTABLE}` clause in the view definition takes precedence over a hint specified in the query referencing the view.

Index-Level Optimizer Hints

Index-level hints affect which index-processing strategies the optimizer uses for particular tables or indexes. These hint types affect use of Index Condition Pushdown (ICP), Multi-Range Read (MRR), Index Merge, and range optimizations (see Section 8.2.1, “Optimizing SELECT Statements”).

Syntax of index-level hints:

```
hint_name([@query_block_name] tbl_name [index_name [, index_name] ...])
hint_name(tbl_name@query_block_name [index_name [, index_name] ...])
```

The syntax refers to these terms:

- *`hint_name`*: These hint names are permitted:

  - `GROUP_INDEX`, `NO_GROUP_INDEX`: Enable or disable the specified index or indexes for index scans for `GROUP BY` operations. Equivalent to the index hints `FORCE INDEX FOR GROUP BY`, `IGNORE INDEX FOR GROUP BY`. Available in MySQL 8.0.20 and later.

- `INDEX`, `NO_INDEX`: Acts as the combination of `JOIN_INDEX`, `GROUP_INDEX`, and `ORDER_INDEX`, forcing the server to use the specified index or indexes for any and all scopes, or as the combination of `NO_JOIN_INDEX`, `NO_GROUP_INDEX`, and `NO_ORDER_INDEX`, which causes the server to ignore the specified index or indexes for any and all scopes. Equivalent to `FORCE INDEX`, `IGNORE INDEX`. Available beginning with MySQL 8.0.20.

- `INDEX_MERGE`, `NO_INDEX_MERGE`: Enable or disable the Index Merge access method for the specified table or indexes. For information about this access method, see Section 8.2.1.3, “Index Merge Optimization”. These hints apply to all three Index Merge algorithms.

  The `INDEX_MERGE` hint forces the optimizer to use Index Merge for the specified table using the specified set of indexes. If no index is specified, the optimizer considers all possible index combinations and selects the least expensive one. The hint may be ignored if the index combination is inapplicable to the given statement.

  The `NO_INDEX_MERGE` hint disables Index Merge combinations that involve any of the specified indexes. If the hint specifies no indexes, Index Merge is not permitted for the table.

- `JOIN_INDEX`, `NO_JOIN_INDEX`: Forces MySQL to use or ignore the specified index or indexes for any access method, such as `ref`, `range`, `index_merge`, and so on. Equivalent to `FORCE INDEX FOR JOIN`, `IGNORE INDEX FOR JOIN`. Available in MySQL 8.0.20 and later.

- `MRR`, `NO_MRR`: Enable or disable MRR for the specified table or indexes. MRR hints apply only to `InnoDB` and `MyISAM` tables. For information about this access method, see Section 8.2.1.11, “Multi-Range Read Optimization”.

- `NO_ICP`: Disable ICP for the specified table or indexes. By default, ICP is a candidate optimization strategy, so there is no hint for enabling it. For information about this access method, see Section 8.2.1.6, “Index Condition Pushdown Optimization”.

- `NO_RANGE_OPTIMIZATION`: Disable index range access for the specified table or indexes. This hint also disables Index Merge and Loose Index Scan for the table or indexes. By default, range access is a candidate optimization strategy, so there is no hint for enabling it.

  This hint may be useful when the number of ranges may be high and range optimization would require many resources.

- `ORDER_INDEX`, `NO_ORDER_INDEX`: Cause MySQL to use or to ignore the specified index or indexes for sorting rows. Equivalent to `FORCE INDEX FOR ORDER BY`, `IGNORE INDEX FOR ORDER BY`. Available beginning with MySQL 8.0.20.

- `SKIP_SCAN`, `NO_SKIP_SCAN`: Enable or disable the Skip Scan access method for the specified table or indexes. For information about this access method, see Skip Scan Range Access Method. These hints are available as of MySQL 8.0.13.

  The `SKIP_SCAN` hint forces the optimizer to use Skip Scan for the specified table using the specified set of indexes. If no index is specified, the optimizer considers all possible indexes and selects the least expensive one. The hint may be ignored if the index is inapplicable to the given statement.

  The `NO_SKIP_SCAN` hint disables Skip Scan for the specified indexes. If the hint specifies no indexes, Skip Scan is not permitted for the table.

- *`tbl_name`*: The table to which the hint applies.

- *`index_name`*: The name of an index in the named table. The hint applies to all indexes that it names. If the hint names no indexes, it applies to all indexes in the table.

  To refer to a primary key, use the name `PRIMARY`. To see the index names for a table, use `SHOW INDEX`.

- *query_block_name*: The query block to which the hint applies. If the hint includes no leading *@query_block_name*, the hint applies to the query block in which it occurs. For *tbl_name@query_block_name* syntax, the hint applies to the named table in the named query block. To assign a name to a query block, see Optimizer Hints for Naming Query Blocks.

Examples:

```
SELECT /*+ INDEX_MERGE(t1 f3, PRIMARY) */ f2 FROM t1
  WHERE f1 = 'o' AND f2 = f3 AND f3 <= 4;
SELECT /*+ MRR(t1) */ * FROM t1 WHERE f2 <= 3 AND 3 <= f3;
SELECT /*+ NO_RANGE_OPTIMIZATION(t3 PRIMARY, f2_idx) */ f1
  FROM t3 WHERE f1 > 30 AND f1 < 33;
INSERT INTO t3(f1, f2, f3)
  (SELECT /*+ NO_ICP(t2) */ t2.f1, t2.f2, t2.f3 FROM t1,t2
   WHERE t1.f1=t2.f1 AND t2.f2 BETWEEN t1.f1
   AND t1.f2 AND t2.f2 + 1 >= t1.f1 + 1);
SELECT /*+ SKIP_SCAN(t1 PRIMARY) */ f1, f2
  FROM t1 WHERE f2 > 40;
```

The following examples use the Index Merge hints, but other index-level hints follow the same principles regarding hint ignoring and precedence of optimizer hints in relation to the `optimizer_switch` system variable or index hints.

Assume that table `t1` has columns `a`, `b`, `c`, and `d`; and that indexes named `i_a`, `i_b`, and `i_c` exist on `a`, `b`, and `c`, respectively:

```
SELECT /*+ INDEX_MERGE(t1 i_a, i_b, i_c)*/ * FROM t1
  WHERE a = 1 AND b = 2 AND c = 3 AND d = 4;
```

Index Merge is used for `(i_a, i_b, i_c)` in this case.

```
SELECT /*+ INDEX_MERGE(t1 i_a, i_b, i_c)*/ * FROM t1
  WHERE b = 1 AND c = 2 AND d = 3;
```

Index Merge is used for `(i_b, i_c)` in this case.

```
/*+ INDEX_MERGE(t1 i_a, i_b) NO_INDEX_MERGE(t1 i_b) */
```

`NO_INDEX_MERGE` is ignored because there is a preceding hint for the same table.

```
/*+ NO_INDEX_MERGE(t1 i_a, i_b) INDEX_MERGE(t1 i_b) */
```

`INDEX_MERGE` is ignored because there is a preceding hint for the same table.

For the `INDEX_MERGE` and `NO_INDEX_MERGE` optimizer hints, these precedence rules apply:

- If an optimizer hint is specified and is applicable, it takes precedence over the Index Merge-related flags of the `optimizer_switch` system variable.

  ```
  SET optimizer_switch='index_merge_intersection=off';
  SELECT /*+ INDEX_MERGE(t1 i_b, i_c) */ * FROM t1
  WHERE b = 1 AND c = 2 AND d = 3;
  ```

  The hint takes precedence over `optimizer_switch`. Index Merge is used for `(i_b, i_c)` in this case.

  ```
  SET optimizer_switch='index_merge_intersection=on';
  SELECT /*+ INDEX_MERGE(t1 i_b) */ * FROM t1
  WHERE b = 1 AND c = 2 AND d = 3;
  ```

  The hint specifies only one index, so it is inapplicable, and the `optimizer_switch` flag (`on`) applies. Index Merge is used if the optimizer assesses it to be cost efficient.

  ```
  SET optimizer_switch='index_merge_intersection=off';
  SELECT /*+ INDEX_MERGE(t1 i_b) */ * FROM t1
  WHERE b = 1 AND c = 2 AND d = 3;
  ```

The hint specifies only one index, so it is inapplicable, and the `optimizer_switch` flag (`off`) applies. Index Merge is not used.

- The index-level optimizer hints `GROUP_INDEX`, `INDEX`, `JOIN_INDEX`, and `ORDER_INDEX` all take precedence over the equivalent `FORCE INDEX` hints; that is, they cause the `FORCE INDEX` hints to be ignored. Likewise, the `NO_GROUP_INDEX`, `NO_INDEX`, `NO_JOIN_INDEX`, and `NO_ORDER_INDEX` hints all take precedence over any `IGNORE INDEX` equivalents, also causing them to be ignored.

  The index-level optimizer hints `GROUP_INDEX`, `NO_GROUP_INDEX`, `INDEX`,`NO_INDEX`, `JOIN_INDEX`,`NO_JOIN_INDEX`, `ORDER_INDEX`, and `NO_ORDER_INDEX` hints all take precedence over all other optimizer hints, including other index-level optimizer hints. Any other optimizer hints are applied only to the indexes permitted by these.

  The `GROUP_INDEX`, `INDEX`, `JOIN_INDEX`, and `ORDER_INDEX` hints are all equivalent to `FORCE INDEX` and not to `USE INDEX`. This is because using one or more of these hints means that a table scan is used only if there is no way to use one of the named indexes to find rows in the table. To cause MySQL to use the same index or set of indexes as with a given instance of `USE INDEX`, you can use `NO_INDEX`, `NO_JOIN_INDEX`, `NO_GROUP_INDEX`, `NO_ORDER_INDEX`, or some combination of these.

  To replicate the effect that `USE INDEX` has in the query `SELECT a,c FROM t1 USE INDEX FOR ORDER BY (i_a) ORDER BY a`, you can use the `NO_ORDER_INDEX` optimizer hint to cover all indexes on the table except the one that is desired like this:

```
SELECT /*+ NO_ORDER_INDEX(t1 i_b,i_c) */ a,c
    FROM t1
    ORDER BY a;
```

  Attempting to combine `NO_ORDER_INDEX` for the table as a whole with `USE INDEX FOR ORDER BY` does not work to do this, because `NO_ORDER_BY` causes `USE INDEX` to be ignored, as shown here:

```
mysql> EXPLAIN SELECT /*+ NO_ORDER_INDEX(t1) */ a,c FROM t1
    ->     USE INDEX FOR ORDER BY (i_a) ORDER BY a\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: t1
   partitions: NULL
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 256
     filtered: 100.00
        Extra: Using filesort
```

- The `USE INDEX`, `FORCE INDEX`, and `IGNORE INDEX` index hints have higher priority than the `INDEX_MERGE` and `NO_INDEX_MERGE` optimizer hints.

```
/*+ INDEX_MERGE(t1 i_a, i_b, i_c) */ ... IGNORE INDEX i_a
```

  `IGNORE INDEX` takes precedence over `INDEX_MERGE`, so index `i_a` is excluded from the possible ranges for Index Merge.

```
/*+ NO_INDEX_MERGE(t1 i_a, i_b) */ ... FORCE INDEX i_a, i_b
```

  Index Merge is disallowed for `i_a, i_b` because of `FORCE INDEX`, but the optimizer is forced to use either `i_a` or `i_b` for `range` or `ref` access. There are no conflicts; both hints are applicable.

- If an `IGNORE INDEX` hint names multiple indexes, those indexes are unavailable for Index Merge.

- The `FORCE INDEX` and `USE INDEX` hints make only the named indexes to be available for Index Merge.

```
SELECT /*+ INDEX_MERGE(t1 i_a, i_b, i_c) */ a FROM t1
FORCE INDEX (i_a, i_b) WHERE c = 'h' AND a = 2 AND b = 'b';
```

  The Index Merge intersection access algorithm is used for `(i_a, i_b)`. The same is true if `FORCE INDEX` is changed to `USE INDEX`.

Subquery Optimizer Hints

Subquery hints affect whether to use semijoin transformations and which semijoin strategies to permit, and, when semijoins are not used, whether to use subquery materialization or `IN`-to-`EXISTS` transformations. For more information about these optimizations, see Section 8.2.2, “Optimizing Subqueries, Derived Tables, View References, and Common Table Expressions”.

Syntax of hints that affect semijoin strategies:

```
hint_name([@query_block_name] [strategy [, strategy] ...])
```

The syntax refers to these terms:

- *`hint_name`*: These hint names are permitted:
  - `SEMIJOIN`, `NO_SEMIJOIN`: Enable or disable the named semijoin strategies.
- *`strategy`*: A semijoin strategy to be enabled or disabled. These strategy names are permitted: `DUPSWEEDOUT`, `FIRSTMATCH`, `LOOSESCAN`, `MATERIALIZATION`.

  For `SEMIJOIN` hints, if no strategies are named, semijoin is used if possible based on the strategies enabled according to the `optimizer_switch` system variable. If strategies are named but inapplicable for the statement, `DUPSWEEDOUT` is used.

  For `NO_SEMIJOIN` hints, if no strategies are named, semijoin is not used. If strategies are named that rule out all applicable strategies for the statement, `DUPSWEEDOUT` is used.

If one subquery is nested within another and both are merged into a semijoin of an outer query, any specification of semijoin strategies for the innermost query are ignored. `SEMIJOIN` and `NO_SEMIJOIN` hints can still be used to enable or disable semijoin transformations for such nested subqueries.

If `DUPSWEEDOUT` is disabled, on occasion the optimizer may generate a query plan that is far from optimal. This occurs due to heuristic pruning during greedy search, which can be avoided by setting `optimizer_prune_level=0`.

Examples:

```
SELECT /*+ NO_SEMIJOIN(@subq1 FIRSTMATCH, LOOSESCAN) */ * FROM t2
  WHERE t2.a IN (SELECT /*+ QB_NAME(subq1) */ a FROM t3);
SELECT /*+ SEMIJOIN(@subq1 MATERIALIZATION, DUPSWEEDOUT) */ * FROM t2
  WHERE t2.a IN (SELECT /*+ QB_NAME(subq1) */ a FROM t3);
```

Syntax of hints that affect whether to use subquery materialization or `IN`-to-`EXISTS` transformations:

```
SUBQUERY([@query_block_name] strategy)
```

The hint name is always `SUBQUERY`.

For `SUBQUERY` hints, these *`strategy`* values are permitted: `INTOEXISTS`, `MATERIALIZATION`.

Examples:

```
SELECT id, a IN (SELECT /*+ SUBQUERY(MATERIALIZATION) */ a FROM t1) FROM t2;
SELECT * FROM t2 WHERE t2.a IN (SELECT /*+ SUBQUERY(INTOEXISTS) */ a FROM t1);
```

For semijoin and `SUBQUERY` hints, a leading *`@query_block_name`* specifies the query block to which the hint applies. If the hint includes no leading *`@query_block_name`*, the hint applies to the query

block in which it occurs. To assign a name to a query block, see Optimizer Hints for Naming Query Blocks.

If a hint comment contains multiple subquery hints, the first is used. If there are other following hints of that type, they produce a warning. Following hints of other types are silently ignored.

Statement Execution Time Optimizer Hints

The `MAX_EXECUTION_TIME` hint is permitted only for `SELECT` statements. It places a limit *`N`* (a timeout value in milliseconds) on how long a statement is permitted to execute before the server terminates it:

```
MAX_EXECUTION_TIME(N)
```

Example with a timeout of 1 second (1000 milliseconds):

```
SELECT /*+ MAX_EXECUTION_TIME(1000) */ * FROM t1 INNER JOIN t2 WHERE ...
```

The `MAX_EXECUTION_TIME(N)` hint sets a statement execution timeout of *`N`* milliseconds. If this option is absent or *`N`* is 0, the statement timeout established by the `max_execution_time` system variable applies.

The `MAX_EXECUTION_TIME` hint is applicable as follows:

- For statements with multiple `SELECT` keywords, such as unions or statements with subqueries, `MAX_EXECUTION_TIME` applies to the entire statement and must appear after the first `SELECT`.
- It applies to read-only `SELECT` statements. Statements that are not read only are those that invoke a stored function that modifies data as a side effect.
- It does not apply to `SELECT` statements in stored programs and is ignored.

Variable-Setting Hint Syntax

The `SET_VAR` hint sets the session value of a system variable temporarily (for the duration of a single statement). Examples:

```
SELECT /*+ SET_VAR(sort_buffer_size = 16M) */ name FROM people ORDER BY name;
INSERT /*+ SET_VAR(foreign_key_checks=OFF) */ INTO t2 VALUES(2);
SELECT /*+ SET_VAR(optimizer_switch = 'mrr_cost_based=off') */ 1;
```

Syntax of the `SET_VAR` hint:

```
SET_VAR(var_name = value)
```

`var_name` names a system variable that has a session value (although not all such variables can be named, as explained later). *`value`* is the value to assign to the variable; the value must be a scalar.

`SET_VAR` makes a temporary variable change, as demonstrated by these statements:

```
mysql> SELECT @@unique_checks;
+-----------------+
| @@unique_checks |
+-----------------+
|               1 |
+-----------------+
mysql> SELECT /*+ SET_VAR(unique_checks=OFF) */ @@unique_checks;
+-----------------+
| @@unique_checks |
+-----------------+
|               0 |
+-----------------+
mysql> SELECT @@unique_checks;
+-----------------+
| @@unique_checks |
+-----------------+
|               1 |
+-----------------+
```

With SET_VAR, there is no need to save and restore the variable value. This enables you to replace multiple statements by a single statement. Consider this sequence of statements:

```
SET @saved_val = @@SESSION.var_name;
SET @@SESSION.var_name = value;
SELECT ...
SET @@SESSION.var_name = @saved_val;
```

The sequence can be replaced by this single statement:

```
SELECT /*+ SET_VAR(var_name = value) ...
```

Standalone SET statements permit any of these syntaxes for naming session variables:

```
SET SESSION var_name = value;
SET @@SESSION.var_name = value;
SET @@.var_name = value;
```

Because the SET_VAR hint applies only to session variables, session scope is implicit, and SESSION, @@SESSION., and @@ are neither needed nor permitted. Including explicit session-indicator syntax results in the SET_VAR hint being ignored with a warning.

Not all session variables are permitted for use with SET_VAR. Individual system variable descriptions indicate whether each variable is hintable; see Section 5.1.8, “Server System Variables”. You can also check a system variable at runtime by attempting to use it with SET_VAR. If the variable is not hintable, a warning occurs:

```
mysql> SELECT /*+ SET_VAR(collation_server = 'utf8mb4') */ 1;
+---+
| 1 |
+---+
| 1 |
+---+
1 row in set, 1 warning (0.00 sec)

mysql> SHOW WARNINGS\G
*************************** 1. row ***************************
  Level: Warning
   Code: 4537
Message: Variable 'collation_server' cannot be set using SET_VAR hint.
```

SET_VAR syntax permits setting only a single variable, but multiple hints can be given to set multiple variables:

```
SELECT /*+ SET_VAR(optimizer_switch = 'mrr_cost_based=off')
           SET_VAR(max_heap_table_size = 1G) */ 1;
```

If several hints with the same variable name appear in the same statement, the first one is applied and the others are ignored with a warning:

```
SELECT /*+ SET_VAR(max_heap_table_size = 1G)
           SET_VAR(max_heap_table_size = 3G) */ 1;
```

In this case, the second hint is ignored with a warning that it is conflicting.

A SET_VAR hint is ignored with a warning if no system variable has the specified name or the variable value is incorrect:

```
SELECT /*+ SET_VAR(max_size = 1G) */ 1;
SELECT /*+ SET_VAR(optimizer_switch = 'mrr_cost_based=yes') */ 1;
```

For the first statement, there is no max_size variable. For the second statement, mrr_cost_based takes values of on or off, so attempting to set it to yes is incorrect. In each case, the hint is ignored with a warning.

The SET_VAR hint is permitted only at the statement level. If used in a subquery, the hint is ignored with a warning.

Replicas ignore SET_VAR hints in replicated statements to avoid the potential for security issues.

Resource Group Hint Syntax

The RESOURCE_GROUP optimizer hint is used for resource group management (see Section 5.1.16, “Resource Groups”). This hint assigns the thread that executes a statement to the named resource group temporarily (for the duration of the statement). It requires the RESOURCE_GROUP_ADMIN or RESOURCE_GROUP_USER privilege.

Examples:

```
SELECT /*+ RESOURCE_GROUP(USR_default) */ name FROM people ORDER BY name;
INSERT /*+ RESOURCE_GROUP(Batch) */ INTO t2 VALUES(2);
```

Syntax of the RESOURCE_GROUP hint:

```
RESOURCE_GROUP(group_name)
```

group_name indicates the resource group to which the thread should be assigned for the duration of statement execution. If the group is nonexistent, a warning occurs and the hint is ignored.

The RESOURCE_GROUP hint must appear after the initial statement keyword (SELECT, INSERT, REPLACE, UPDATE, or DELETE).

An alternative to RESOURCE_GROUP is the SET RESOURCE GROUP statement, which nontemporarily assigns threads to a resource group. See Section 13.7.2.4, “SET RESOURCE GROUP Statement”.

Optimizer Hints for Naming Query Blocks

Table-level, index-level, and subquery optimizer hints permit specific query blocks to be named as part of their argument syntax. To create these names, use the QB_NAME hint, which assigns a name to the query block in which it occurs:

```
QB_NAME(name)
```

QB_NAME hints can be used to make explicit in a clear way which query blocks other hints apply to. They also permit all non-query block name hints to be specified within a single hint comment for easier understanding of complex statements. Consider the following statement:

```
SELECT ...
  FROM (SELECT ...
  FROM (SELECT ... FROM ...)) ...
```

QB_NAME hints assign names to query blocks in the statement:

```
SELECT /*+ QB_NAME(qb1) */ ...
  FROM (SELECT /*+ QB_NAME(qb2) */ ...
  FROM (SELECT /*+ QB_NAME(qb3) */ ... FROM ...)) ...
```

Then other hints can use those names to refer to the appropriate query blocks:

```
SELECT /*+ QB_NAME(qb1) MRR(@qb1 t1) BKA(@qb2) NO_MRR(@qb3t1 idx1, id2) */ ...
  FROM (SELECT /*+ QB_NAME(qb2) */ ...
  FROM (SELECT /*+ QB_NAME(qb3) */ ... FROM ...)) ...
```

The resulting effect is as follows:

- MRR(@qb1 t1) applies to table t1 in query block qb1.
- BKA(@qb2) applies to query block qb2.
- NO_MRR(@qb3 t1 idx1, id2) applies to indexes idx1 and idx2 in table t1 in query block qb3.

Query block names are identifiers and follow the usual rules about what names are valid and how to quote them (see Section 9.2, “Schema Object Names”). For example, a query block name that contains spaces must be quoted, which can be done using backticks:

```
SELECT /*+ BKA(@`my hint name`) */ ...
  FROM (SELECT /*+ QB_NAME(`my hint name`) */ ...) ...
```

If the ANSI_QUOTES SQL mode is enabled, it is also possible to quote query block names within double quotation marks:

```
SELECT /*+ BKA(@"my hint name") */ ...
  FROM (SELECT /*+ QB_NAME("my hint name") */ ...) ...
```

8.9.4 Index Hints

Index hints give the optimizer information about how to choose indexes during query processing. Index hints, described here, differ from optimizer hints, described in Section 8.9.3, “Optimizer Hints”. Index and optimizer hints may be used separately or together.

Index hints apply to SELECT and UPDATE statements. They also work with multi-table DELETE statements, but not with single-table DELETE, as shown later in this section.

Index hints are specified following a table name. (For the general syntax for specifying tables in a SELECT statement, see Section 13.2.13.2, “JOIN Clause”.) The syntax for referring to an individual table, including index hints, looks like this:

```
tbl_name [[AS] alias] [index_hint_list]

index_hint_list:
    index_hint [index_hint] ...

index_hint:
    USE {INDEX|KEY}
      [FOR {JOIN|ORDER BY|GROUP BY}] ([index_list])
  | {IGNORE|FORCE} {INDEX|KEY}
      [FOR {JOIN|ORDER BY|GROUP BY}] (index_list)

index_list:
    index_name [, index_name] ...
```

The USE INDEX (*index_list*) hint tells MySQL to use only one of the named indexes to find rows in the table. The alternative syntax IGNORE INDEX (*index_list*) tells MySQL to not use some particular index or indexes. These hints are useful if EXPLAIN shows that MySQL is using the wrong index from the list of possible indexes.

The FORCE INDEX hint acts like USE INDEX (*index_list*), with the addition that a table scan is assumed to be *very* expensive. In other words, a table scan is used only if there is no way to use one of the named indexes to find rows in the table.

Note

As of MySQL 8.0.20, the server supports the index-level optimizer hints JOIN_INDEX, GROUP_INDEX, ORDER_INDEX, and INDEX, which are equivalent to and intended to supersede FORCE INDEX index hints, as well as the NO_JOIN_INDEX, NO_GROUP_INDEX, NO_ORDER_INDEX, and NO_INDEX optimizer hints, which are equivalent to and intended to supersede IGNORE INDEX index hints. Thus, you should expect USE INDEX, FORCE INDEX, and IGNORE INDEX to be deprecated in a future release of MySQL, and at some time thereafter to be removed altogether.

These index-level optimizer hints are supported with both single-table and multi-table DELETE statements.

For more information, see Index-Level Optimizer Hints.

Each hint requires index names, not column names. To refer to a primary key, use the name PRIMARY. To see the index names for a table, use the SHOW INDEX statement or the Information Schema STATISTICS table.

An *index_name* value need not be a full index name. It can be an unambiguous prefix of an index name. If a prefix is ambiguous, an error occurs.

Examples:

```
SELECT * FROM table1 USE INDEX (col1_index,col2_index)
  WHERE col1=1 AND col2=2 AND col3=3;

SELECT * FROM table1 IGNORE INDEX (col3_index)
  WHERE col1=1 AND col2=2 AND col3=3;
```

The syntax for index hints has the following characteristics:

- It is syntactically valid to omit *index_list* for `USE INDEX`, which means “use no indexes.” Omitting *index_list* for `FORCE INDEX` or `IGNORE INDEX` is a syntax error.
- You can specify the scope of an index hint by adding a `FOR` clause to the hint. This provides more fine-grained control over optimizer selection of an execution plan for various phases of query processing. To affect only the indexes used when MySQL decides how to find rows in the table and how to process joins, use `FOR JOIN`. To influence index usage for sorting or grouping rows, use `FOR ORDER BY` or `FOR GROUP BY`.
- You can specify multiple index hints:

  ```
  SELECT * FROM t1 USE INDEX (i1) IGNORE INDEX FOR ORDER BY (i2) ORDER BY a;
  ```

  It is not an error to name the same index in several hints (even within the same hint):

  ```
  SELECT * FROM t1 USE INDEX (i1) USE INDEX (i1,i1);
  ```

  However, it is an error to mix `USE INDEX` and `FORCE INDEX` for the same table:

  ```
  SELECT * FROM t1 USE INDEX FOR JOIN (i1) FORCE INDEX FOR JOIN (i2);
  ```

If an index hint includes no `FOR` clause, the scope of the hint is to apply to all parts of the statement. For example, this hint:

```
IGNORE INDEX (i1)
```

is equivalent to this combination of hints:

```
IGNORE INDEX FOR JOIN (i1)
IGNORE INDEX FOR ORDER BY (i1)
IGNORE INDEX FOR GROUP BY (i1)
```

In MySQL 5.0, hint scope with no `FOR` clause was to apply only to row retrieval. To cause the server to use this older behavior when no `FOR` clause is present, enable the `old` system variable at server startup. Take care about enabling this variable in a replication setup. With statement-based binary logging, having different modes for the source and replicas might lead to replication errors.

When index hints are processed, they are collected in a single list by type (`USE`, `FORCE`, `IGNORE`) and by scope (`FOR JOIN`, `FOR ORDER BY`, `FOR GROUP BY`). For example:

```
SELECT * FROM t1
  USE INDEX () IGNORE INDEX (i2) USE INDEX (i1) USE INDEX (i2);
```

is equivalent to:

```
SELECT * FROM t1
   USE INDEX (i1,i2) IGNORE INDEX (i2);
```

The index hints then are applied for each scope in the following order:

1. `{USE|FORCE} INDEX` is applied if present. (If not, the optimizer-determined set of indexes is used.)

2. `IGNORE INDEX` is applied over the result of the previous step. For example, the following two queries are equivalent:

```
SELECT * FROM t1 USE INDEX (i1) IGNORE INDEX (i2) USE INDEX (i2);

SELECT * FROM t1 USE INDEX (i1);
```

For `FULLTEXT` searches, index hints work as follows:

- For natural language mode searches, index hints are silently ignored. For example, `IGNORE INDEX(i1)` is ignored with no warning and the index is still used.
- For boolean mode searches, index hints with `FOR ORDER BY` or `FOR GROUP BY` are silently ignored. Index hints with `FOR JOIN` or no `FOR` modifier are honored. In contrast to how hints apply for non-`FULLTEXT` searches, the hint is used for all phases of query execution (finding rows and retrieval, grouping, and ordering). This is true even if the hint is given for a non-`FULLTEXT` index.

  For example, the following two queries are equivalent:

```
SELECT * FROM t
  USE INDEX (index1)
  IGNORE INDEX FOR ORDER BY (index1)
  IGNORE INDEX FOR GROUP BY (index1)
  WHERE ... IN BOOLEAN MODE ... ;

SELECT * FROM t
  USE INDEX (index1)
  WHERE ... IN BOOLEAN MODE ... ;
```

Index hints work with `DELETE` statements, but only if you use multi-table `DELETE` syntax, as shown here:

```
mysql> EXPLAIN DELETE FROM t1 USE INDEX(col2)
    -> WHERE col1 BETWEEN 1 AND 100 AND COL2 BETWEEN 1 AND 100\G
ERROR 1064 (42000): You have an error in your SQL syntax; check the manual that
corresponds to your MySQL server version for the right syntax to use near 'use
index(col2) where col1 between 1 and 100 and col2 between 1 and 100' at line 1

mysql> EXPLAIN DELETE t1.* FROM t1 USE INDEX(col2)
    -> WHERE col1 BETWEEN 1 AND 100 AND COL2 BETWEEN 1 AND 100\G
*************************** 1. row ***************************
           id: 1
  select_type: DELETE
        table: t1
   partitions: NULL
         type: range
possible_keys: col2
          key: col2
      key_len: 5
          ref: NULL
         rows: 72
     filtered: 11.11
        Extra: Using where
1 row in set, 1 warning (0.00 sec)
```

8.9.5 The Optimizer Cost Model

To generate execution plans, the optimizer uses a cost model that is based on estimates of the cost of various operations that occur during query execution. The optimizer has a set of compiled-in default “cost constants” available to it to make decisions regarding execution plans.

The optimizer also has a database of cost estimates to use during execution plan construction. These estimates are stored in the `server_cost` and `engine_cost` tables in the `mysql` system database and are configurable at any time. The intent of these tables is to make it possible to easily adjust the cost estimates that the optimizer uses when it attempts to arrive at query execution plans.

- Cost Model General Operation

Cost Model General Operation

The configurable optimizer cost model works like this:

- The server reads the cost model tables into memory at startup and uses the in-memory values at runtime. Any non-`NULL` cost estimate specified in the tables takes precedence over the corresponding compiled-in default cost constant. Any `NULL` estimate indicates to the optimizer to use the compiled-in default.
- At runtime, the server may re-read the cost tables. This occurs when a storage engine is dynamically loaded or when a `FLUSH OPTIMIZER_COSTS` statement is executed.
- Cost tables enable server administrators to easily adjust cost estimates by changing entries in the tables. It is also easy to revert to a default by setting an entry's cost to `NULL`. The optimizer uses the in-memory cost values, so changes to the tables should be followed by `FLUSH OPTIMIZER_COSTS` to take effect.
- The in-memory cost estimates that are current when a client session begins apply throughout that session until it ends. In particular, if the server re-reads the cost tables, any changed estimates apply only to subsequently started sessions. Existing sessions are unaffected.
- Cost tables are specific to a given server instance. The server does not replicate cost table changes to replicas.

The Cost Model Database

The optimizer cost model database consists of two tables in the `mysql` system database that contain cost estimate information for operations that occur during query execution:

- `server_cost`: Optimizer cost estimates for general server operations
- `engine_cost`: Optimizer cost estimates for operations specific to particular storage engines

The `server_cost` table contains these columns:

- `cost_name`

  The name of a cost estimate used in the cost model. The name is not case-sensitive. If the server does not recognize the cost name when it reads this table, it writes a warning to the error log.

- `cost_value`

  The cost estimate value. If the value is non-`NULL`, the server uses it as the cost. Otherwise, it uses the default estimate (the compiled-in value). DBAs can change a cost estimate by updating this column. If the server finds that the cost value is invalid (nonpositive) when it reads this table, it writes a warning to the error log.

  To override a default cost estimate (for an entry that specifies `NULL`), set the cost to a non-`NULL` value. To revert to the default, set the value to `NULL`. Then execute `FLUSH OPTIMIZER_COSTS` to tell the server to re-read the cost tables.

- `last_update`

  The time of the last row update.

- `comment`

  A descriptive comment associated with the cost estimate. DBAs can use this column to provide information about why a cost estimate row stores a particular value.

- `default_value`

  The default (compiled-in) value for the cost estimate. This column is a read-only generated column that retains its value even if the associated cost estimate is changed. For rows added to the table at runtime, the value of this column is `NULL`.

The primary key for the `server_cost` table is the `cost_name` column, so it is not possible to create multiple entries for any cost estimate.

The server recognizes these `cost_name` values for the `server_cost` table:

- `disk_temptable_create_cost`, `disk_temptable_row_cost`

  The cost estimates for internally created temporary tables stored in a disk-based storage engine (either `InnoDB` or `MyISAM`). Increasing these values increases the cost estimate of using internal temporary tables and makes the optimizer prefer query plans with less use of them. For information about such tables, see Section 8.4.4, “Internal Temporary Table Use in MySQL”.

  The larger default values for these disk parameters compared to the default values for the corresponding memory parameters (`memory_temptable_create_cost`, `memory_temptable_row_cost`) reflects the greater cost of processing disk-based tables.

- `key_compare_cost`

  The cost of comparing record keys. Increasing this value causes a query plan that compares many keys to become more expensive. For example, a query plan that performs a `filesort` becomes relatively more expensive compared to a query plan that avoids sorting by using an index.

- `memory_temptable_create_cost`, `memory_temptable_row_cost`

  The cost estimates for internally created temporary tables stored in the `MEMORY` storage engine. Increasing these values increases the cost estimate of using internal temporary tables and makes the optimizer prefer query plans with less use of them. For information about such tables, see Section 8.4.4, “Internal Temporary Table Use in MySQL”.

  The smaller default values for these memory parameters compared to the default values for the corresponding disk parameters (`disk_temptable_create_cost`, `disk_temptable_row_cost`) reflects the lesser cost of processing memory-based tables.

- `row_evaluate_cost`

  The cost of evaluating record conditions. Increasing this value causes a query plan that examines many rows to become more expensive compared to a query plan that examines fewer rows. For example, a table scan becomes relatively more expensive compared to a range scan that reads fewer rows.

The `engine_cost` table contains these columns:

- `engine_name`

  The name of the storage engine to which this cost estimate applies. The name is not case-sensitive. If the value is `default`, it applies to all storage engines that have no named entry of their own. If the server does not recognize the engine name when it reads this table, it writes a warning to the error log.

- `device_type`

  The device type to which this cost estimate applies. The column is intended for specifying different cost estimates for different storage device types, such as hard disk drives versus solid state drives. Currently, this information is not used and 0 is the only permitted value.

- `cost_name`

Same as in the `server_cost` table.

- `cost_value`

  Same as in the `server_cost` table.

- `last_update`

  Same as in the `server_cost` table.

- `comment`

  Same as in the `server_cost` table.

- `default_value`

  The default (compiled-in) value for the cost estimate. This column is a read-only generated column that retains its value even if the associated cost estimate is changed. For rows added to the table at runtime, the value of this column is `NULL`, with the exception that if the row has the same `cost_name` value as one of the original rows, the `default_value` column has the same value as that row.

The primary key for the `engine_cost` table is a tuple comprising the (`cost_name`, `engine_name`, `device_type`) columns, so it is not possible to create multiple entries for any combination of values in those columns.

The server recognizes these `cost_name` values for the `engine_cost` table:

- `io_block_read_cost`

  The cost of reading an index or data block from disk. Increasing this value causes a query plan that reads many disk blocks to become more expensive compared to a query plan that reads fewer disk blocks. For example, a table scan becomes relatively more expensive compared to a range scan that reads fewer blocks.

- `memory_block_read_cost`

  Similar to `io_block_read_cost`, but represents the cost of reading an index or data block from an in-memory database buffer.

If the `io_block_read_cost` and `memory_block_read_cost` values differ, the execution plan may change between two runs of the same query. Suppose that the cost for memory access is less than the cost for disk access. In that case, at server startup before data has been read into the buffer pool, you may get a different plan than after the query has been run because then the data is in memory.

Making Changes to the Cost Model Database

For DBAs who wish to change the cost model parameters from their defaults, try doubling or halving the value and measuring the effect.

Changes to the `io_block_read_cost` and `memory_block_read_cost` parameters are most likely to yield worthwhile results. These parameter values enable cost models for data access methods to take into account the costs of reading information from different sources; that is, the cost of reading information from disk versus reading information already in a memory buffer. For example, all other things being equal, setting `io_block_read_cost` to a value larger than `memory_block_read_cost` causes the optimizer to prefer query plans that read information already held in memory to plans that must read from disk.

This example shows how to change the default value for `io_block_read_cost`:

```
UPDATE mysql.engine_cost
  SET cost_value = 2.0
  WHERE cost_name = 'io_block_read_cost';
```

```
FLUSH OPTIMIZER_COSTS;
```

This example shows how to change the value of `io_block_read_cost` only for the `InnoDB` storage engine:

```
INSERT INTO mysql.engine_cost
  VALUES ('InnoDB', 0, 'io_block_read_cost', 3.0,
  CURRENT_TIMESTAMP, 'Using a slower disk for InnoDB');
FLUSH OPTIMIZER_COSTS;
```

8.9.6 Optimizer Statistics

The `column_statistics` data dictionary table stores histogram statistics about column values, for use by the optimizer in constructing query execution plans. To perform histogram management, use the `ANALYZE TABLE` statement.

The `column_statistics` table has these characteristics:

- The table contains statistics for columns of all data types except geometry types (spatial data) and `JSON`.
- The table is persistent so that column statistics need not be created each time the server starts.
- The server performs updates to the table; users do not.

The `column_statistics` table is not directly accessible by users because it is part of the data dictionary. Histogram information is available using `INFORMATION_SCHEMA.COLUMN_STATISTICS`, which is implemented as a view on the data dictionary table. `COLUMN_STATISTICS` has these columns:

- `SCHEMA_NAME`, `TABLE_NAME`, `COLUMN_NAME`: The names of the schema, table, and column for which the statistics apply.
- `HISTOGRAM`: A `JSON` value describing the column statistics, stored as a histogram.

Column histograms contain buckets for parts of the range of values stored in the column. Histograms are `JSON` objects to permit flexibility in the representation of column statistics. Here is a sample histogram object:

```
{
  "buckets": [
    [
      1,
      0.3333333333333333
    ],
    [
      2,
      0.6666666666666666
    ],
    [
      3,
      1
    ]
  ],
  "null-values": 0,
  "last-updated": "2017-03-24 13:32:40.000000",
  "sampling-rate": 1,
  "histogram-type": "singleton",
  "number-of-buckets-specified": 128,
  "data-type": "int",
  "collation-id": 8
}
```

Histogram objects have these keys:

- `buckets`: The histogram buckets. Bucket structure depends on the histogram type.

  For `singleton` histograms, buckets contain two values:

  - Value 1: The value for the bucket. The type depends on the column data type.
  - Value 2: A double representing the cumulative frequency for the value. For example, .25 and .75 indicate that 25% and 75% of the values in the column are less than or equal to the bucket value.

  For `equi-height` histograms, buckets contain four values:

  - Values 1, 2: The lower and upper inclusive values for the bucket. The type depends on the column data type.
  - Value 3: A double representing the cumulative frequency for the value. For example, .25 and .75 indicate that 25% and 75% of the values in the column are less than or equal to the bucket upper value.
  - Value 4: The number of distinct values in the range from the bucket lower value to its upper value.

- `null-values`: A number between 0.0 and 1.0 indicating the fraction of column values that are SQL `NULL` values. If 0, the column contains no `NULL` values.
- `last-updated`: When the histogram was generated, as a UTC value in *`YYYY-MM-DD hh:mm:ss.uuuuuu`* format.
- `sampling-rate`: A number between 0.0 and 1.0 indicating the fraction of data that was sampled to create the histogram. A value of 1 means that all of the data was read (no sampling).
- `histogram-type`: The histogram type:
  - `singleton`: One bucket represents one single value in the column. This histogram type is created when the number of distinct values in the column is less than or equal to the number of buckets specified in the `ANALYZE TABLE` statement that generated the histogram.
  - `equi-height`: One bucket represents a range of values. This histogram type is created when the number of distinct values in the column is greater than the number of buckets specified in the `ANALYZE TABLE` statement that generated the histogram.
- `number-of-buckets-specified`: The number of buckets specified in the `ANALYZE TABLE` statement that generated the histogram.
- `data-type`: The type of data this histogram contains. This is needed when reading and parsing histograms from persistent storage into memory. The value is one of `int`, `uint` (unsigned integer), `double`, `decimal`, `datetime`, or `string` (includes character and binary strings).
- `collation-id`: The collation ID for the histogram data. It is mostly meaningful when the `data-type` value is `string`. Values correspond to `ID` column values in the Information Schema `COLLATIONS` table.

To extract particular values from the histogram objects, you can use `JSON` operations. For example:

```
mysql> SELECT
         TABLE_NAME, COLUMN_NAME,
         HISTOGRAM->>'$."data-type"' AS 'data-type',
         JSON_LENGTH(HISTOGRAM->>'$."buckets"') AS 'bucket-count'
       FROM INFORMATION_SCHEMA.COLUMN_STATISTICS;
+-----------------+-------------+-----------+--------------+
| TABLE_NAME      | COLUMN_NAME | data-type | bucket-count |
+-----------------+-------------+-----------+--------------+
| country         | Population  | int       |          226 |
| city            | Population  | int       |         1024 |
| countrylanguage | Language    | string    |          457 |
+-----------------+-------------+-----------+--------------+
```

The optimizer uses histogram statistics, if applicable, for columns of any data type for which statistics are collected. The optimizer applies histogram statistics to determine row estimates based on the

selectivity (filtering effect) of column value comparisons against constant values. Predicates of these forms qualify for histogram use:

```
col_name = constant
col_name <> constant
col_name != constant
col_name > constant
col_name < constant
col_name >= constant
col_name <= constant
col_name IS NULL
col_name IS NOT NULL
col_name BETWEEN constant AND constant
col_name NOT BETWEEN constant AND constant
col_name IN (constant[, constant] ...)
col_name NOT IN (constant[, constant] ...)
```

For example, these statements contain predicates that qualify for histogram use:

```
SELECT * FROM orders WHERE amount BETWEEN 100.0 AND 300.0;
SELECT * FROM tbl WHERE col1 = 15 AND col2 > 100;
```

The requirement for comparison against a constant value includes functions that are constant, such as `ABS()` and `FLOOR()`:

```
SELECT * FROM tbl WHERE col1 < ABS(-34);
```

Histogram statistics are useful primarily for nonindexed columns. Adding an index to a column for which histogram statistics are applicable might also help the optimizer make row estimates. The tradeoffs are:

- An index must be updated when table data is modified.
- A histogram is created or updated only on demand, so it adds no overhead when table data is modified. On the other hand, the statistics become progressively more out of date when table modifications occur, until the next time they are updated.

The optimizer prefers range optimizer row estimates to those obtained from histogram statistics. If the optimizer determines that the range optimizer applies, it does not use histogram statistics.

For columns that are indexed, row estimates can be obtained for equality comparisons using index dives (see Section 8.2.1.2, “Range Optimization”). In this case, histogram statistics are not necessarily useful because index dives can yield better estimates.

In some cases, use of histogram statistics may not improve query execution (for example, if the statistics are out of date). To check whether this is the case, use `ANALYZE TABLE` to regenerate the histogram statistics, then run the query again.

Alternatively, to disable histogram statistics, use `ANALYZE TABLE` to drop them. A different method of disabling histogram statistics is to turn off the `condition_fanout_filter` flag of the `optimizer_switch` system variable (although this may disable other optimizations as well):

```
SET optimizer_switch='condition_fanout_filter=off';
```

If histogram statistics are used, the resulting effect is visible using `EXPLAIN`. Consider the following query, where no index is available for column `col1`:

```
SELECT * FROM t1 WHERE col1 < 24;
```

If histogram statistics indicate that 57% of the rows in `t1` satisfy the `col1 < 24` predicate, filtering can occur even in the absence of an index, and `EXPLAIN` shows 57.00 in the `filtered` column.

8.10 Buffering and Caching

MySQL uses several strategies that cache information in memory buffers to increase performance.

8.10.1 InnoDB Buffer Pool Optimization

`InnoDB` maintains a storage area called the buffer pool for caching data and indexes in memory. Knowing how the `InnoDB` buffer pool works, and taking advantage of it to keep frequently accessed data in memory, is an important aspect of MySQL tuning.

For an explanation of the inner workings of the `InnoDB` buffer pool, an overview of its LRU replacement algorithm, and general configuration information, see Section 15.5.1, “Buffer Pool”.

For additional `InnoDB` buffer pool configuration and tuning information, see these sections:

- Section 15.8.3.4, “Configuring InnoDB Buffer Pool Prefetching (Read-Ahead)”
- Section 15.8.3.5, “Configuring Buffer Pool Flushing”
- Section 15.8.3.3, “Making the Buffer Pool Scan Resistant”
- Section 15.8.3.2, “Configuring Multiple Buffer Pool Instances”
- Section 15.8.3.6, “Saving and Restoring the Buffer Pool State”
- Section 15.8.3.1, “Configuring InnoDB Buffer Pool Size”

8.10.2 The MyISAM Key Cache

To minimize disk I/O, the `MyISAM` storage engine exploits a strategy that is used by many database management systems. It employs a cache mechanism to keep the most frequently accessed table blocks in memory:

- For index blocks, a special structure called the *key cache* (or *key buffer*) is maintained. The structure contains a number of block buffers where the most-used index blocks are placed.
- For data blocks, MySQL uses no special cache. Instead it relies on the native operating system file system cache.

This section first describes the basic operation of the `MyISAM` key cache. Then it discusses features that improve key cache performance and that enable you to better control cache operation:

- Multiple sessions can access the cache concurrently.
- You can set up multiple key caches and assign table indexes to specific caches.

To control the size of the key cache, use the `key_buffer_size` system variable. If this variable is set equal to zero, no key cache is used. The key cache also is not used if the `key_buffer_size` value is too small to allocate the minimal number of block buffers (8).

When the key cache is not operational, index files are accessed using only the native file system buffering provided by the operating system. (In other words, table index blocks are accessed using the same strategy as that employed for table data blocks.)

An index block is a contiguous unit of access to the `MyISAM` index files. Usually the size of an index block is equal to the size of nodes of the index B-tree. (Indexes are represented on disk using a B-tree data structure. Nodes at the bottom of the tree are leaf nodes. Nodes above the leaf nodes are nonleaf nodes.)

All block buffers in a key cache structure are the same size. This size can be equal to, greater than, or less than the size of a table index block. Usually one these two values is a multiple of the other.

When data from any table index block must be accessed, the server first checks whether it is available in some block buffer of the key cache. If it is, the server accesses data in the key cache rather than on disk. That is, it reads from the cache or writes into it rather than reading from or writing to disk. Otherwise, the server chooses a cache block buffer containing a different table index block (or blocks)

and replaces the data there by a copy of required table index block. As soon as the new index block is in the cache, the index data can be accessed.

If it happens that a block selected for replacement has been modified, the block is considered “dirty.” In this case, prior to being replaced, its contents are flushed to the table index from which it came.

Usually the server follows an *LRU (Least Recently Used)* strategy: When choosing a block for replacement, it selects the least recently used index block. To make this choice easier, the key cache module maintains all used blocks in a special list (*LRU chain*) ordered by time of use. When a block is accessed, it is the most recently used and is placed at the end of the list. When blocks need to be replaced, blocks at the beginning of the list are the least recently used and become the first candidates for eviction.

The `InnoDB` storage engine also uses an LRU algorithm, to manage its buffer pool. See Section 15.5.1, “Buffer Pool”.

8.10.2.1 Shared Key Cache Access

Threads can access key cache buffers simultaneously, subject to the following conditions:

- A buffer that is not being updated can be accessed by multiple sessions.
- A buffer that is being updated causes sessions that need to use it to wait until the update is complete.
- Multiple sessions can initiate requests that result in cache block replacements, as long as they do not interfere with each other (that is, as long as they need different index blocks, and thus cause different cache blocks to be replaced).

Shared access to the key cache enables the server to improve throughput significantly.

8.10.2.2 Multiple Key Caches

Note

As of MySQL 8.0, the compound-part structured-variable syntax discussed here for referring to multiple `MyISAM` key caches is deprecated.

Shared access to the key cache improves performance but does not eliminate contention among sessions entirely. They still compete for control structures that manage access to the key cache buffers. To reduce key cache access contention further, MySQL also provides multiple key caches. This feature enables you to assign different table indexes to different key caches.

Where there are multiple key caches, the server must know which cache to use when processing queries for a given `MyISAM` table. By default, all `MyISAM` table indexes are cached in the default key cache. To assign table indexes to a specific key cache, use the `CACHE INDEX` statement (see Section 13.7.8.2, “CACHE INDEX Statement”). For example, the following statement assigns indexes from the tables `t1`, `t2`, and `t3` to the key cache named `hot_cache`:

```
mysql> CACHE INDEX t1, t2, t3 IN hot_cache;
+---------+--------------------+----------+----------+
| Table   | Op                 | Msg_type | Msg_text |
+---------+--------------------+----------+----------+
| test.t1 | assign_to_keycache | status   | OK       |
| test.t2 | assign_to_keycache | status   | OK       |
| test.t3 | assign_to_keycache | status   | OK       |
+---------+--------------------+----------+----------+
```

The key cache referred to in a `CACHE INDEX` statement can be created by setting its size with a `SET GLOBAL` parameter setting statement or by using server startup options. For example:

```
mysql> SET GLOBAL keycache1.key_buffer_size=128*1024;
```

To destroy a key cache, set its size to zero:

```
mysql> SET GLOBAL keycache1.key_buffer_size=0;
```

You cannot destroy the default key cache. Any attempt to do this is ignored:

```
mysql> SET GLOBAL key_buffer_size = 0;

mysql> SHOW VARIABLES LIKE 'key_buffer_size';
+-----------------+---------+
| Variable_name   | Value   |
+-----------------+---------+
| key_buffer_size | 8384512 |
+-----------------+---------+
```

Key cache variables are structured system variables that have a name and components. For `keycache1.key_buffer_size`, `keycache1` is the cache variable name and `key_buffer_size` is the cache component. See Section 5.1.9.5, “Structured System Variables”, for a description of the syntax used for referring to structured key cache system variables.

By default, table indexes are assigned to the main (default) key cache created at the server startup. When a key cache is destroyed, all indexes assigned to it are reassigned to the default key cache.

For a busy server, you can use a strategy that involves three key caches:

- A “hot” key cache that takes up 20% of the space allocated for all key caches. Use this for tables that are heavily used for searches but that are not updated.
- A “cold” key cache that takes up 20% of the space allocated for all key caches. Use this cache for medium-sized, intensively modified tables, such as temporary tables.
- A “warm” key cache that takes up 60% of the key cache space. Employ this as the default key cache, to be used by default for all other tables.

One reason the use of three key caches is beneficial is that access to one key cache structure does not block access to the others. Statements that access tables assigned to one cache do not compete with statements that access tables assigned to another cache. Performance gains occur for other reasons as well:

- The hot cache is used only for retrieval queries, so its contents are never modified. Consequently, whenever an index block needs to be pulled in from disk, the contents of the cache block chosen for replacement need not be flushed first.
- For an index assigned to the hot cache, if there are no queries requiring an index scan, there is a high probability that the index blocks corresponding to nonleaf nodes of the index B-tree remain in the cache.
- An update operation most frequently executed for temporary tables is performed much faster when the updated node is in the cache and need not be read from disk first. If the size of the indexes of the temporary tables are comparable with the size of cold key cache, the probability is very high that the updated node is in the cache.

The `CACHE INDEX` statement sets up an association between a table and a key cache, but the association is lost each time the server restarts. If you want the association to take effect each time the server starts, one way to accomplish this is to use an option file: Include variable settings that configure your key caches, and an `init_file` system variable that names a file containing `CACHE INDEX` statements to be executed. For example:

```
key_buffer_size = 4G
hot_cache.key_buffer_size = 2G
cold_cache.key_buffer_size = 2G
init_file=/path/to/data-directory/mysqld_init.sql
```

The statements in `mysqld_init.sql` are executed each time the server starts. The file should contain one SQL statement per line. The following example assigns several tables each to `hot_cache` and `cold_cache`:

```
CACHE INDEX db1.t1, db1.t2, db2.t3 IN hot_cache
CACHE INDEX db1.t4, db2.t5, db2.t6 IN cold_cache
```

8.10.2.3 Midpoint Insertion Strategy

By default, the key cache management system uses a simple LRU strategy for choosing key cache blocks to be evicted, but it also supports a more sophisticated method called the *midpoint insertion strategy.*

When using the midpoint insertion strategy, the LRU chain is divided into two parts: a hot sublist and a warm sublist. The division point between two parts is not fixed, but the key cache management system takes care that the warm part is not “too short,” always containing at least `key_cache_division_limit` percent of the key cache blocks. `key_cache_division_limit` is a component of structured key cache variables, so its value is a parameter that can be set per cache.

When an index block is read from a table into the key cache, it is placed at the end of the warm sublist. After a certain number of hits (accesses of the block), it is promoted to the hot sublist. At present, the number of hits required to promote a block (3) is the same for all index blocks.

A block promoted into the hot sublist is placed at the end of the list. The block then circulates within this sublist. If the block stays at the beginning of the sublist for a long enough time, it is demoted to the warm sublist. This time is determined by the value of the `key_cache_age_threshold` component of the key cache.

The threshold value prescribes that, for a key cache containing *`N`* blocks, the block at the beginning of the hot sublist not accessed within the last *`N`* `* key_cache_age_threshold / 100` hits is to be moved to the beginning of the warm sublist. It then becomes the first candidate for eviction, because blocks for replacement always are taken from the beginning of the warm sublist.

The midpoint insertion strategy enables you to keep more-valued blocks always in the cache. If you prefer to use the plain LRU strategy, leave the `key_cache_division_limit` value set to its default of 100.

The midpoint insertion strategy helps to improve performance when execution of a query that requires an index scan effectively pushes out of the cache all the index blocks corresponding to valuable high-level B-tree nodes. To avoid this, you must use a midpoint insertion strategy with the `key_cache_division_limit` set to much less than 100. Then valuable frequently hit nodes are preserved in the hot sublist during an index scan operation as well.

8.10.2.4 Index Preloading

If there are enough blocks in a key cache to hold blocks of an entire index, or at least the blocks corresponding to its nonleaf nodes, it makes sense to preload the key cache with index blocks before starting to use it. Preloading enables you to put the table index blocks into a key cache buffer in the most efficient way: by reading the index blocks from disk sequentially.

Without preloading, the blocks are still placed into the key cache as needed by queries. Although the blocks stay in the cache, because there are enough buffers for all of them, they are fetched from disk in random order, and not sequentially.

To preload an index into a cache, use the `LOAD INDEX INTO CACHE` statement. For example, the following statement preloads nodes (index blocks) of indexes of the tables `t1` and `t2`:

```
mysql> LOAD INDEX INTO CACHE t1, t2 IGNORE LEAVES;
+---------+--------------+----------+----------+
| Table   | Op           | Msg_type | Msg_text |
+---------+--------------+----------+----------+
| test.t1 | preload_keys | status   | OK       |
| test.t2 | preload_keys | status   | OK       |
+---------+--------------+----------+----------+
```

The `IGNORE LEAVES` modifier causes only blocks for the nonleaf nodes of the index to be preloaded. Thus, the statement shown preloads all index blocks from `t1`, but only blocks for the nonleaf nodes from `t2`.

If an index has been assigned to a key cache using a `CACHE INDEX` statement, preloading places index blocks into that cache. Otherwise, the index is loaded into the default key cache.

8.10.2.5 Key Cache Block Size

It is possible to specify the size of the block buffers for an individual key cache using the `key_cache_block_size` variable. This permits tuning of the performance of I/O operations for index files.

The best performance for I/O operations is achieved when the size of read buffers is equal to the size of the native operating system I/O buffers. But setting the size of key nodes equal to the size of the I/O buffer does not always ensure the best overall performance. When reading the big leaf nodes, the server pulls in a lot of unnecessary data, effectively preventing reading other leaf nodes.

To control the size of blocks in the `.MYI` index file of `MyISAM` tables, use the `--myisam-block-size` option at server startup.

8.10.2.6 Restructuring a Key Cache

A key cache can be restructured at any time by updating its parameter values. For example:

```
mysql> SET GLOBAL cold_cache.key_buffer_size=4*1024*1024;
```

If you assign to either the `key_buffer_size` or `key_cache_block_size` key cache component a value that differs from the component's current value, the server destroys the cache's old structure and creates a new one based on the new values. If the cache contains any dirty blocks, the server saves them to disk before destroying and re-creating the cache. Restructuring does not occur if you change other key cache parameters.

When restructuring a key cache, the server first flushes the contents of any dirty buffers to disk. After that, the cache contents become unavailable. However, restructuring does not block queries that need to use indexes assigned to the cache. Instead, the server directly accesses the table indexes using native file system caching. File system caching is not as efficient as using a key cache, so although queries execute, a slowdown can be anticipated. After the cache has been restructured, it becomes available again for caching indexes assigned to it, and the use of file system caching for the indexes ceases.

8.10.3 Caching of Prepared Statements and Stored Programs

For certain statements that a client might execute multiple times during a session, the server converts the statement to an internal structure and caches that structure to be used during execution. Caching enables the server to perform more efficiently because it avoids the overhead of reconverting the statement should it be needed again during the session. Conversion and caching occurs for these statements:

- Prepared statements, both those processed at the SQL level (using the `PREPARE` statement) and those processed using the binary client/server protocol (using the `mysql_stmt_prepare()` C API function). The `max_prepared_stmt_count` system variable controls the total number of statements the server caches. (The sum of the number of prepared statements across all sessions.)
- Stored programs (stored procedures and functions, triggers, and events). In this case, the server converts and caches the entire program body. The `stored_program_cache` system variable indicates the approximate number of stored programs the server caches per session.

The server maintains caches for prepared statements and stored programs on a per-session basis. Statements cached for one session are not accessible to other sessions. When a session ends, the server discards any statements cached for it.

When the server uses a cached internal statement structure, it must take care that the structure does not go out of date. Metadata changes can occur for an object used by the statement, causing a mismatch between the current object definition and the definition as represented in the internal statement structure. Metadata changes occur for DDL statements such as those that create, drop, alter, rename, or truncate tables, or that analyze, optimize, or repair tables. Table content changes (for example, with `INSERT` or `UPDATE`) do not change metadata, nor do `SELECT` statements.

Here is an illustration of the problem. Suppose that a client prepares this statement:

```
PREPARE s1 FROM 'SELECT * FROM t1';
```

The `SELECT *` expands in the internal structure to the list of columns in the table. If the set of columns in the table is modified with `ALTER TABLE`, the prepared statement goes out of date. If the server does not detect this change the next time the client executes `s1`, the prepared statement returns incorrect results.

To avoid problems caused by metadata changes to tables or views referred to by the prepared statement, the server detects these changes and automatically reprepares the statement when it is next executed. That is, the server reparses the statement and rebuilds the internal structure. Reparsing also occurs after referenced tables or views are flushed from the table definition cache, either implicitly to make room for new entries in the cache, or explicitly due to `FLUSH TABLES`.

Similarly, if changes occur to objects used by a stored program, the server reparses affected statements within the program.

The server also detects metadata changes for objects in expressions. These might be used in statements specific to stored programs, such as `DECLARE CURSOR` or flow-control statements such as `IF`, `CASE`, and `RETURN`.

To avoid reparsing entire stored programs, the server reparses affected statements or expressions within a program only as needed. Examples:

- Suppose that metadata for a table or view is changed. Reparsing occurs for a `SELECT *` within the program that accesses the table or view, but not for a `SELECT *` that does not access the table or view.
- When a statement is affected, the server reparses it only partially if possible. Consider this `CASE` statement:

  ```
  CASE case_expr
    WHEN when_expr1 ...
    WHEN when_expr2 ...
    WHEN when_expr3 ...
    ...
  END CASE
  ```

  If a metadata change affects only `WHEN when_expr3`, that expression is reparsed. `case_expr` and the other `WHEN` expressions are not reparsed.

Reparsing uses the default database and SQL mode that were in effect for the original conversion to internal form.

The server attempts reparsing up to three times. An error occurs if all attempts fail.

Reparsing is automatic, but to the extent that it occurs, diminishes prepared statement and stored program performance.

For prepared statements, the `Com_stmt_reprepare` status variable tracks the number of repreparations.

8.11 Optimizing Locking Operations

MySQL manages contention for table contents using locking:

- Internal locking is performed within the MySQL server itself to manage contention for table contents by multiple threads. This type of locking is internal because it is performed entirely by the server and involves no other programs. See Section 8.11.1, “Internal Locking Methods”.
- External locking occurs when the server and other programs lock `MyISAM` table files to coordinate among themselves which program can access the tables at which time. See Section 8.11.5, “External Locking”.

8.11.1 Internal Locking Methods

This section discusses internal locking; that is, locking performed within the MySQL server itself to manage contention for table contents by multiple sessions. This type of locking is internal because it is performed entirely by the server and involves no other programs. For locking performed on MySQL files by other programs, see Section 8.11.5, “External Locking”.

- Row-Level Locking
- Table-Level Locking
- Choosing the Type of Locking

Row-Level Locking

MySQL uses row-level locking for `InnoDB` tables to support simultaneous write access by multiple sessions, making them suitable for multi-user, highly concurrent, and OLTP applications.

To avoid deadlocks when performing multiple concurrent write operations on a single `InnoDB` table, acquire necessary locks at the start of the transaction by issuing a `SELECT ... FOR UPDATE` statement for each group of rows expected to be modified, even if the data change statements come later in the transaction. If transactions modify or lock more than one table, issue the applicable statements in the same order within each transaction. Deadlocks affect performance rather than representing a serious error, because `InnoDB` automatically detects deadlock conditions by default and rolls back one of the affected transactions.

On high concurrency systems, deadlock detection can cause a slowdown when numerous threads wait for the same lock. At times, it may be more efficient to disable deadlock detection and rely on the `innodb_lock_wait_timeout` setting for transaction rollback when a deadlock occurs. Deadlock detection can be disabled using the `innodb_deadlock_detect` configuration option.

Advantages of row-level locking:

- Fewer lock conflicts when different sessions access different rows.
- Fewer changes for rollbacks.
- Possible to lock a single row for a long time.

Table-Level Locking

MySQL uses table-level locking for `MyISAM`, `MEMORY`, and `MERGE` tables, permitting only one session to update those tables at a time. This locking level makes these storage engines more suitable for read-only, read-mostly, or single-user applications.

These storage engines avoid deadlocks by always requesting all needed locks at once at the beginning of a query and always locking the tables in the same order. The tradeoff is that this strategy reduces concurrency; other sessions that want to modify the table must wait until the current data change statement finishes.

Advantages of table-level locking:

- Relatively little memory required (row locking requires memory per row or group of rows locked)

- Fast when used on a large part of the table because only a single lock is involved.
- Fast if you often do `GROUP BY` operations on a large part of the data or must scan the entire table frequently.

MySQL grants table write locks as follows:

1. If there are no locks on the table, put a write lock on it.
2. Otherwise, put the lock request in the write lock queue.

MySQL grants table read locks as follows:

1. If there are no write locks on the table, put a read lock on it.
2. Otherwise, put the lock request in the read lock queue.

Table updates are given higher priority than table retrievals. Therefore, when a lock is released, the lock is made available to the requests in the write lock queue and then to the requests in the read lock queue. This ensures that updates to a table are not "starved" even when there is heavy `SELECT` activity for the table. However, if there are many updates for a table, `SELECT` statements wait until there are no more updates.

For information on altering the priority of reads and writes, see Section 8.11.2, "Table Locking Issues".

You can analyze the table lock contention on your system by checking the `Table_locks_immediate` and `Table_locks_waited` status variables, which indicate the number of times that requests for table locks could be granted immediately and the number that had to wait, respectively:

```
mysql> SHOW STATUS LIKE 'Table%';
+-----------------------+---------+
| Variable_name         | Value   |
+-----------------------+---------+
| Table_locks_immediate | 1151552 |
| Table_locks_waited    | 15324   |
+-----------------------+---------+
```

The Performance Schema lock tables also provide locking information. See Section 27.12.13, "Performance Schema Lock Tables".

The `MyISAM` storage engine supports concurrent inserts to reduce contention between readers and writers for a given table: If a `MyISAM` table has no free blocks in the middle of the data file, rows are always inserted at the end of the data file. In this case, you can freely mix concurrent `INSERT` and `SELECT` statements for a `MyISAM` table without locks. That is, you can insert rows into a `MyISAM` table at the same time other clients are reading from it. Holes can result from rows having been deleted from or updated in the middle of the table. If there are holes, concurrent inserts are disabled but are enabled again automatically when all holes have been filled with new data. To control this behavior, use the `concurrent_insert` system variable. See Section 8.11.3, "Concurrent Inserts".

If you acquire a table lock explicitly with `LOCK TABLES`, you can request a `READ LOCAL` lock rather than a `READ` lock to enable other sessions to perform concurrent inserts while you have the table locked.

To perform many `INSERT` and `SELECT` operations on a table `t1` when concurrent inserts are not possible, you can insert rows into a temporary table `temp_t1` and update the real table with the rows from the temporary table:

```
mysql> LOCK TABLES t1 WRITE, temp_t1 WRITE;
mysql> INSERT INTO t1 SELECT * FROM temp_t1;
mysql> DELETE FROM temp_t1;
mysql> UNLOCK TABLES;
```

Choosing the Type of Locking

Generally, table locks are superior to row-level locks in the following cases:

- Most statements for the table are reads.
- Statements for the table are a mix of reads and writes, where writes are updates or deletes for a single row that can be fetched with one key read:

```
UPDATE tbl_name SET column=value WHERE unique_key_col=key_value;
DELETE FROM tbl_name WHERE unique_key_col=key_value;
```

- `SELECT` combined with concurrent `INSERT` statements, and very few `UPDATE` or `DELETE` statements.
- Many scans or `GROUP BY` operations on the entire table without any writers.

With higher-level locks, you can more easily tune applications by supporting locks of different types, because the lock overhead is less than for row-level locks.

Options other than row-level locking:

- Versioning (such as that used in MySQL for concurrent inserts) where it is possible to have one writer at the same time as many readers. This means that the database or table supports different views for the data depending on when access begins. Other common terms for this are “time travel,” “copy on write,” or “copy on demand.”
- Copy on demand is in many cases superior to row-level locking. However, in the worst case, it can use much more memory than using normal locks.
- Instead of using row-level locks, you can employ application-level locks, such as those provided by `GET_LOCK()` and `RELEASE_LOCK()` in MySQL. These are advisory locks, so they work only with applications that cooperate with each other. See Section 12.15, “Locking Functions”.

8.11.2 Table Locking Issues

`InnoDB` tables use row-level locking so that multiple sessions and applications can read from and write to the same table simultaneously, without making each other wait or producing inconsistent results. For this storage engine, avoid using the `LOCK TABLES` statement, because it does not offer any extra protection, but instead reduces concurrency. The automatic row-level locking makes these tables suitable for your busiest databases with your most important data, while also simplifying application logic since you do not need to lock and unlock tables. Consequently, the `InnoDB` storage engine is the default in MySQL.

MySQL uses table locking (instead of page, row, or column locking) for all storage engines except `InnoDB`. The locking operations themselves do not have much overhead. But because only one session can write to a table at any one time, for best performance with these other storage engines, use them primarily for tables that are queried often and rarely inserted into or updated.

- Performance Considerations Favoring InnoDB
- Workarounds for Locking Performance Issues

Performance Considerations Favoring InnoDB

When choosing whether to create a table using `InnoDB` or a different storage engine, keep in mind the following disadvantages of table locking:

- Table locking enables many sessions to read from a table at the same time, but if a session wants to write to a table, it must first get exclusive access, meaning it might have to wait for other sessions to finish with the table first. During the update, all other sessions that want to access this particular table must wait until the update is done.
- Table locking causes problems when a session is waiting because the disk is full and free space needs to become available before the session can proceed. In this case, all sessions that want to access the problem table are also put in a waiting state until more disk space is made available.

- A SELECT statement that takes a long time to run prevents other sessions from updating the table in the meantime, making the other sessions appear slow or unresponsive. While a session is waiting to get exclusive access to the table for updates, other sessions that issue SELECT statements queue up behind it, reducing concurrency even for read-only sessions.

Workarounds for Locking Performance Issues

The following items describe some ways to avoid or reduce contention caused by table locking:

- Consider switching the table to the InnoDB storage engine, either using CREATE TABLE ... ENGINE=INNODB during setup, or using ALTER TABLE ... ENGINE=INNODB for an existing table. See Chapter 15, *The InnoDB Storage Engine* for more details about this storage engine.
- Optimize SELECT statements to run faster so that they lock tables for a shorter time. You might have to create some summary tables to do this.
- Start mysqld with --low-priority-updates. For storage engines that use only table-level locking (such as MyISAM, MEMORY, and MERGE), this gives all statements that update (modify) a table lower priority than SELECT statements. In this case, the second SELECT statement in the preceding scenario would execute before the UPDATE statement, and would not wait for the first SELECT to finish.
- To specify that all updates issued in a specific connection should be done with low priority, set the low_priority_updates server system variable equal to 1.
- To give a specific INSERT, UPDATE, or DELETE statement lower priority, use the LOW_PRIORITY attribute.
- To give a specific SELECT statement higher priority, use the HIGH_PRIORITY attribute. See Section 13.2.13, “SELECT Statement”.
- Start mysqld with a low value for the max_write_lock_count system variable to force MySQL to temporarily elevate the priority of all SELECT statements that are waiting for a table after a specific number of write locks to the table occur (for example, for insert operations). This permits read locks after a certain number of write locks.
- If you have problems with mixed SELECT and DELETE statements, the LIMIT option to DELETE may help. See Section 13.2.2, “DELETE Statement”.
- Using SQL_BUFFER_RESULT with SELECT statements can help to make the duration of table locks shorter. See Section 13.2.13, “SELECT Statement”.
- Splitting table contents into separate tables may help, by allowing queries to run against columns in one table, while updates are confined to columns in a different table.
- You could change the locking code in mysys/thr_lock.c to use a single queue. In this case, write locks and read locks would have the same priority, which might help some applications.

8.11.3 Concurrent Inserts

The MyISAM storage engine supports concurrent inserts to reduce contention between readers and writers for a given table: If a MyISAM table has no holes in the data file (deleted rows in the middle), an INSERT statement can be executed to add rows to the end of the table at the same time that SELECT statements are reading rows from the table. If there are multiple INSERT statements, they are queued and performed in sequence, concurrently with the SELECT statements. The results of a concurrent INSERT may not be visible immediately.

The concurrent_insert system variable can be set to modify the concurrent-insert processing. By default, the variable is set to AUTO (or 1) and concurrent inserts are handled as just described. If concurrent_insert is set to NEVER (or 0), concurrent inserts are disabled. If the variable is set to ALWAYS (or 2), concurrent inserts at the end of the table are permitted even for tables that have deleted rows. See also the description of the concurrent_insert system variable.

If you are using the binary log, concurrent inserts are converted to normal inserts for `CREATE ... SELECT` or `INSERT ... SELECT` statements. This is done to ensure that you can re-create an exact copy of your tables by applying the log during a backup operation. See Section 5.4.4, “The Binary Log”. In addition, for those statements a read lock is placed on the selected-from table such that inserts into that table are blocked. The effect is that concurrent inserts for that table must wait as well.

With `LOAD DATA`, if you specify `CONCURRENT` with a `MyISAM` table that satisfies the condition for concurrent inserts (that is, it contains no free blocks in the middle), other sessions can retrieve data from the table while `LOAD DATA` is executing. Use of the `CONCURRENT` option affects the performance of `LOAD DATA` a bit, even if no other session is using the table at the same time.

If you specify `HIGH_PRIORITY`, it overrides the effect of the `--low-priority-updates` option if the server was started with that option. It also causes concurrent inserts not to be used.

For `LOCK TABLE`, the difference between `READ LOCAL` and `READ` is that `READ LOCAL` permits nonconflicting `INSERT` statements (concurrent inserts) to execute while the lock is held. However, this cannot be used if you are going to manipulate the database using processes external to the server while you hold the lock.

8.11.4 Metadata Locking

MySQL uses metadata locking to manage concurrent access to database objects and to ensure data consistency. Metadata locking applies not just to tables, but also to schemas, stored programs (procedures, functions, triggers, scheduled events), tablespaces, user locks acquired with the `GET_LOCK()` function (see Section 12.15, “Locking Functions”), and locks acquired with the locking service described in Section 5.6.9.1, “The Locking Service”.

The Performance Schema `metadata_locks` table exposes metadata lock information, which can be useful for seeing which sessions hold locks, are blocked waiting for locks, and so forth. For details, see Section 27.12.13.3, “The metadata_locks Table”.

Metadata locking does involve some overhead, which increases as query volume increases. Metadata contention increases the more that multiple queries attempt to access the same objects.

Metadata locking is not a replacement for the table definition cache, and its mutexes and locks differ from the `LOCK_open` mutex. The following discussion provides some information about how metadata locking works.

- Metadata Lock Acquisition
- Metadata Lock Release

Metadata Lock Acquisition

If there are multiple waiters for a given lock, the highest-priority lock request is satisfied first, with an exception related to the `max_write_lock_count` system variable. Write lock requests have higher priority than read lock requests. However, if `max_write_lock_count` is set to some low value (say, 10), read lock requests may be preferred over pending write lock requests if the read lock requests have already been passed over in favor of 10 write lock requests. Normally this behavior does not occur because `max_write_lock_count` by default has a very large value.

Statements acquire metadata locks one by one, not simultaneously, and perform deadlock detection in the process.

DML statements normally acquire locks in the order in which tables are mentioned in the statement.

DDL statements, `LOCK TABLES`, and other similar statements try to reduce the number of possible deadlocks between concurrent DDL statements by acquiring locks on explicitly named tables in name order. Locks might be acquired in a different order for implicitly used tables (such as tables in foreign key relationships that also must be locked).

For example, `RENAME TABLE` is a DDL statement that acquires locks in name order:

- This `RENAME TABLE` statement renames `tbla` to something else, and renames `tblc` to `tbla`:

  ```
  RENAME TABLE tbla TO tbld, tblc TO tbla;
  ```

  The statement acquires metadata locks, in order, on `tbla`, `tblc`, and `tbld` (because `tbld` follows `tblc` in name order):

- This slightly different statement also renames `tbla` to something else, and renames `tblc` to `tbla`:

  ```
  RENAME TABLE tbla TO tblb, tblc TO tbla;
  ```

  In this case, the statement acquires metadata locks, in order, on `tbla`, `tblb`, and `tblc` (because `tblb` precedes `tblc` in name order):

Both statements acquire locks on `tbla` and `tblc`, in that order, but differ in whether the lock on the remaining table name is acquired before or after `tblc`.

Metadata lock acquisition order can make a difference in operation outcome when multiple transactions execute concurrently, as the following example illustrates.

Begin with two tables `x` and `x_new` that have identical structure. Three clients issue statements that involve these tables:

Client 1:

```
LOCK TABLE x WRITE, x_new WRITE;
```

The statement requests and acquires write locks in name order on `x` and `x_new`.

Client 2:

```
INSERT INTO x VALUES(1);
```

The statement requests and blocks waiting for a write lock on `x`.

Client 3:

```
RENAME TABLE x TO x_old, x_new TO x;
```

The statement requests exclusive locks in name order on `x`, `x_new`, and `x_old`, but blocks waiting for the lock on `x`.

Client 1:

```
UNLOCK TABLES;
```

The statement releases the write locks on `x` and `x_new`. The exclusive lock request for `x` by Client 3 has higher priority than the write lock request by Client 2, so Client 3 acquires its lock on `x`, then also on `x_new` and `x_old`, performs the renaming, and releases its locks. Client 2 then acquires its lock on `x`, performs the insert, and releases its lock.

Lock acquisition order results in the `RENAME TABLE` executing before the `INSERT`. The `x` into which the insert occurs is the table that was named `x_new` when Client 2 issued the insert and was renamed to `x` by Client 3:

```
mysql> SELECT * FROM x;
+------+
| i    |
+------+
|    1 |
+------+

mysql> SELECT * FROM x_old;
Empty set (0.01 sec)
```

Now begin instead with tables named `x` and `new_x` that have identical structure. Again, three clients issue statements that involve these tables:

Client 1:

```
LOCK TABLE x WRITE, new_x WRITE;
```

The statement requests and acquires write locks in name order on `new_x` and `x`.

Client 2:

```
INSERT INTO x VALUES(1);
```

The statement requests and blocks waiting for a write lock on `x`.

Client 3:

```
RENAME TABLE x TO old_x, new_x TO x;
```

The statement requests exclusive locks in name order on `new_x`, `old_x`, and `x`, but blocks waiting for the lock on `new_x`.

Client 1:

```
UNLOCK TABLES;
```

The statement releases the write locks on `x` and `new_x`. For `x`, the only pending request is by Client 2, so Client 2 acquires its lock, performs the insert, and releases the lock. For `new_x`, the only pending request is by Client 3, which is permitted to acquire that lock (and also the lock on `old_x`). The rename operation still blocks for the lock on `x` until the Client 2 insert finishes and releases its lock. Then Client 3 acquires the lock on `x`, performs the rename, and releases its lock.

In this case, lock acquisition order results in the `INSERT` executing before the `RENAME TABLE`. The `x` into which the insert occurs is the original `x`, now renamed to `old_x` by the rename operation:

```
mysql> SELECT * FROM x;
Empty set (0.01 sec)

mysql> SELECT * FROM old_x;
+------+
| i    |
+------+
|    1 |
+------+
```

If order of lock acquisition in concurrent statements makes a difference to an application in operation outcome, as in the preceding example, you may be able to adjust the table names to affect the order of lock acquisition.

Metadata locks are extended, as necessary, to tables related by a foreign key constraint to prevent conflicting DML and DDL operations from executing concurrently on the related tables. When updating a parent table, a metadata lock is taken on the child table while updating foreign key metadata. Foreign key metadata is owned by the child table.

Metadata Lock Release

To ensure transaction serializability, the server must not permit one session to perform a data definition language (DDL) statement on a table that is used in an uncompleted explicitly or implicitly started transaction in another session. The server achieves this by acquiring metadata locks on tables used within a transaction and deferring release of those locks until the transaction ends. A metadata lock on a table prevents changes to the table's structure. This locking approach has the implication that a table that is being used by a transaction within one session cannot be used in DDL statements by other sessions until the transaction ends.

This principle applies not only to transactional tables, but also to nontransactional tables. Suppose that a session begins a transaction that uses transactional table `t` and nontransactional table `nt` as follows:

```
START TRANSACTION;
SELECT * FROM t;
SELECT * FROM nt;
```

The server holds metadata locks on both `t` and `nt` until the transaction ends. If another session attempts a DDL or write lock operation on either table, it blocks until metadata lock release at transaction end. For example, a second session blocks if it attempts any of these operations:

```
DROP TABLE t;
ALTER TABLE t ...;
DROP TABLE nt;
ALTER TABLE nt ...;
LOCK TABLE t ... WRITE;
```

The same behavior applies for The `LOCK TABLES ... READ`. That is, explicitly or implicitly started transactions that update any table (transactional or nontransactional) block and are blocked by `LOCK TABLES ... READ` for that table.

If the server acquires metadata locks for a statement that is syntactically valid but fails during execution, it does not release the locks early. Lock release is still deferred to the end of the transaction because the failed statement is written to the binary log and the locks protect log consistency.

In autocommit mode, each statement is in effect a complete transaction, so metadata locks acquired for the statement are held only to the end of the statement.

Metadata locks acquired during a `PREPARE` statement are released once the statement has been prepared, even if preparation occurs within a multiple-statement transaction.

As of MySQL 8.0.13, for XA transactions in `PREPARED` state, metadata locks are maintained across client disconnects and server restarts, until an `XA COMMIT` or `XA ROLLBACK` is executed.

8.11.5 External Locking

External locking is the use of file system locking to manage contention for `MyISAM` database tables by multiple processes. External locking is used in situations where a single process such as the MySQL server cannot be assumed to be the only process that requires access to tables. Here are some examples:

- If you run multiple servers that use the same database directory (not recommended), each server must have external locking enabled.
- If you use `myisamchk` to perform table maintenance operations on `MyISAM` tables, you must either ensure that the server is not running, or that the server has external locking enabled so that it locks table files as necessary to coordinate with `myisamchk` for access to the tables. The same is true for use of `myisampack` to pack `MyISAM` tables.

  If the server is run with external locking enabled, you can use `myisamchk` at any time for read operations such a checking tables. In this case, if the server tries to update a table that `myisamchk` is using, the server waits for `myisamchk` to finish before it continues.

  If you use `myisamchk` for write operations such as repairing or optimizing tables, or if you use `myisampack` to pack tables, you *must* always ensure that the `mysqld` server is not using the table. If you do not stop `mysqld`, at least do a `mysqladmin flush-tables` before you run `myisamchk`. Your tables *may become corrupted* if the server and `myisamchk` access the tables simultaneously.

With external locking in effect, each process that requires access to a table acquires a file system lock for the table files before proceeding to access the table. If all necessary locks cannot be acquired, the process is blocked from accessing the table until the locks can be obtained (after the process that currently holds the locks releases them).

External locking affects server performance because the server must sometimes wait for other processes before it can access tables.

External locking is unnecessary if you run a single server to access a given data directory (which is the usual case) and if no other programs such as `myisamchk` need to modify tables while the server is running. If you only *read* tables with other programs, external locking is not required, although `myisamchk` might report warnings if the server changes tables while `myisamchk` is reading them.

With external locking disabled, to use `myisamchk`, you must either stop the server while `myisamchk` executes or else lock and flush the tables before running `myisamchk`. To avoid this requirement, use the `CHECK TABLE` and `REPAIR TABLE` statements to check and repair `MyISAM` tables.

For `mysqld`, external locking is controlled by the value of the `skip_external_locking` system variable. When this variable is enabled, external locking is disabled, and vice versa. External locking is disabled by default.

Use of external locking can be controlled at server startup by using the `--external-locking` or `--skip-external-locking` option.

If you do use external locking option to enable updates to `MyISAM` tables from many MySQL processes, do not start the server with the `delay_key_write` system variable set to `ALL` or use the `DELAY_KEY_WRITE=1` table option for any shared tables. Otherwise, index corruption can occur.

The easiest way to satisfy this condition is to always use `--external-locking` together with `--delay-key-write=OFF`. (This is not done by default because in many setups it is useful to have a mixture of the preceding options.)

8.12 Optimizing the MySQL Server

This section discusses optimization techniques for the database server, primarily dealing with system configuration rather than tuning SQL statements. The information in this section is appropriate for DBAs who want to ensure performance and scalability across the servers they manage; for developers constructing installation scripts that include setting up the database; and people running MySQL themselves for development, testing, and so on who want to maximize their own productivity.

8.12.1 Optimizing Disk I/O

This section describes ways to configure storage devices when you can devote more and faster storage hardware to the database server. For information about optimizing an `InnoDB` configuration to improve I/O performance, see Section 8.5.8, “Optimizing InnoDB Disk I/O”.

- Disk seeks are a huge performance bottleneck. This problem becomes more apparent when the amount of data starts to grow so large that effective caching becomes impossible. For large databases where you access data more or less randomly, you can be sure that you need at least one disk seek to read and a couple of disk seeks to write things. To minimize this problem, use disks with low seek times.

- Increase the number of available disk spindles (and thereby reduce the seek overhead) by either symlinking files to different disks or striping the disks:

  - Using symbolic links

    This means that, for `MyISAM` tables, you symlink the index file and data files from their usual location in the data directory to another disk (that may also be striped). This makes both the seek and read times better, assuming that the disk is not used for other purposes as well. See Section 8.12.2, “Using Symbolic Links”.

    Symbolic links are not supported for use with `InnoDB` tables. However, it is possible to place `InnoDB` data and log files on different physical disks. For more information, see Section 8.5.8, “Optimizing InnoDB Disk I/O”.

  - Striping

Striping means that you have many disks and put the first block on the first disk, the second block on the second disk, and the *N*-th block on the (*N* `MOD` *number_of_disks*) disk, and so on. This means if your normal data size is less than the stripe size (or perfectly aligned), you get much better performance. Striping is very dependent on the operating system and the stripe size, so benchmark your application with different stripe sizes. See Section 8.13.2, “Using Your Own Benchmarks”.

The speed difference for striping is *very* dependent on the parameters. Depending on how you set the striping parameters and number of disks, you may get differences measured in orders of magnitude. You have to choose to optimize for random or sequential access.

- For reliability, you may want to use RAID 0+1 (striping plus mirroring), but in this case, you need 2 × *N* drives to hold *N* drives of data. This is probably the best option if you have the money for it. However, you may also have to invest in some volume-management software to handle it efficiently.
- A good option is to vary the RAID level according to how critical a type of data is. For example, store semi-important data that can be regenerated on a RAID 0 disk, but store really important data such as host information and logs on a RAID 0+1 or RAID *N* disk. RAID *N* can be a problem if you have many writes, due to the time required to update the parity bits.
- You can also set the parameters for the file system that the database uses:

  If you do not need to know when files were last accessed (which is not really useful on a database server), you can mount your file systems with the `-o noatime` option. That skips updates to the last access time in inodes on the file system, which avoids some disk seeks.

  On many operating systems, you can set a file system to be updated asynchronously by mounting it with the `-o async` option. If your computer is reasonably stable, this should give you better performance without sacrificing too much reliability. (This flag is on by default on Linux.)

Using NFS with MySQL

You should be cautious when considering whether to use NFS with MySQL. Potential issues, which vary by operating system and NFS version, include the following:

- MySQL data and log files placed on NFS volumes becoming locked and unavailable for use. Locking issues may occur in cases where multiple instances of MySQL access the same data directory or where MySQL is shut down improperly, due to a power outage, for example. NFS version 4 addresses underlying locking issues with the introduction of advisory and lease-based locking. However, sharing a data directory among MySQL instances is not recommended.
- Data inconsistencies introduced due to messages received out of order or lost network traffic. To avoid this issue, use TCP with `hard` and `intr` mount options.
- Maximum file size limitations. NFS Version 2 clients can only access the lowest 2GB of a file (signed 32 bit offset). NFS Version 3 clients support larger files (up to 64 bit offsets). The maximum supported file size also depends on the local file system of the NFS server.

Using NFS within a professional SAN environment or other storage system tends to offer greater reliability than using NFS outside of such an environment. However, NFS within a SAN environment may be slower than directly attached or bus-attached non-rotational storage.

If you choose to use NFS, NFS Version 4 or later is recommended, as is testing your NFS setup thoroughly before deploying into a production environment.

8.12.2 Using Symbolic Links

You can move databases or tables from the database directory to other locations and replace them with symbolic links to the new locations. You might want to do this, for example, to move a database

to a file system with more free space or increase the speed of your system by spreading your tables to different disks.

For `InnoDB` tables, use the `DATA DIRECTORY` clause of the `CREATE TABLE` statement instead of symbolic links, as explained in Section 15.6.1.2, “Creating Tables Externally”. This new feature is a supported, cross-platform technique.

The recommended way to do this is to symlink entire database directories to a different disk. Symlink `MyISAM` tables only as a last resort.

To determine the location of your data directory, use this statement:

```
SHOW VARIABLES LIKE 'datadir';
```

8.12.2.1 Using Symbolic Links for Databases on Unix

On Unix, symlink a database using this procedure:

1. Create the database using `CREATE DATABASE`:

   ```
   mysql> CREATE DATABASE mydb1;
   ```

   Using `CREATE DATABASE` creates the database in the MySQL data directory and permits the server to update the data dictionary with information about the database directory.

2. Stop the server to ensure that no activity occurs in the new database while it is being moved.

3. Move the database directory to some disk where you have free space. For example, use `tar` or `mv`. If you use a method that copies rather than moves the database directory, remove the original database directory after copying it.

4. Create a soft link in the data directory to the moved database directory:

   ```
   $> ln -s /path/to/mydb1 /path/to/datadir
   ```

   The command creates a symlink named `mydb1` in the data directory.

5. Restart the server.

8.12.2.2 Using Symbolic Links for MyISAM Tables on Unix

Note

Symbolic link support as described here, along with the `--symbolic-links` option that controls it, and is deprecated; expect these to be removed in a future version of MySQL. In addition, the option is disabled by default.

Symlinks are fully supported only for `MyISAM` tables. For files used by tables for other storage engines, you may get strange problems if you try to use symbolic links. For `InnoDB` tables, use the alternative technique explained in Section 15.6.1.2, “Creating Tables Externally” instead.

Do not symlink tables on systems that do not have a fully operational `realpath()` call. (Linux and Solaris support `realpath()`). To determine whether your system supports symbolic links, check the value of the `have_symlink` system variable using this statement:

```
SHOW VARIABLES LIKE 'have_symlink';
```

The handling of symbolic links for `MyISAM` tables works as follows:

- In the data directory, you always have the data (`.MYD`) file and the index (`.MYI`) file. The data file and index file can be moved elsewhere and replaced in the data directory by symlinks.
- You can symlink the data file and the index file independently to different directories.

- To instruct a running MySQL server to perform the symlinking, use the `DATA DIRECTORY` and `INDEX DIRECTORY` options to `CREATE TABLE`. See Section 13.1.20, “CREATE TABLE Statement”. Alternatively, if `mysqld` is not running, symlinking can be accomplished manually using `ln -s` from the command line.

  

  Note

  The path used with either or both of the `DATA DIRECTORY` and `INDEX DIRECTORY` options may not include the MySQL `data` directory. (Bug #32167)

- `myisamchk` does not replace a symlink with the data file or index file. It works directly on the file to which the symlink points. Any temporary files are created in the directory where the data file or index file is located. The same is true for the `ALTER TABLE`, `OPTIMIZE TABLE`, and `REPAIR TABLE` statements.

- 

  Note

  When you drop a table that is using symlinks, *both the symlink and the file to which the symlink points are dropped*. This is an extremely good reason *not* to run `mysqld` as the `root` operating system user or permit operating system users to have write access to MySQL database directories.

- If you rename a table with `ALTER TABLE ... RENAME` or `RENAME TABLE` and you do not move the table to another database, the symlinks in the database directory are renamed to the new names and the data file and index file are renamed accordingly.

- If you use `ALTER TABLE ... RENAME` or `RENAME TABLE` to move a table to another database, the table is moved to the other database directory. If the table name changed, the symlinks in the new database directory are renamed to the new names and the data file and index file are renamed accordingly.

- If you are not using symlinks, start `mysqld` with the `--skip-symbolic-links` option to ensure that no one can use `mysqld` to drop or rename a file outside of the data directory.

These table symlink operations are not supported:

- `ALTER TABLE` ignores the `DATA DIRECTORY` and `INDEX DIRECTORY` table options.

8.12.2.3 Using Symbolic Links for Databases on Windows

On Windows, symbolic links can be used for database directories. This enables you to put a database directory at a different location (for example, on a different disk) by setting up a symbolic link to it. Use of database symlinks on Windows is similar to their use on Unix, although the procedure for setting up the link differs.

Suppose that you want to place the database directory for a database named `mydb` at `D:\data\mydb`. To do this, create a symbolic link in the MySQL data directory that points to `D:\data\mydb`. However, before creating the symbolic link, make sure that the `D:\data\mydb` directory exists by creating it if necessary. If you already have a database directory named `mydb` in the data directory, move it to `D:\data`. Otherwise, the symbolic link has no effect. To avoid problems, make sure that the server is not running when you move the database directory.

On Windows, you can create a symlink using the `mklink` command. This command requires administrative privileges.

1. Make sure that the desired path to the database exists. For this example, we use `D:\data\mydb`, and a database named `mydb`.

2. If the database does not already exist, issue `CREATE DATABASE mydb` in the `mysql` client to create it.

3. Stop the MySQL service.
4. Using Windows Explorer or the command line, move the directory `mydb` from the data directory to `D:\data`, replacing the directory of the same name.
5. If you are not already using the command prompt, open it, and change location to the data directory, like this:

   ```
   C:\> cd \path\to\datadir
   ```

   If your MySQL installation is in the default location, you can use this:

   ```
   C:\> cd C:\ProgramData\MySQL\MySQL Server 8.0\Data
   ```

6. In the data directory, create a symlink named `mydb` that points to the location of the database directory:

   ```
   C:\> mklink /d mydb D:\data\mydb
   ```

7. Start the MySQL service.

After this, all tables created in the database `mydb` are created in `D:\data\mydb`.

Alternatively, on any version of Windows supported by MySQL, you can create a symbolic link to a MySQL database by creating a `.sym` file in the data directory that contains the path to the destination directory. The file should be named *`db_name`*`.sym`, where *`db_name`* is the database name.

Support for database symbolic links on Windows using `.sym` files is enabled by default. If you do not need `.sym` file symbolic links, you can disable support for them by starting `mysqld` with the `--skip-symbolic-links` option. To determine whether your system supports `.sym` file symbolic links, check the value of the `have_symlink` system variable using this statement:

```
SHOW VARIABLES LIKE 'have_symlink';
```

To create a `.sym` file symlink, use this procedure:

1. Change location into the data directory:

   ```
   C:\> cd \path\to\datadir
   ```

2. In the data directory, create a text file named `mydb.sym` that contains this path name: `D:\data\mydb\`

Note

The path name to the new database and tables should be absolute. If you specify a relative path, the location is relative to the `mydb.sym` file.

After this, all tables created in the database `mydb` are created in `D:\data\mydb`.

8.12.3 Optimizing Memory Use

8.12.3.1 How MySQL Uses Memory

MySQL allocates buffers and caches to improve performance of database operations. The default configuration is designed to permit a MySQL server to start on a virtual machine that has approximately 512MB of RAM. You can improve MySQL performance by increasing the values of certain cache and buffer-related system variables. You can also modify the default configuration to run MySQL on systems with limited memory.

The following list describes some of the ways that MySQL uses memory. Where applicable, relevant system variables are referenced. Some items are storage engine or feature specific.

- The `InnoDB` buffer pool is a memory area that holds cached `InnoDB` data for tables, indexes, and other auxiliary buffers. For efficiency of high-volume read operations, the buffer pool is divided into pages that can potentially hold multiple rows. For efficiency of cache management, the buffer pool is implemented as a linked list of pages; data that is rarely used is aged out of the cache, using a variation of the LRU algorithm. For more information, see Section 15.5.1, “Buffer Pool”.

  The size of the buffer pool is important for system performance:

  - `InnoDB` allocates memory for the entire buffer pool at server startup, using `malloc()` operations. The `innodb_buffer_pool_size` system variable defines the buffer pool size. Typically, a recommended `innodb_buffer_pool_size` value is 50 to 75 percent of system memory. `innodb_buffer_pool_size` can be configured dynamically, while the server is running. For more information, see Section 15.8.3.1, “Configuring InnoDB Buffer Pool Size”.

  - On systems with a large amount of memory, you can improve concurrency by dividing the buffer pool into multiple buffer pool instances. The `innodb_buffer_pool_instances` system variable defines the number of buffer pool instances.

  - A buffer pool that is too small may cause excessive churning as pages are flushed from the buffer pool only to be required again a short time later.

  - A buffer pool that is too large may cause swapping due to competition for memory.

- The storage engine interface enables the optimizer to provide information about the size of the record buffer to be used for scans that the optimizer estimates are likely to read multiple rows. The buffer size can vary based on the size of the estimate. `InnoDB` uses this variable-size buffering capability to take advantage of row prefetching, and to reduce the overhead of latching and B-tree navigation.

- All threads share the `MyISAM` key buffer. The `key_buffer_size` system variable determines its size.

  For each `MyISAM` table the server opens, the index file is opened once; the data file is opened once for each concurrently running thread that accesses the table. For each concurrent thread, a table structure, column structures for each column, and a buffer of size `3 * N` are allocated (where *`N`* is the maximum row length, not counting `BLOB` columns). A `BLOB` column requires five to eight bytes plus the length of the `BLOB` data. The `MyISAM` storage engine maintains one extra row buffer for internal use.

- The `myisam_use_mmap` system variable can be set to 1 to enable memory-mapping for all `MyISAM` tables.

- If an internal in-memory temporary table becomes too large (as determined using the `tmp_table_size` and `max_heap_table_size` system variables), MySQL automatically converts the table from in-memory to on-disk format. As of MySQL 8.0.16, on-disk temporary tables always use the `InnoDB` storage engine. (Previously, the storage engine employed for this purpose was determined by the `internal_tmp_disk_storage_engine` system variable, which is no longer supported.) You can increase the permissible temporary table size as described in Section 8.4.4, “Internal Temporary Table Use in MySQL”.

  For `MEMORY` tables explicitly created with `CREATE TABLE`, only the `max_heap_table_size` system variable determines how large a table can grow, and there is no conversion to on-disk format.

- The MySQL Performance Schema is a feature for monitoring MySQL server execution at a low level. The Performance Schema dynamically allocates memory incrementally, scaling its memory use to actual server load, instead of allocating required memory during server startup. Once memory is allocated, it is not freed until the server is restarted. For more information, see Section 27.17, “The Performance Schema Memory-Allocation Model”.

- Each thread that the server uses to manage client connections requires some thread-specific space. The following list indicates these and which system variables control their size:

  - A stack (`thread_stack`)
  - A connection buffer (`net_buffer_length`)
  - A result buffer (`net_buffer_length`)

  The connection buffer and result buffer each begin with a size equal to `net_buffer_length` bytes, but are dynamically enlarged up to `max_allowed_packet` bytes as needed. The result buffer shrinks to `net_buffer_length` bytes after each SQL statement. While a statement is running, a copy of the current statement string is also allocated.

  Each connection thread uses memory for computing statement digests. The server allocates `max_digest_length` bytes per session. See Section 27.10, “Performance Schema Statement Digests and Sampling”.

- All threads share the same base memory.
- When a thread is no longer needed, the memory allocated to it is released and returned to the system unless the thread goes back into the thread cache. In that case, the memory remains allocated.
- Each request that performs a sequential scan of a table allocates a *read buffer*. The `read_buffer_size` system variable determines the buffer size.
- When reading rows in an arbitrary sequence (for example, following a sort), a *random-read buffer* may be allocated to avoid disk seeks. The `read_rnd_buffer_size` system variable determines the buffer size.
- All joins are executed in a single pass, and most joins can be done without even using a temporary table. Most temporary tables are memory-based hash tables. Temporary tables with a large row length (calculated as the sum of all column lengths) or that contain `BLOB` columns are stored on disk.
- Most requests that perform a sort allocate a sort buffer and zero to two temporary files depending on the result set size. See Section B.3.3.5, “Where MySQL Stores Temporary Files”.
- Almost all parsing and calculating is done in thread-local and reusable memory pools. No memory overhead is needed for small items, thus avoiding the normal slow memory allocation and freeing. Memory is allocated only for unexpectedly large strings.
- For each table having `BLOB` columns, a buffer is enlarged dynamically to read in larger `BLOB` values. If you scan a table, the buffer grows as large as the largest `BLOB` value.
- MySQL requires memory and descriptors for the table cache. Handler structures for all in-use tables are saved in the table cache and managed as “First In, First Out” (FIFO). The `table_open_cache` system variable defines the initial table cache size; see Section 8.4.3.1, “How MySQL Opens and Closes Tables”.

  MySQL also requires memory for the table definition cache. The `table_definition_cache` system variable defines the number of table definitions that can be stored in the table definition cache. If you use a large number of tables, you can create a large table definition cache to speed up the opening of tables. The table definition cache takes less space and does not use file descriptors, unlike the table cache.

- A `FLUSH TABLES` statement or `mysqladmin flush-tables` command closes all tables that are not in use at once and marks all in-use tables to be closed when the currently executing thread finishes. This effectively frees most in-use memory. `FLUSH TABLES` does not return until all tables have been closed.
- The server caches information in memory as a result of `GRANT`, `CREATE USER`, `CREATE SERVER`, and `INSTALL PLUGIN` statements. This memory is not released by the corresponding `REVOKE`,

`DROP USER`, `DROP SERVER`, and `UNINSTALL PLUGIN` statements, so for a server that executes many instances of the statements that cause caching, there is an increase in cached memory use unless it is freed with `FLUSH PRIVILEGES`.

- In a replication topology, the following settings affect memory usage, and can be adjusted as required:
  - The `max_allowed_packet` system variable on a replication source limits the maximum message size that the source sends to its replicas for processing. This setting defaults to 64M.
  - The system variable `replica_pending_jobs_size_max` (from MySQL 8.0.26) or `slave_pending_jobs_size_max` (before MySQL 8.0.26) on a multithreaded replica sets the maximum amount of memory that is made available for holding messages awaiting processing. This setting defaults to 128M. The memory is only allocated when needed, but it might be used if your replication topology handles large transactions sometimes. It is a soft limit, and larger transactions can be processed.
  - The `rpl_read_size` system variable on a replication source or replica controls the minimum amount of data in bytes that is read from the binary log files and relay log files. The default is 8192 bytes. A buffer the size of this value is allocated for each thread that reads from the binary log and relay log files, including dump threads on sources and coordinator threads on replicas.
  - The `binlog_transaction_dependency_history_size` system variable limits the number of row hashes held as an in-memory history.
  - The `max_binlog_cache_size` system variable specifies the upper limit of memory usage by an individual transaction.
  - The `max_binlog_stmt_cache_size` system variable specifies the upper limit of memory usage by the statement cache.

`ps` and other system status programs may report that `mysqld` uses a lot of memory. This may be caused by thread stacks on different memory addresses. For example, the Solaris version of `ps` counts the unused memory between stacks as used memory. To verify this, check available swap with `swap -s`. We test `mysqld` with several memory-leakage detectors (both commercial and Open Source), so there should be no memory leaks.

8.12.3.2 Monitoring MySQL Memory Usage

The following example demonstrates how to use Performance Schema and sys schema to monitor MySQL memory usage.

Most Performance Schema memory instrumentation is disabled by default. Instruments can be enabled by updating the `ENABLED` column of the Performance Schema `setup_instruments` table. Memory instruments have names in the form of `memory/code_area/instrument_name`, where *`code_area`* is a value such as `sql` or `innodb`, and *`instrument_name`* is the instrument detail.

1. To view available MySQL memory instruments, query the Performance Schema `setup_instruments` table. The following query returns hundreds of memory instruments for all code areas.

```
mysql> SELECT * FROM performance_schema.setup_instruments
       WHERE NAME LIKE '%memory%';
```

   You can narrow results by specifying a code area. For example, you can limit results to `InnoDB` memory instruments by specifying `innodb` as the code area.

```
mysql> SELECT * FROM performance_schema.setup_instruments
       WHERE NAME LIKE '%memory/innodb%';
+----------------------------------------------+---------+-------+
| NAME                                         | ENABLED | TIMED |
+----------------------------------------------+---------+-------+
```

```
| memory/innodb/adaptive hash index          | NO      | NO    |
| memory/innodb/buf_buf_pool                 | NO      | NO    |
| memory/innodb/dict_stats_bg_recalc_pool_t | NO      | NO    |
| memory/innodb/dict_stats_index_map_t       | NO      | NO    |
| memory/innodb/dict_stats_n_diff_on_level   | NO      | NO    |
| memory/innodb/other                        | NO      | NO    |
| memory/innodb/row_log_buf                  | NO      | NO    |
| memory/innodb/row_merge_sort               | NO      | NO    |
| memory/innodb/std                          | NO      | NO    |
| memory/innodb/trx_sys_t::rw_trx_ids        | NO      | NO    |
...
```

Depending on your MySQL installation, code areas may include `performance_schema`, `sql`, `client`, `innodb`, `myisam`, `csv`, `memory`, `blackhole`, `archive`, `partition`, and others.

2. To enable memory instruments, add a `performance-schema-instrument` rule to your MySQL configuration file. For example, to enable all memory instruments, add this rule to your configuration file and restart the server:

```
performance-schema-instrument='memory/%=COUNTED'
```

Note

Enabling memory instruments at startup ensures that memory allocations that occur at startup are counted.

After restarting the server, the `ENABLED` column of the Performance Schema `setup_instruments` table should report `YES` for memory instruments that you enabled. The `TIMED` column in the `setup_instruments` table is ignored for memory instruments because memory operations are not timed.

```
mysql> SELECT * FROM performance_schema.setup_instruments
       WHERE NAME LIKE '%memory/innodb%';
+-------------------------------------------+---------+-------+
| NAME                                      | ENABLED | TIMED |
+-------------------------------------------+---------+-------+
| memory/innodb/adaptive hash index         | NO      | NO    |
| memory/innodb/buf_buf_pool                | NO      | NO    |
| memory/innodb/dict_stats_bg_recalc_pool_t | NO      | NO    |
| memory/innodb/dict_stats_index_map_t      | NO      | NO    |
| memory/innodb/dict_stats_n_diff_on_level  | NO      | NO    |
| memory/innodb/other                       | NO      | NO    |
| memory/innodb/row_log_buf                 | NO      | NO    |
| memory/innodb/row_merge_sort              | NO      | NO    |
| memory/innodb/std                         | NO      | NO    |
| memory/innodb/trx_sys_t::rw_trx_ids       | NO      | NO    |
...
```

3. Query memory instrument data. In this example, memory instrument data is queried in the Performance Schema `memory_summary_global_by_event_name` table, which summarizes data by `EVENT_NAME`. The `EVENT_NAME` is the name of the instrument.

   The following query returns memory data for the `InnoDB` buffer pool. For column descriptions, see Section 27.12.20.10, “Memory Summary Tables”.

```
mysql> SELECT * FROM performance_schema.memory_summary_global_by_event_name
       WHERE EVENT_NAME LIKE 'memory/innodb/buf_buf_pool'\G
                 EVENT_NAME: memory/innodb/buf_buf_pool
                COUNT_ALLOC: 1
                 COUNT_FREE: 0
   SUM_NUMBER_OF_BYTES_ALLOC: 137428992
    SUM_NUMBER_OF_BYTES_FREE: 0
              LOW_COUNT_USED: 0
          CURRENT_COUNT_USED: 1
             HIGH_COUNT_USED: 1
    LOW_NUMBER_OF_BYTES_USED: 0
CURRENT_NUMBER_OF_BYTES_USED: 137428992
   HIGH_NUMBER_OF_BYTES_USED: 137428992
```

The same underlying data can be queried using the `sys` schema `memory_global_by_current_bytes` table, which shows current memory usage within the server globally, broken down by allocation type.

```
mysql> SELECT * FROM sys.memory_global_by_current_bytes
       WHERE event_name LIKE 'memory/innodb/buf_buf_pool'\G
*************************** 1. row ***************************
       event_name: memory/innodb/buf_buf_pool
    current_count: 1
    current_alloc: 131.06 MiB
current_avg_alloc: 131.06 MiB
       high_count: 1
       high_alloc: 131.06 MiB
   high_avg_alloc: 131.06 MiB
```

This `sys` schema query aggregates currently allocated memory (`current_alloc`) by code area:

```
mysql> SELECT SUBSTRING_INDEX(event_name,'/',2) AS
       code_area, FORMAT_BYTES(SUM(current_alloc))
       AS current_alloc
       FROM sys.x$memory_global_by_current_bytes
       GROUP BY SUBSTRING_INDEX(event_name,'/',2)
       ORDER BY SUM(current_alloc) DESC;
+---------------------------+---------------+
| code_area                 | current_alloc |
+---------------------------+---------------+
| memory/innodb             | 843.24 MiB    |
| memory/performance_schema | 81.29 MiB     |
| memory/mysys              | 8.20 MiB      |
| memory/sql                | 2.47 MiB      |
| memory/memory             | 174.01 KiB    |
| memory/myisam             | 46.53 KiB     |
| memory/blackhole          | 512 bytes     |
| memory/federated          | 512 bytes     |
| memory/csv                | 512 bytes     |
| memory/vio                | 496 bytes     |
+---------------------------+---------------+
```

Note

Prior to MySQL 8.0.16, `sys.format_bytes()` was used for `FORMAT_BYTES()`.

For more information about `sys` schema, see Chapter 28, *MySQL sys Schema*.

8.12.3.3 Enabling Large Page Support

Some hardware and operating system architectures support memory pages greater than the default (usually 4KB). The actual implementation of this support depends on the underlying hardware and operating system. Applications that perform a lot of memory accesses may obtain performance improvements by using large pages due to reduced Translation Lookaside Buffer (TLB) misses.

In MySQL, large pages can be used by `InnoDB`, to allocate memory for its buffer pool and additional memory pool.

Standard use of large pages in MySQL attempts to use the largest size supported, up to 4MB. Under Solaris, a “super large pages” feature enables uses of pages up to 256MB. This feature is available for recent SPARC platforms. It can be enabled or disabled by using the `--super-large-pages` or `--skip-super-large-pages` option.

MySQL also supports the Linux implementation of large page support (which is called HugeTLB in Linux).

Before large pages can be used on Linux, the kernel must be enabled to support them and it is necessary to configure the HugeTLB memory pool. For reference, the HugeTBL API is documented in the `Documentation/vm/hugetlbpage.txt` file of your Linux sources.

The kernels for some recent systems such as Red Hat Enterprise Linux may have the large pages feature enabled by default. To check whether this is true for your kernel, use the following command and look for output lines containing “huge”:

```
$> grep -i huge /proc/meminfo
AnonHugePages:   2658304 kB
ShmemHugePages:        0 kB
HugePages_Total:       0
HugePages_Free:        0
HugePages_Rsvd:        0
HugePages_Surp:        0
Hugepagesize:       2048 kB
Hugetlb:               0 kB
```

The nonempty command output indicates that large page support is present, but the zero values indicate that no pages are configured for use.

If your kernel needs to be reconfigured to support large pages, consult the `hugetlbpage.txt` file for instructions.

Assuming that your Linux kernel has large page support enabled, configure it for use by MySQL using the following steps:

1. Determine the number of large pages needed. This is the size of the InnoDB buffer pool divided by the large page size, which we can calculate as `innodb_buffer_pool_size` / `Hugepagesize`. Assuming the default value for the buffer pool size and using the `Hugepagesize` value obtained from `/proc/meminfo`, this is 134217728 / 2048, or 65536 (64K). We call this value *P*.

2. As system root, open the file `/etc/sysctl.conf` in a text editor, and add the line shown here, where *P* is the number of large pages obtained in the previous step:

   ```
   vm.nr_hugepages=P
   ```

   Using the actual value obtained previously, the additional line should look like this:

   ```
   vm.nr_huge_pages=65536
   ```

   Save the updated file.

3. As system root, run the following command:

   ```
   $> sudo sysctl -p
   ```

   

   Note

   On some systems the large pages file may be named slightly differently; for example, some distributions call it `nr_hugepages`. In the event `sysctl` returns an error relating to the file name, check the name of the corresponding file in `/proc/sys/vm` and use that instead.

   To verify the large page configuration, check `/proc/meminfo` again as described previously. Now you should see some additional nonzero values in the output, similar to this:

   ```
   $> grep -i huge /proc/meminfo
   AnonHugePages:   2686976 kB
   ShmemHugePages:        0 kB
   HugePages_Total:     233
   HugePages_Free:      233
   HugePages_Rsvd:        0
   HugePages_Surp:        0
   Hugepagesize:       2048 kB
   Hugetlb:          477184 kB
   ```

4. Optionally, you may wish to compact the Linux VM. You can do this using a sequence of commands, possibly in a script file, similar to what is shown here:

```
sync
sync
sync
echo 3 > /proc/sys/vm/drop_caches
echo 1 > /proc/sys/vm/compact_memory
```

See your operating platform documentation for more information about how to do this.

5. Check any configuration files such as `my.cnf` used by the server, and make sure that `innodb_buffer_pool_chunk_size` is set larger than the huge page size. The default for this variable is 128M.

6. Large page support in the MySQL server is disabled by default. To enable it, start the server with `--large-pages`. You can also do so by adding the following line to the `[mysqld]` section of the server `my.cnf` file:

   ```
   large-pages=ON
   ```

   With this option enabled, `InnoDB` uses large pages automatically for its buffer pool and additional memory pool. If `InnoDB` cannot do this, it falls back to use of traditional memory and writes a warning to the error log: `Warning: Using conventional memory pool`.

You can verify that MySQL is now using large pages by checking `/proc/meminfo` again after restarting `mysqld`, like this:

```
$> grep -i huge /proc/meminfo
AnonHugePages:   2516992 kB
ShmemHugePages:        0 kB
HugePages_Total:     233
HugePages_Free:      222
HugePages_Rsvd:       55
HugePages_Surp:        0
Hugepagesize:       2048 kB
Hugetlb:          477184 kB
```

8.13 Measuring Performance (Benchmarking)

To measure performance, consider the following factors:

- Whether you are measuring the speed of a single operation on a quiet system, or how a set of operations (a “workload”) works over a period of time. With simple tests, you usually test how changing one aspect (a configuration setting, the set of indexes on a table, the SQL clauses in a query) affects performance. Benchmarks are typically long-running and elaborate performance tests, where the results could dictate high-level choices such as hardware and storage configuration, or how soon to upgrade to a new MySQL version.
- For benchmarking, sometimes you must simulate a heavy database workload to get an accurate picture.
- Performance can vary depending on so many different factors that a difference of a few percentage points might not be a decisive victory. The results might shift the opposite way when you test in a different environment.
- Certain MySQL features help or do not help performance depending on the workload. For completeness, always test performance with those features turned on and turned off. The most important feature to try with each workload is the adaptive hash index for `InnoDB` tables.

This section progresses from simple and direct measurement techniques that a single developer can do, to more complicated ones that require additional expertise to perform and interpret the results.

8.13.1 Measuring the Speed of Expressions and Functions

To measure the speed of a specific MySQL expression or function, invoke the `BENCHMARK()` function using the `mysql` client program. Its syntax is `BENCHMARK(loop_count,expr)`. The return value is

always zero, but `mysql` prints a line displaying approximately how long the statement took to execute. For example:

```
mysql> SELECT BENCHMARK(1000000,1+1);
+------------------------+
| BENCHMARK(1000000,1+1) |
+------------------------+
|                      0 |
+------------------------+
1 row in set (0.32 sec)
```

This result was obtained on a Pentium II 400MHz system. It shows that MySQL can execute 1,000,000 simple addition expressions in 0.32 seconds on that system.

The built-in MySQL functions are typically highly optimized, but there may be some exceptions. `BENCHMARK()` is an excellent tool for finding out if some function is a problem for your queries.

8.13.2 Using Your Own Benchmarks

Benchmark your application and database to find out where the bottlenecks are. After fixing one bottleneck (or by replacing it with a “dummy” module), you can proceed to identify the next bottleneck. Even if the overall performance for your application currently is acceptable, you should at least make a plan for each bottleneck and decide how to solve it if someday you really need the extra performance.

A free benchmark suite is the Open Source Database Benchmark, available at http://osdb.sourceforge.net/.

It is very common for a problem to occur only when the system is very heavily loaded. We have had many customers who contact us when they have a (tested) system in production and have encountered load problems. In most cases, performance problems turn out to be due to issues of basic database design (for example, table scans are not good under high load) or problems with the operating system or libraries. Most of the time, these problems would be much easier to fix if the systems were not already in production.

To avoid problems like this, benchmark your whole application under the worst possible load:

- The `mysqlslap` program can be helpful for simulating a high load produced by multiple clients issuing queries simultaneously. See Section 4.5.8, “mysqlslap — A Load Emulation Client”.
- You can also try benchmarking packages such as SysBench and DBT2, available at https://launchpad.net/sysbench, and http://osdldbt.sourceforge.net/#dbt2.

These programs or packages can bring a system to its knees, so be sure to use them only on your development systems.

8.13.3 Measuring Performance with performance_schema

You can query the tables in the `performance_schema` database to see real-time information about the performance characteristics of your server and the applications it is running. See Chapter 27, *MySQL Performance Schema* for details.

8.14 Examining Server Thread (Process) Information

To ascertain what your MySQL server is doing, it can be helpful to examine the process list, which indicates the operations currently being performed by the set of threads executing within the server. For example:

```
mysql> SHOW PROCESSLIST\G
*************************** 1. row ***************************
     Id: 5
   User: event_scheduler
   Host: localhost
     db: NULL
```

```
Command: Daemon
   Time: 2756681
  State: Waiting on empty queue
   Info: NULL
*************************** 2. row ***************************
     Id: 20
   User: me
   Host: localhost:52943
     db: test
Command: Query
   Time: 0
  State: starting
   Info: SHOW PROCESSLIST
```

Threads can be killed with the `KILL` statement. See Section 13.7.8.4, “KILL Statement”.

8.14.1 Accessing the Process List

The following discussion enumerates the sources of process information, the privileges required to see process information, and describes the content of process list entries.

- Sources of Process Information
- Privileges Required to Access the Process List
- Content of Process List Entries

Sources of Process Information

Process information is available from these sources:

- The `SHOW PROCESSLIST` statement: Section 13.7.7.29, “SHOW PROCESSLIST Statement”
- The `mysqladmin processlist` command: Section 4.5.2, “mysqladmin — A MySQL Server Administration Program”
- The `INFORMATION_SCHEMA PROCESSLIST` table: Section 26.3.23, “The INFORMATION_SCHEMA PROCESSLIST Table”
- The Performance Schema `processlist` table: Section 27.12.21.6, “The processlist Table”
- The Performance Schema `threads` table columns with names having a prefix of `PROCESSLIST_`: Section 27.12.21.7, “The threads Table”
- The `sys` schema `processlist` and `session` views: Section 28.4.3.22, “The processlist and x$processlist Views”, and Section 28.4.3.33, “The session and x$session Views”

The `threads` table compares to `SHOW PROCESSLIST`, `INFORMATION_SCHEMA PROCESSLIST`, and `mysqladmin processlist` as follows:

- Access to the `threads` table does not require a mutex and has minimal impact on server performance. The other sources have negative performance consequences because they require a mutex.

  

  Note

  As of MySQL 8.0.22, an alternative implementation for `SHOW PROCESSLIST` is available based on the Performance Schema `processlist` table, which, like the `threads` table, does not require a mutex and has better performance characteristics. For details, see Section 27.12.21.6, “The processlist Table”.

- The `threads` table displays background threads, which the other sources do not. It also provides additional information for each thread that the other sources do not, such as whether the thread is a foreground or background thread, and the location within the server associated with the thread. This means that the `threads` table can be used to monitor thread activity the other sources cannot.

- You can enable or disable Performance Schema thread monitoring, as described in Section 27.12.21.7, “The threads Table”.

For these reasons, DBAs who perform server monitoring using one of the other thread information sources may wish to monitor using the `threads` table instead.

The `sys` schema `processlist` view presents information from the Performance Schema `threads` table in a more accessible format. The `sys` schema `session` view presents information about user sessions like the `sys` schema `processlist` view, but with background processes filtered out.

Privileges Required to Access the Process List

For most sources of process information, if you have the `PROCESS` privilege, you can see all threads, even those belonging to other users. Otherwise (without the `PROCESS` privilege), nonanonymous users have access to information about their own threads but not threads for other users, and anonymous users have no access to thread information.

The Performance Schema `threads` table also provides thread information, but table access uses a different privilege model. See Section 27.12.21.7, “The threads Table”.

Content of Process List Entries

Each process list entry contains several pieces of information. The following list describes them using the labels from `SHOW PROCESSLIST` output. Other process information sources use similar labels.

- `Id` is the connection identifier for the client associated with the thread.
- `User` and `Host` indicate the account associated with the thread.
- `db` is the default database for the thread, or `NULL` if none has been selected.
- `Command` and `State` indicate what the thread is doing.

  Most states correspond to very quick operations. If a thread stays in a given state for many seconds, there might be a problem that needs to be investigated.

  The following sections list the possible `Command` values, and `State` values grouped by category. The meaning for some of these values is self-evident. For others, additional description is provided.

  

  Note

  Applications that examine process list information should be aware that the commands and states are subject to change.

- `Time` indicates how long the thread has been in its current state. The thread's notion of the current time may be altered in some cases: The thread can change the time with `SET TIMESTAMP = value`. For a replica SQL thread, the value is the number of seconds between the timestamp of the last replicated event and the real time of the replica host. See Section 17.2.3, “Replication Threads”.
- `Info` indicates the statement the thread is executing, or `NULL` if it is executing no statement. For `SHOW PROCESSLIST`, this value contains only the first 100 characters of the statement. To see complete statements, use `SHOW FULL PROCESSLIST` (or query a different process information source).

8.14.2 Thread Command Values

A thread can have any of the following `Command` values:

- `Binlog Dump`

  This is a thread on a replication source for sending binary log contents to a replica.

- `Change user`

The thread is executing a change user operation.

- `Close stmt`

  The thread is closing a prepared statement.

- `Connect`

  Used by replication receiver threads connected to the source, and by replication worker threads.

- `Connect Out`

  A replica is connecting to its source.

- `Create DB`

  The thread is executing a create database operation.

- `Daemon`

  This thread is internal to the server, not a thread that services a client connection.

- `Debug`

  The thread is generating debugging information.

- `Delayed insert`

  The thread is a delayed insert handler.

- `Drop DB`

  The thread is executing a drop database operation.

- `Error`

- `Execute`

  The thread is executing a prepared statement.

- `Fetch`

  The thread is fetching the results from executing a prepared statement.

- `Field List`

  The thread is retrieving information for table columns.

- `Init DB`

  The thread is selecting a default database.

- `Kill`

  The thread is killing another thread.

- `Long Data`

  The thread is retrieving long data in the result of executing a prepared statement.

- `Ping`

  The thread is handling a server ping request.

- `Prepare`

The thread is preparing a prepared statement.

- `Processlist`

  The thread is producing information about server threads.

- `Query`

  Employed for user clients while executing queries by single-threaded replication applier threads, as well as by the replication coordinator thread.

- `Quit`

  The thread is terminating.

- `Refresh`

  The thread is flushing table, logs, or caches, or resetting status variable or replication server information.

- `Register Slave`

  The thread is registering a replica server.

- `Reset stmt`

  The thread is resetting a prepared statement.

- `Set option`

  The thread is setting or resetting a client statement execution option.

- `Shutdown`

  The thread is shutting down the server.

- `Sleep`

  The thread is waiting for the client to send a new statement to it.

- `Statistics`

  The thread is producing server status information.

- `Time`

  Unused.

8.14.3 General Thread States

The following list describes thread `State` values that are associated with general query processing and not more specialized activities such as replication. Many of these are useful only for finding bugs in the server.

- `After create`

  This occurs when the thread creates a table (including internal temporary tables), at the end of the function that creates the table. This state is used even if the table could not be created due to some error.

- `altering table`

  The server is in the process of executing an in-place `ALTER TABLE`.

- `Analyzing`

  The thread is calculating a `MyISAM` table key distributions (for example, for `ANALYZE TABLE`).

- `checking permissions`

  The thread is checking whether the server has the required privileges to execute the statement.

- `Checking table`

  The thread is performing a table check operation.

- `cleaning up`

  The thread has processed one command and is preparing to free memory and reset certain state variables.

- `closing tables`

  The thread is flushing the changed table data to disk and closing the used tables. This should be a fast operation. If not, verify that you do not have a full disk and that the disk is not in very heavy use.

- `committing alter table to storage engine`

  The server has finished an in-place `ALTER TABLE` and is committing the result.

- `converting HEAP to ondisk`

  The thread is converting an internal temporary table from a `MEMORY` table to an on-disk table.

- `copy to tmp table`

  The thread is processing an `ALTER TABLE` statement. This state occurs after the table with the new structure has been created but before rows are copied into it.

  For a thread in this state, the Performance Schema can be used to obtain about the progress of the copy operation. See Section 27.12.5, “Performance Schema Stage Event Tables”.

- `Copying to group table`

  If a statement has different `ORDER BY` and `GROUP BY` criteria, the rows are sorted by group and copied to a temporary table.

- `Copying to tmp table`

  The server is copying to a temporary table in memory.

- `Copying to tmp table on disk`

  The server is copying to a temporary table on disk. The temporary result set has become too large (see Section 8.4.4, “Internal Temporary Table Use in MySQL”). Consequently, the thread is changing the temporary table from in-memory to disk-based format to save memory.

- `Creating index`

  The thread is processing `ALTER TABLE ... ENABLE KEYS` for a `MyISAM` table.

- `Creating sort index`

  The thread is processing a `SELECT` that is resolved using an internal temporary table.

- `creating table`

  The thread is creating a table. This includes creation of temporary tables.

- `Creating tmp table`

  The thread is creating a temporary table in memory or on disk. If the table is created in memory but later is converted to an on-disk table, the state during that operation is `Copying to tmp table on disk`.

- `deleting from main table`

  The server is executing the first part of a multiple-table delete. It is deleting only from the first table, and saving columns and offsets to be used for deleting from the other (reference) tables.

- `deleting from reference tables`

  The server is executing the second part of a multiple-table delete and deleting the matched rows from the other tables.

- `discard_or_import_tablespace`

  The thread is processing an `ALTER TABLE ... DISCARD TABLESPACE` or `ALTER TABLE ... IMPORT TABLESPACE` statement.

- `end`

  This occurs at the end but before the cleanup of `ALTER TABLE`, `CREATE VIEW`, `DELETE`, `INSERT`, `SELECT`, or `UPDATE` statements.

  For the `end` state, the following operations could be happening:

  - Writing an event to the binary log
  - Freeing memory buffers, including for blobs

- `executing`

  The thread has begun executing a statement.

- `Execution of init_command`

  The thread is executing statements in the value of the `init_command` system variable.

- `freeing items`

  The thread has executed a command. This state is usually followed by `cleaning up`.

- `FULLTEXT initialization`

  The server is preparing to perform a natural-language full-text search.

- `init`

  This occurs before the initialization of `ALTER TABLE`, `DELETE`, `INSERT`, `SELECT`, or `UPDATE` statements. Actions taken by the server in this state include flushing the binary log and the `InnoDB` log.

- `Killed`

  Someone has sent a `KILL` statement to the thread and it should abort next time it checks the kill flag. The flag is checked in each major loop in MySQL, but in some cases it might still take a short time for the thread to die. If the thread is locked by some other thread, the kill takes effect as soon as the other thread releases its lock.

- `Locking system tables`

  The thread is trying to lock a system table (for example, a time zone or log table).

- `logging slow query`

  The thread is writing a statement to the slow-query log.

- `login`

  The initial state for a connection thread until the client has been authenticated successfully.

- `manage keys`

  The server is enabling or disabling a table index.

- `Opening system tables`

  The thread is trying to open a system table (for example, a time zone or log table).

- `Opening tables`

  The thread is trying to open a table. This is should be very fast procedure, unless something prevents opening. For example, an `ALTER TABLE` or a `LOCK TABLE` statement can prevent opening a table until the statement is finished. It is also worth checking that your `table_open_cache` value is large enough.

  For system tables, the `Opening system tables` state is used instead.

- `optimizing`

  The server is performing initial optimizations for a query.

- `preparing`

  This state occurs during query optimization.

- `preparing for alter table`

  The server is preparing to execute an in-place `ALTER TABLE`.

- `Purging old relay logs`

  The thread is removing unneeded relay log files.

- `query end`

  This state occurs after processing a query but before the `freeing items` state.

- `Receiving from client`

  The server is reading a packet from the client.

- `Removing duplicates`

  The query was using `SELECT DISTINCT` in such a way that MySQL could not optimize away the distinct operation at an early stage. Because of this, MySQL requires an extra stage to remove all duplicated rows before sending the result to the client.

- `removing tmp table`

  The thread is removing an internal temporary table after processing a `SELECT` statement. This state is not used if no temporary table was created.

- `rename`

  The thread is renaming a table.

- `rename result table`

  The thread is processing an `ALTER TABLE` statement, has created the new table, and is renaming it to replace the original table.

- `Reopen tables`

  The thread got a lock for the table, but noticed after getting the lock that the underlying table structure changed. It has freed the lock, closed the table, and is trying to reopen it.

- `Repair by sorting`

  The repair code is using a sort to create indexes.

- `Repair done`

  The thread has completed a multithreaded repair for a `MyISAM` table.

- `Repair with keycache`

  The repair code is using creating keys one by one through the key cache. This is much slower than `Repair by sorting`.

- `Rolling back`

  The thread is rolling back a transaction.

- `Saving state`

  For `MyISAM` table operations such as repair or analysis, the thread is saving the new table state to the `.MYI` file header. State includes information such as number of rows, the `AUTO_INCREMENT` counter, and key distributions.

- `Searching rows for update`

  The thread is doing a first phase to find all matching rows before updating them. This has to be done if the `UPDATE` is changing the index that is used to find the involved rows.

- `Sending data`

  Prior to MySQL 8.0.17: The thread is reading and processing rows for a `SELECT` statement, and sending data to the client. Because operations occurring during this state tend to perform large amounts of disk access (reads), it is often the longest-running state over the lifetime of a given query. *MySQL 8.0.17 and later*: This state is no longer indicated separately, but rather is included in the `Executing` state.

- `Sending to client`

  The server is writing a packet to the client.

- `setup`

  The thread is beginning an `ALTER TABLE` operation.

- `Sorting for group`

  The thread is doing a sort to satisfy a `GROUP BY`.

- `Sorting for order`

  The thread is doing a sort to satisfy an `ORDER BY`.

- `Sorting index`

The thread is sorting index pages for more efficient access during a `MyISAM` table optimization operation.

- `Sorting result`

  For a `SELECT` statement, this is similar to `Creating sort index`, but for nontemporary tables.

- `starting`

  The first stage at the beginning of statement execution.

- `statistics`

  The server is calculating statistics to develop a query execution plan. If a thread is in this state for a long time, the server is probably disk-bound performing other work.

- `System lock`

  The thread has called `mysql_lock_tables()` and the thread state has not been updated since. This is a very general state that can occur for many reasons.

  For example, the thread is going to request or is waiting for an internal or external system lock for the table. This can occur when `InnoDB` waits for a table-level lock during execution of `LOCK TABLES`. If this state is being caused by requests for external locks and you are not using multiple `mysqld` servers that are accessing the same `MyISAM` tables, you can disable external system locks with the `--skip-external-locking` option. However, external locking is disabled by default, so it is likely that this option has no effect. For `SHOW PROFILE`, this state means the thread is requesting the lock (not waiting for it).

  For system tables, the `Locking system tables` state is used instead.

- `update`

  The thread is getting ready to start updating the table.

- `Updating`

  The thread is searching for rows to update and is updating them.

- `updating main table`

  The server is executing the first part of a multiple-table update. It is updating only the first table, and saving columns and offsets to be used for updating the other (reference) tables.

- `updating reference tables`

  The server is executing the second part of a multiple-table update and updating the matched rows from the other tables.

- `User lock`

  The thread is going to request or is waiting for an advisory lock requested with a `GET_LOCK()` call. For `SHOW PROFILE`, this state means the thread is requesting the lock (not waiting for it).

- `User sleep`

  The thread has invoked a `SLEEP()` call.

- `Waiting for commit lock`

  `FLUSH TABLES WITH READ LOCK` is waiting for a commit lock.

- `waiting for handler commit`

The thread is waiting for a transaction to commit versus other parts of query processing.

- `Waiting for tables`

  The thread got a notification that the underlying structure for a table has changed and it needs to reopen the table to get the new structure. However, to reopen the table, it must wait until all other threads have closed the table in question.

  This notification takes place if another thread has used `FLUSH TABLES` or one of the following statements on the table in question: `FLUSH TABLES` *`tbl_name`*, `ALTER TABLE`, `RENAME TABLE`, `REPAIR TABLE`, `ANALYZE TABLE`, or `OPTIMIZE TABLE`.

- `Waiting for table flush`

  The thread is executing `FLUSH TABLES` and is waiting for all threads to close their tables, or the thread got a notification that the underlying structure for a table has changed and it needs to reopen the table to get the new structure. However, to reopen the table, it must wait until all other threads have closed the table in question.

  This notification takes place if another thread has used `FLUSH TABLES` or one of the following statements on the table in question: `FLUSH TABLES` *`tbl_name`*, `ALTER TABLE`, `RENAME TABLE`, `REPAIR TABLE`, `ANALYZE TABLE`, or `OPTIMIZE TABLE`.

- `Waiting for` *`lock_type`* `lock`

  The server is waiting to acquire a `THR_LOCK` lock or a lock from the metadata locking subsystem, where *`lock_type`* indicates the type of lock.

  This state indicates a wait for a `THR_LOCK`:

  - `Waiting for table level lock`

  These states indicate a wait for a metadata lock:

  - `Waiting for event metadata lock`
  - `Waiting for global read lock`
  - `Waiting for schema metadata lock`
  - `Waiting for stored function metadata lock`
  - `Waiting for stored procedure metadata lock`
  - `Waiting for table metadata lock`
  - `Waiting for trigger metadata lock`

  For information about table lock indicators, see Section 8.11.1, “Internal Locking Methods”. For information about metadata locking, see Section 8.11.4, “Metadata Locking”. To see which locks are blocking lock requests, use the Performance Schema lock tables described at Section 27.12.13, “Performance Schema Lock Tables”.

- `Waiting on cond`

  A generic state in which the thread is waiting for a condition to become true. No specific state information is available.

- `Writing to net`

  The server is writing a packet to the network.

8.14.4 Replication Source Thread States

The following list shows the most common states you may see in the `State` column for the `Binlog Dump` thread of the replication source. If you see no `Binlog Dump` threads on a source, this means that replication is not running; that is, that no replicas are currently connected.

In MySQL 8.0.26, incompatible changes were made to instrumentation names, including the names of thread stages, containing the terms “master”, which is changed to “source”, “slave”, which is changed to “replica”, and “mts” (for “multithreaded slave”), which is changed to “mta” (for “multithreaded applier”). Monitoring tools that work with these instrumentation names might be impacted. If the incompatible changes have an impact for you, set the `terminology_use_previous` system variable to `BEFORE_8_0_26` to make MySQL Server use the old versions of the names for the objects specified in the previous list. This enables monitoring tools that rely on the old names to continue working until they can be updated to use the new names.

Set the `terminology_use_previous` system variable with session scope to support individual functions, or global scope to be a default for all new sessions. When global scope is used, the slow query log contains the old versions of the names.

- `Finished reading one binlog; switching to next binlog`

  The thread has finished reading a binary log file and is opening the next one to send to the replica.

- `Master has sent all binlog to slave; waiting for more updates`

  From MySQL 8.0.26: `Source has sent all binlog to replica; waiting for more updates`

  The thread has read all remaining updates from the binary logs and sent them to the replica. The thread is now idle, waiting for new events to appear in the binary log resulting from new updates occurring on the source.

- `Sending binlog event to slave`

  From MySQL 8.0.26: `Sending binlog event to replica`

  Binary logs consist of *events*, where an event is usually an update plus some other information. The thread has read an event from the binary log and is now sending it to the replica.

- `Waiting to finalize termination`

  A very brief state that occurs as the thread is stopping.

8.14.5 Replication I/O (Receiver) Thread States

The following list shows the most common states you see in the `State` column for a replication I/O (receiver) thread on a replica server. This state also appears in the `Replica_IO_State` column displayed by `SHOW REPLICA STATUS` (or before MySQL 8.0.22, `SHOW REPLICA STATUS`), so you can get a good view of what is happening by using that statement.

In MySQL 8.0.26, incompatible changes were made to instrumentation names, including the names of thread stages, containing the terms “master”, which is changed to “source”, “slave”, which is changed to “replica”, and “mts” (for “multithreaded slave”), which is changed to “mta” (for “multithreaded applier”). Monitoring tools that work with these instrumentation names might be impacted. If the incompatible changes have an impact for you, set the `terminology_use_previous` system variable to `BEFORE_8_0_26` to make MySQL Server use the old versions of the names for the objects specified in the previous list. This enables monitoring tools that rely on the old names to continue working until they can be updated to use the new names.

Set the `terminology_use_previous` system variable with session scope to support individual functions, or global scope to be a default for all new sessions. When global scope is used, the slow query log contains the old versions of the names.

- `Checking master version`

  From MySQL 8.0.26: `Checking source version`

  A state that occurs very briefly, after the connection to the source is established.

- `Connecting to master`

  From MySQL 8.0.26: `Connecting to source`

  The thread is attempting to connect to the source.

- `Queueing master event to the relay log`

  From MySQL 8.0.26: `Queueing source event to the relay log`

  The thread has read an event and is copying it to the relay log so that the SQL thread can process it.

- `Reconnecting after a failed binlog dump request`

  The thread is trying to reconnect to the source.

- `Reconnecting after a failed master event read`

  From MySQL 8.0.26: `Reconnecting after a failed source event read`

  The thread is trying to reconnect to the source. When connection is established again, the state becomes `Waiting for master to send event`.

- `Registering slave on master`

  From MySQL 8.0.26: `Registering replica on source`

  A state that occurs very briefly after the connection to the source is established.

- `Requesting binlog dump`

  A state that occurs very briefly, after the connection to the source is established. The thread sends to the source a request for the contents of its binary logs, starting from the requested binary log file name and position.

- `Waiting for its turn to commit`

  A state that occurs when the replica thread is waiting for older worker threads to commit if `replica_preserve_commit_order` or `slave_preserve_commit_order` is enabled.

- `Waiting for master to send event`

  From MySQL 8.0.26: `Waiting for source to send event`

  The thread has connected to the source and is waiting for binary log events to arrive. This can last for a long time if the source is idle. If the wait lasts for `replica_net_timeout` or `slave_net_timeout` seconds, a timeout occurs. At that point, the thread considers the connection to be broken and makes an attempt to reconnect.

- `Waiting for master update`

  From MySQL 8.0.26: `Waiting for source update`

  The initial state before `Connecting to master` or `Connecting to source`.

- `Waiting for slave mutex on exit`

  From MySQL 8.0.26: `Waiting for replica mutex on exit`

A state that occurs briefly as the thread is stopping.

- `Waiting for the slave SQL thread to free enough relay log space`

  From MySQL 8.0.26: `Waiting for the replica SQL thread to free enough relay log space`

  You are using a nonzero `relay_log_space_limit` value, and the relay logs have grown large enough that their combined size exceeds this value. The I/O (receiver) thread is waiting until the SQL (applier) thread frees enough space by processing relay log contents so that it can delete some relay log files.

- `Waiting to reconnect after a failed binlog dump request`

  If the binary log dump request failed (due to disconnection), the thread goes into this state while it sleeps, then tries to reconnect periodically. The interval between retries can be specified using the `CHANGE REPLICATION SOURCE TO` statement (from MySQL 8.0.23) or `CHANGE MASTER TO` statement (before MySQL 8.0.23).

- `Waiting to reconnect after a failed master event read`

  From MySQL 8.0.26: `Waiting to reconnect after a failed source event read`

  An error occurred while reading (due to disconnection). The thread is sleeping for the number of seconds set by the `CHANGE REPLICATION SOURCE TO` statement (from MySQL 8.0.23) or `CHANGE MASTER TO` statement (before MySQL 8.0.23), which defaults to 60, before attempting to reconnect.

8.14.6 Replication SQL Thread States

The following list shows the most common states you may see in the `State` column for a replication SQL thread on a replica server.

In MySQL 8.0.26, incompatible changes were made to instrumentation names, including the names of thread stages, containing the terms “master”, which is changed to “source”, “slave”, which is changed to “replica”, and “mts” (for “multithreaded slave”), which is changed to “mta” (for “multithreaded applier”). Monitoring tools that work with these instrumentation names might be impacted. If the incompatible changes have an impact for you, set the `terminology_use_previous` system variable to `BEFORE_8_0_26` to make MySQL Server use the old versions of the names for the objects specified in the previous list. This enables monitoring tools that rely on the old names to continue working until they can be updated to use the new names.

Set the `terminology_use_previous` system variable with session scope to support individual functions, or global scope to be a default for all new sessions. When global scope is used, the slow query log contains the old versions of the names.

- `Making temporary file (append) before replaying LOAD DATA INFILE`

  The thread is executing a `LOAD DATA` statement and is appending the data to a temporary file containing the data from which the replica reads rows.

- `Making temporary file (create) before replaying LOAD DATA INFILE`

  The thread is executing a `LOAD DATA` statement and is creating a temporary file containing the data from which the replica reads rows. This state can only be encountered if the original `LOAD DATA` statement was logged by a source running a version of MySQL lower than MySQL 5.0.3.

- `Reading event from the relay log`

  The thread has read an event from the relay log so that the event can be processed.

- `Slave has read all relay log; waiting for more updates`

  From MySQL 8.0.26: `Replica has read all relay log; waiting for more updates`

  The thread has processed all events in the relay log files, and is now waiting for the I/O (receiver) thread to write new events to the relay log.

- `Waiting for an event from Coordinator`

  Using the multithreaded replica (`replica_parallel_workers` or `slave_parallel_workers` is greater than 1), one of the replica worker threads is waiting for an event from the coordinator thread.

- `Waiting for slave mutex on exit`

  From MySQL 8.0.26: `Waiting for replica mutex on exit`

  A very brief state that occurs as the thread is stopping.

- `Waiting for Slave Workers to free pending events`

  From MySQL 8.0.26: `Waiting for Replica Workers to free pending events`

  This waiting action occurs when the total size of events being processed by Workers exceeds the size of the `replica_pending_jobs_size_max` or `slave_pending_jobs_size_max` system variable. The Coordinator resumes scheduling when the size drops below this limit. This state occurs only when `replica_parallel_workers` or `slave_parallel_workers` is set greater than 0.

- `Waiting for the next event in relay log`

  The initial state before `Reading event from the relay log`.

- `Waiting until MASTER_DELAY seconds after master executed event`

  From MySQL 8.0.26: `Waiting until SOURCE_DELAY seconds after master executed event`

  The SQL thread has read an event but is waiting for the replica delay to lapse. This delay is set with the `SOURCE_DELAY` | `MASTER_DELAY` option of the `CHANGE REPLICATION SOURCE TO` statement (from MySQL 8.0.23) or `CHANGE MASTER TO` statement (before MySQL 8.0.23).

The `Info` column for the SQL thread may also show the text of a statement. This indicates that the thread has read an event from the relay log, extracted the statement from it, and may be executing it.

8.14.7 Replication Connection Thread States

These thread states occur on a replica server but are associated with connection threads, not with the I/O or SQL threads.

In MySQL 8.0.26, incompatible changes were made to instrumentation names, including the names of thread stages, containing the terms “master”, which is changed to “source”, “slave”, which is changed to “replica”, and “mts” (for “multithreaded slave”), which is changed to “mta” (for “multithreaded applier”). Monitoring tools that work with these instrumentation names might be impacted. If the incompatible changes have an impact for you, set the `terminology_use_previous` system variable to `BEFORE_8_0_26` to make MySQL Server use the old versions of the names for the objects specified in the previous list. This enables monitoring tools that rely on the old names to continue working until they can be updated to use the new names.

Set the `terminology_use_previous` system variable with session scope to support individual functions, or global scope to be a default for all new sessions. When global scope is used, the slow query log contains the old versions of the names.

- `Changing master`

From MySQL 8.0.26: `Changing replication source`

The thread is processing a `CHANGE REPLICATION SOURCE TO` statement (from MySQL 8.0.23) or `CHANGE MASTER TO` statement (before MySQL 8.0.23).

- `Killing slave`

  The thread is processing a `STOP REPLICA` statement.

- `Opening master dump table`

  This state occurs after `Creating table from master dump`.

- `Reading master dump table data`

  This state occurs after `Opening master dump table`.

- `Rebuilding the index on master dump table`

  This state occurs after `Reading master dump table data`.

8.14.8 NDB Cluster Thread States

- `Committing events to binlog`
- `Opening mysql.ndb_apply_status`
- `Processing events`

  The thread is processing events for binary logging.

- `Processing events from schema table`

  The thread is doing the work of schema replication.

- `Shutting down`
- `Syncing ndb table schema operation and binlog`

  This is used to have a correct binary log of schema operations for NDB.

- `Waiting for allowed to take ndbcluster global schema lock`

  The thread is waiting for permission to take a global schema lock.

- `Waiting for event from ndbcluster`

  The server is acting as an SQL node in an NDB Cluster, and is connected to a cluster management node.

- `Waiting for first event from ndbcluster`
- `Waiting for ndbcluster binlog update to reach current position`
- `Waiting for ndbcluster global schema lock`

  The thread is waiting for a global schema lock held by another thread to be released.

- `Waiting for ndbcluster to start`
- `Waiting for schema epoch`

  The thread is waiting for a schema epoch (that is, a global checkpoint).

8.14.9 Event Scheduler Thread States

These states occur for the Event Scheduler thread, threads that are created to execute scheduled events, or threads that terminate the scheduler.

- `Clearing`

  The scheduler thread or a thread that was executing an event is terminating and is about to end.

- `Initialized`

  The scheduler thread or a thread that executes an event has been initialized.

- `Waiting for next activation`

  The scheduler has a nonempty event queue but the next activation is in the future.

- `Waiting for scheduler to stop`

  The thread issued `SET GLOBAL event_scheduler=OFF` and is waiting for the scheduler to stop.

- `Waiting on empty queue`

  The scheduler's event queue is empty and it is sleeping.

Chapter 9 Language Structure

Table of Contents

This chapter discusses the rules for writing the following elements of SQL statements when using MySQL:

- Literal values such as strings and numbers
- Identifiers such as database, table, and column names
- Keywords and reserved words
- User-defined and system variables
- Expressions
- Query attributes
- Comments

9.1 Literal Values

This section describes how to write literal values in MySQL. These include strings, numbers, hexadecimal and bit values, boolean values, and `NULL`. The section also covers various nuances that you may encounter when dealing with these basic types in MySQL.

9.1.1 String Literals

A string is a sequence of bytes or characters, enclosed within either single quote (`'`) or double quote (`"`) characters. Examples:

```
'a string'
"another string"
```

Quoted strings placed next to each other are concatenated to a single string. The following lines are equivalent:

```
'a string'
'a' ' ' 'string'
```

If the ANSI_QUOTES SQL mode is enabled, string literals can be quoted only within single quotation marks because a string quoted within double quotation marks is interpreted as an identifier.

A *binary string* is a string of bytes. Every binary string has a character set and collation named binary. A *nonbinary string* is a string of characters. It has a character set other than binary and a collation that is compatible with the character set.

For both types of strings, comparisons are based on the numeric values of the string unit. For binary strings, the unit is the byte; comparisons use numeric byte values. For nonbinary strings, the unit is the character and some character sets support multibyte characters; comparisons use numeric character code values. Character code ordering is a function of the string collation. (For more information, see Section 10.8.5, “The binary Collation Compared to _bin Collations”.)

Note

Within the mysql client, binary strings display using hexadecimal notation, depending on the value of the --binary-as-hex. For more information about that option, see Section 4.5.1, “mysql — The MySQL Command-Line Client”.

A character string literal may have an optional character set introducer and COLLATE clause, to designate it as a string that uses a particular character set and collation:

```
[_charset_name]'string' [COLLATE collation_name]
```

Examples:

```
SELECT _latin1'string';
SELECT _binary'string';
SELECT _utf8mb4'string' COLLATE utf8mb4_danish_ci;
```

You can use N'literal' (or n'literal') to create a string in the national character set. These statements are equivalent:

```
SELECT N'some text';
SELECT n'some text';
SELECT _utf8'some text';
```

For information about these forms of string syntax, see Section 10.3.7, “The National Character Set”, and Section 10.3.8, “Character Set Introducers”.

Within a string, certain sequences have special meaning unless the NO_BACKSLASH_ESCAPES SQL mode is enabled. Each of these sequences begins with a backslash (\), known as the *escape character*. MySQL recognizes the escape sequences shown in Table 9.1, “Special Character Escape Sequences”. For all other escape sequences, backslash is ignored. That is, the escaped character is interpreted as if it was not escaped. For example, \x is just x. These sequences are case-sensitive. For example, \b is interpreted as a backspace, but \B is interpreted as B. Escape processing is done according to the character set indicated by the character_set_connection system variable. This is true even for strings that are preceded by an introducer that indicates a different character set, as discussed in Section 10.3.6, “Character String Literal Character Set and Collation”.

Table 9.1 Special Character Escape Sequences

Escape Sequence	Character Represented by Sequence
\0	An ASCII NUL (X'00') character
\'	A single quote (') character
\"	A double quote (") character
\b	A backspace character
\n	A newline (linefeed) character
\r	A carriage return character
\t	A tab character
\Z	ASCII 26 (Control+Z); see note following the table

Escape Sequence	Character Represented by Sequence
`\\`	A backslash (`\`) character
`\%`	A `%` character; see note following the table
`\_`	A `_` character; see note following the table

The ASCII 26 character can be encoded as `\Z` to enable you to work around the problem that ASCII 26 stands for END-OF-FILE on Windows. ASCII 26 within a file causes problems if you try to use `mysql db_name < file_name`.

The `\%` and `\_` sequences are used to search for literal instances of `%` and `_` in pattern-matching contexts where they would otherwise be interpreted as wildcard characters. See the description of the `LIKE` operator in Section 12.8.1, “String Comparison Functions and Operators”. If you use `\%` or `\_` outside of pattern-matching contexts, they evaluate to the strings `\%` and `\_`, not to `%` and `_`.

There are several ways to include quote characters within a string:

- A `'` inside a string quoted with `'` may be written as `''`.
- A `"` inside a string quoted with `"` may be written as `""`.
- Precede the quote character by an escape character (`\`).
- A `'` inside a string quoted with `"` needs no special treatment and need not be doubled or escaped. In the same way, `"` inside a string quoted with `'` needs no special treatment.

The following `SELECT` statements demonstrate how quoting and escaping work:

```
mysql> SELECT 'hello', '"hello"', '""hello""', 'hel''lo', '\'hello';
+-------+---------+-----------+--------+--------+
| hello | "hello" | ""hello"" | hel'lo | 'hello |
+-------+---------+-----------+--------+--------+

mysql> SELECT "hello", "'hello'", "''hello''", "hel""lo", "\"hello";
+-------+---------+-----------+--------+--------+
| hello | 'hello' | ''hello'' | hel"lo | "hello |
+-------+---------+-----------+--------+--------+

mysql> SELECT 'This\nIs\nFour\nLines';
+--------------------+
| This
Is
Four
Lines |
+--------------------+

mysql> SELECT 'disappearing\ backslash';
+------------------------+
| disappearing backslash |
+------------------------+
```

To insert binary data into a string column (such as a `BLOB` column), you should represent certain characters by escape sequences. Backslash (`\`) and the quote character used to quote the string must be escaped. In certain client environments, it may also be necessary to escape `NUL` or Control+Z. The `mysql` client truncates quoted strings containing `NUL` characters if they are not escaped, and Control+Z may be taken for END-OF-FILE on Windows if not escaped. For the escape sequences that represent each of these characters, see Table 9.1, “Special Character Escape Sequences”.

When writing application programs, any string that might contain any of these special characters must be properly escaped before the string is used as a data value in an SQL statement that is sent to the MySQL server. You can do this in two ways:

- Process the string with a function that escapes the special characters. In a C program, you can use the `mysql_real_escape_string_quote()` C API function to escape characters. See mysql_real_escape_string_quote(). Within SQL statements that construct other SQL statements, you

can use the `QUOTE()` function. The Perl DBI interface provides a `quote` method to convert special characters to the proper escape sequences. See Section 29.9, “MySQL Perl API”. Other language interfaces may provide a similar capability.

- As an alternative to explicitly escaping special characters, many MySQL APIs provide a placeholder capability that enables you to insert special markers into a statement string, and then bind data values to them when you issue the statement. In this case, the API takes care of escaping special characters in the values for you.

9.1.2 Numeric Literals

Number literals include exact-value (integer and `DECIMAL`) literals and approximate-value (floating-point) literals.

Integers are represented as a sequence of digits. Numbers may include `.` as a decimal separator. Numbers may be preceded by `-` or `+` to indicate a negative or positive value, respectively. Numbers represented in scientific notation with a mantissa and exponent are approximate-value numbers.

Exact-value numeric literals have an integer part or fractional part, or both. They may be signed. Examples: `1`, `.2`, `3.4`, `-5`, `-6.78`, `+9.10`.

Approximate-value numeric literals are represented in scientific notation with a mantissa and exponent. Either or both parts may be signed. Examples: `1.2E3`, `1.2E-3`, `-1.2E3`, `-1.2E-3`.

Two numbers that look similar may be treated differently. For example, `2.34` is an exact-value (fixed-point) number, whereas `2.34E0` is an approximate-value (floating-point) number.

The `DECIMAL` data type is a fixed-point type and calculations are exact. In MySQL, the `DECIMAL` type has several synonyms: `NUMERIC`, `DEC`, `FIXED`. The integer types also are exact-value types. For more information about exact-value calculations, see Section 12.25, “Precision Math”.

The `FLOAT` and `DOUBLE` data types are floating-point types and calculations are approximate. In MySQL, types that are synonymous with `FLOAT` or `DOUBLE` are `DOUBLE PRECISION` and `REAL`.

An integer may be used in floating-point context; it is interpreted as the equivalent floating-point number.

9.1.3 Date and Time Literals

- Standard SQL and ODBC Date and Time Literals
- String and Numeric Literals in Date and Time Context

Date and time values can be represented in several formats, such as quoted strings or as numbers, depending on the exact type of the value and other factors. For example, in contexts where MySQL expects a date, it interprets any of `'2015-07-21'`, `'20150721'`, and `20150721` as a date.

This section describes the acceptable formats for date and time literals. For more information about the temporal data types, such as the range of permitted values, see Section 11.2, “Date and Time Data Types”.

Standard SQL and ODBC Date and Time Literals

Standard SQL requires temporal literals to be specified using a type keyword and a string. The space between the keyword and string is optional.

```
DATE 'str'
TIME 'str'
TIMESTAMP 'str'
```

MySQL recognizes but, unlike standard SQL, does not require the type keyword. Applications that are to be standard-compliant should include the type keyword for temporal literals.

MySQL also recognizes the ODBC syntax corresponding to the standard SQL syntax:

```
{ d 'str' }
{ t 'str' }
{ ts 'str' }
```

MySQL uses the type keywords and the ODBC constructions to produce `DATE`, `TIME`, and `DATETIME` values, respectively, including a trailing fractional seconds part if specified. The `TIMESTAMP` syntax produces a `DATETIME` value in MySQL because `DATETIME` has a range that more closely corresponds to the standard SQL `TIMESTAMP` type, which has a year range from `0001` to `9999`. (The MySQL `TIMESTAMP` year range is `1970` to `2038`.)

String and Numeric Literals in Date and Time Context

MySQL recognizes `DATE` values in these formats:

- As a string in either *`'YYYY-MM-DD'`* or *`'YY-MM-DD'`* format. A “relaxed” syntax is permitted, but is deprecated: Any punctuation character may be used as the delimiter between date parts. For example, `'2012-12-31'`, `'2012/12/31'`, `'2012^12^31'`, and `'2012@12@31'` are equivalent. Beginning with MySQL 8.0.29, using any character other than the dash (`-`) as the delimiter raises a warning, as shown here:

```
mysql> SELECT DATE'2012@12@31';
+------------------+
| DATE'2012@12@31' |
+------------------+
| 2012-12-31       |
+------------------+
1 row in set, 1 warning (0.00 sec)

mysql> SHOW WARNINGS\G
*************************** 1. row ***************************
  Level: Warning
   Code: 4095
Message: Delimiter '@' in position 4 in datetime value '2012@12@31' at row 1 is
deprecated. Prefer the standard '-'.
1 row in set (0.00 sec)
```

- As a string with no delimiters in either *`'YYYYMMDD'`* or *`'YYMMDD'`* format, provided that the string makes sense as a date. For example, `'20070523'` and `'070523'` are interpreted as `'2007-05-23'`, but `'071332'` is illegal (it has nonsensical month and day parts) and becomes `'0000-00-00'`.

- As a number in either *`YYYYMMDD`* or *`YYMMDD`* format, provided that the number makes sense as a date. For example, `19830905` and `830905` are interpreted as `'1983-09-05'`.

MySQL recognizes `DATETIME` and `TIMESTAMP` values in these formats:

- As a string in either *`'YYYY-MM-DD hh:mm:ss'`* or *`'YY-MM-DD hh:mm:ss'`* format. MySQL also permits a “relaxed” syntax here, although this is deprecated: Any punctuation character may be used as the delimiter between date parts or time parts. For example, `'2012-12-31 11:30:45'`, `'2012^12^31 11+30+45'`, `'2012/12/31 11*30*45'`, and `'2012@12@31 11^30^45'` are equivalent. Beginning with MySQL 8.0.29, use of any characters as delimiters in such values, other than the dash (`-`) for the date part and the colon (`:`) for the time part, raises a warning, as shown here:

```
mysql> SELECT TIMESTAMP'2012^12^31 11*30*45';
+---------------------------------+
| TIMESTAMP'2012^12^31 11*30*45' |
+---------------------------------+
| 2012-12-31 11:30:45             |
+---------------------------------+
1 row in set, 1 warning (0.00 sec)

mysql> SHOW WARNINGS\G
*************************** 1. row ***************************
  Level: Warning
```

```
   Code: 4095
Message: Delimiter '^' in position 4 in datetime value '2012^12^31 11*30*45' at
row 1 is deprecated. Prefer the standard '-'.
1 row in set (0.00 sec)
```

The only delimiter recognized between a date and time part and a fractional seconds part is the decimal point.

The date and time parts can be separated by `T` rather than a space. For example, `'2012-12-31 11:30:45'` `'2012-12-31T11:30:45'` are equivalent.

Previously, MySQL supported arbitrary numbers of leading and trailing whitespace characters in date and time values, as well as between the date and time parts of `DATETIME` and `TIMESTAMP` values. In MySQL 8.0.29 and later, this behavior is deprecated, and the presence of excess whitespace characters triggers a warning, as shown here:

```
mysql> SELECT TIMESTAMP'2012-12-31   11-30-45';
+---------------------------------+
| TIMESTAMP'2012-12-31   11-30-45' |
+---------------------------------+
| 2012-12-31 11:30:45             |
+---------------------------------+
1 row in set, 1 warning (0.00 sec)

mysql> SHOW WARNINGS\G
*************************** 1. row ***************************
  Level: Warning
   Code: 4096
Message: Delimiter ' ' in position 11 in datetime value '2012-12-31   11-30-45'
at row 1 is superfluous and is deprecated. Please remove.
1 row in set (0.00 sec)
```

Also beginning with MySQL 8.0.29, a warning is raised when whitespace characters other than the space character is used, like this:

```
mysql> SELECT TIMESTAMP'2021-06-06
    '> 11:15:25';
+-------------------------------+
| TIMESTAMP'2021-06-06
 11:15:25'                      |
+-------------------------------+
| 2021-06-06 11:15:25           |
+-------------------------------+
1 row in set, 1 warning (0.00 sec)

mysql> SHOW WARNINGS\G
*************************** 1. row ***************************
  Level: Warning
   Code: 4095
Message: Delimiter '\n' in position 10 in datetime value '2021-06-06
11:15:25' at row 1 is deprecated. Prefer the standard ' '.
1 row in set (0.00 sec)
```

Only one such warning is raised per temporal value, even though multiple issues may exist with delimiters, whitespace, or both, as shown in the following series of statements:

```
mysql> SELECT TIMESTAMP'2012!-12-31  11:30:45';
+---------------------------------+
| TIMESTAMP'2012!-12-31  11:30:45' |
+---------------------------------+
| 2012-12-31 11:30:45             |
+---------------------------------+
1 row in set, 1 warning (0.00 sec)

mysql> SHOW WARNINGS\G
*************************** 1. row ***************************
  Level: Warning
   Code: 4095
Message: Delimiter '!' in position 4 in datetime value '2012!-12-31  11:30:45'
```

```
at row 1 is deprecated. Prefer the standard '-'.
1 row in set (0.00 sec)

mysql> SELECT TIMESTAMP'2012-12-31  11:30:45';
+---------------------------------+
| TIMESTAMP'2012-12-31  11:30:45' |
+---------------------------------+
| 2012-12-31 11:30:45             |
+---------------------------------+
1 row in set, 1 warning (0.00 sec)

mysql> SHOW WARNINGS\G
*************************** 1. row ***************************
  Level: Warning
   Code: 4096
Message: Delimiter ' ' in position 11 in datetime value '2012-12-31  11:30:45'
at row 1 is superfluous and is deprecated. Please remove.
1 row in set (0.00 sec)

mysql> SELECT TIMESTAMP'2012-12-31 11:30:45';
+-------------------------------+
| TIMESTAMP'2012-12-31 11:30:45' |
+-------------------------------+
| 2012-12-31 11:30:45           |
+-------------------------------+
1 row in set (0.00 sec)
```

- As a string with no delimiters in either `'YYYYMMDDhhmmss'` or `'YYMMDDhhmmss'` format, provided that the string makes sense as a date. For example, `'20070523091528'` and `'070523091528'` are interpreted as `'2007-05-23 09:15:28'`, but `'071122129015'` is illegal (it has a nonsensical minute part) and becomes `'0000-00-00 00:00:00'`.

- As a number in either *YYYYMMDDhhmmss* or *YYMMDDhhmmss* format, provided that the number makes sense as a date. For example, `19830905132800` and `830905132800` are interpreted as `'1983-09-05 13:28:00'`.

A `DATETIME` or `TIMESTAMP` value can include a trailing fractional seconds part in up to microseconds (6 digits) precision. The fractional part should always be separated from the rest of the time by a decimal point; no other fractional seconds delimiter is recognized. For information about fractional seconds support in MySQL, see Section 11.2.6, “Fractional Seconds in Time Values”.

Dates containing two-digit year values are ambiguous because the century is unknown. MySQL interprets two-digit year values using these rules:

- Year values in the range `70-99` become `1970-1999`.

- Year values in the range `00-69` become `2000-2069`.

See also Section 11.2.8, “2-Digit Years in Dates”.

For values specified as strings that include date part delimiters, it is unnecessary to specify two digits for month or day values that are less than `10`. `'2015-6-9'` is the same as `'2015-06-09'`. Similarly, for values specified as strings that include time part delimiters, it is unnecessary to specify two digits for hour, minute, or second values that are less than `10`. `'2015-10-30 1:2:3'` is the same as `'2015-10-30 01:02:03'`.

Values specified as numbers should be 6, 8, 12, or 14 digits long. If a number is 8 or 14 digits long, it is assumed to be in *YYYYMMDD* or *YYYYMMDDhhmmss* format and that the year is given by the first 4 digits. If the number is 6 or 12 digits long, it is assumed to be in *YYMMDD* or *YYMMDDhhmmss* format and that the year is given by the first 2 digits. Numbers that are not one of these lengths are interpreted as though padded with leading zeros to the closest length.

Values specified as nondelimited strings are interpreted according their length. For a string 8 or 14 characters long, the year is assumed to be given by the first 4 characters. Otherwise, the year is assumed to be given by the first 2 characters. The string is interpreted from left to right to find year,

month, day, hour, minute, and second values, for as many parts as are present in the string. This means you should not use strings that have fewer than 6 characters. For example, if you specify `'9903'`, thinking that represents March, 1999, MySQL converts it to the “zero” date value. This occurs because the year and month values are `99` and `03`, but the day part is completely missing. However, you can explicitly specify a value of zero to represent missing month or day parts. For example, to insert the value `'1999-03-00'`, use `'990300'`.

MySQL recognizes `TIME` values in these formats:

- As a string in *`'D hh:mm:ss'`* format. You can also use one of the following “relaxed” syntaxes: *`'hh:mm:ss'`*, *`'hh:mm'`*, *`'D hh:mm'`*, *`'D hh'`*, or *`'ss'`*. Here *`D`* represents days and can have a value from 0 to 34.
- As a string with no delimiters in *`'hhmmss'`* format, provided that it makes sense as a time. For example, `'101112'` is understood as `'10:11:12'`, but `'109712'` is illegal (it has a nonsensical minute part) and becomes `'00:00:00'`.
- As a number in *`hhmmss`* format, provided that it makes sense as a time. For example, `101112` is understood as `'10:11:12'`. The following alternative formats are also understood: *`ss`*, *`mmss`*, or *`hhmmss`*.

A trailing fractional seconds part is recognized in the *`'D hh:mm:ss.fraction'`*, *`'hh:mm:ss.fraction'`*, *`'hhmmss.fraction'`*, and *`hhmmss.fraction`* time formats, where `fraction` is the fractional part in up to microseconds (6 digits) precision. The fractional part should always be separated from the rest of the time by a decimal point; no other fractional seconds delimiter is recognized. For information about fractional seconds support in MySQL, see Section 11.2.6, “Fractional Seconds in Time Values”.

For `TIME` values specified as strings that include a time part delimiter, it is unnecessary to specify two digits for hours, minutes, or seconds values that are less than `10`. `'8:3:2'` is the same as `'08:03:02'`.

Beginning with MySQL 8.0.19, you can specify a time zone offset when inserting `TIMESTAMP` and `DATETIME` values into a table. The offset is appended to the time part of a datetime literal, with no intravening spaces, and uses the same format used for setting the `time_zone` system variable, with the following exceptions:

- For hour values less than 10, a leading zero is required.
- The value `'-00:00'` is rejected.
- Time zone names such as `'EET'` and `'Asia/Shanghai'` cannot be used; `'SYSTEM'` also cannot be used in this context.

The value inserted must not have a zero for the month part, the day part, or both parts. This is enforced beginning with MySQL 8.0.22, regardless of the server SQL mode setting.

This example illustrates inserting datetime values with time zone offsets into `TIMESTAMP` and `DATETIME` columns using different `time_zone` settings, and then retrieving them:

```
mysql> CREATE TABLE ts (
    ->     id INTEGER NOT NULL AUTO_INCREMENT PRIMARY KEY,
    ->     col TIMESTAMP NOT NULL
    -> ) AUTO_INCREMENT = 1;

mysql> CREATE TABLE dt (
    ->     id INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
    ->     col DATETIME NOT NULL
    -> ) AUTO_INCREMENT = 1;

mysql> SET @@time_zone = 'SYSTEM';

mysql> INSERT INTO ts (col) VALUES ('2020-01-01 10:10:10'),
```

```
    ->     ('2020-01-01 10:10:10+05:30'), ('2020-01-01 10:10:10-08:00');

mysql> SET @@time_zone = '+00:00';

mysql> INSERT INTO ts (col) VALUES ('2020-01-01 10:10:10'),
    ->     ('2020-01-01 10:10:10+05:30'), ('2020-01-01 10:10:10-08:00');

mysql> SET @@time_zone = 'SYSTEM';

mysql> INSERT INTO dt (col) VALUES ('2020-01-01 10:10:10'),
    ->     ('2020-01-01 10:10:10+05:30'), ('2020-01-01 10:10:10-08:00');

mysql> SET @@time_zone = '+00:00';

mysql> INSERT INTO dt (col) VALUES ('2020-01-01 10:10:10'),
    ->     ('2020-01-01 10:10:10+05:30'), ('2020-01-01 10:10:10-08:00');

mysql> SET @@time_zone = 'SYSTEM';

mysql> SELECT @@system_time_zone;
+--------------------+
| @@system_time_zone |
+--------------------+
| EST                |
+--------------------+

mysql> SELECT col, UNIX_TIMESTAMP(col) FROM dt ORDER BY id;
+---------------------+---------------------+
| col                 | UNIX_TIMESTAMP(col) |
+---------------------+---------------------+
| 2020-01-01 10:10:10 |          1577891410 |
| 2019-12-31 23:40:10 |          1577853610 |
| 2020-01-01 13:10:10 |          1577902210 |
| 2020-01-01 10:10:10 |          1577891410 |
| 2020-01-01 04:40:10 |          1577871610 |
| 2020-01-01 18:10:10 |          1577920210 |
+---------------------+---------------------+

mysql> SELECT col, UNIX_TIMESTAMP(col) FROM ts ORDER BY id;
+---------------------+---------------------+
| col                 | UNIX_TIMESTAMP(col) |
+---------------------+---------------------+
| 2020-01-01 10:10:10 |          1577891410 |
| 2019-12-31 23:40:10 |          1577853610 |
| 2020-01-01 13:10:10 |          1577902210 |
| 2020-01-01 05:10:10 |          1577873410 |
| 2019-12-31 23:40:10 |          1577853610 |
| 2020-01-01 13:10:10 |          1577902210 |
+---------------------+---------------------+
```

The offset is not displayed when selecting a datetime value, even if one was used when inserting it.

The range of supported offset values is `-13:59` to `+14:00`, inclusive.

Datetime literals that include time zone offsets are accepted as parameter values by prepared statements.

9.1.4 Hexadecimal Literals

Hexadecimal literal values are written using `X'val'` or `0xval` notation, where *`val`* contains hexadecimal digits (`0..9`, `A..F`). Lettercase of the digits and of any leading `X` does not matter. A leading `0x` is case-sensitive and cannot be written as `0X`.

Legal hexadecimal literals:

```
X'01AF'
X'01af'
x'01AF'
x'01af'
0x01AF
```

```
0x01af
```

Illegal hexadecimal literals:

```
X'0G'   (G is not a hexadecimal digit)
0X01AF  (0X must be written as 0x)
```

Values written using `X'val'` notation must contain an even number of digits or a syntax error occurs. To correct the problem, pad the value with a leading zero:

```
mysql> SET @s = X'FFF';
ERROR 1064 (42000): You have an error in your SQL syntax;
check the manual that corresponds to your MySQL server
version for the right syntax to use near 'X'FFF''

mysql> SET @s = X'0FFF';
Query OK, 0 rows affected (0.00 sec)
```

Values written using `0xval` notation that contain an odd number of digits are treated as having an extra leading `0`. For example, `0xaaa` is interpreted as `0x0aaa`.

By default, a hexadecimal literal is a binary string, where each pair of hexadecimal digits represents a character:

```
mysql> SELECT X'4D7953514C', CHARSET(X'4D7953514C');
+---------------+------------------------+
| X'4D7953514C' | CHARSET(X'4D7953514C') |
+---------------+------------------------+
| MySQL         | binary                 |
+---------------+------------------------+
mysql> SELECT 0x5461626c65, CHARSET(0x5461626c65);
+--------------+-----------------------+
| 0x5461626c65 | CHARSET(0x5461626c65) |
+--------------+-----------------------+
| Table        | binary                |
+--------------+-----------------------+
```

A hexadecimal literal may have an optional character set introducer and `COLLATE` clause, to designate it as a string that uses a particular character set and collation:

```
[_charset_name] X'val' [COLLATE collation_name]
```

Examples:

```
SELECT _latin1 X'4D7953514C';
SELECT _utf8mb4 0x4D7953514C COLLATE utf8mb4_danish_ci;
```

The examples use `X'val'` notation, but `0xval` notation permits introducers as well. For information about introducers, see Section 10.3.8, “Character Set Introducers”.

In numeric contexts, MySQL treats a hexadecimal literal like a `BIGINT UNSIGNED` (64-bit unsigned integer). To ensure numeric treatment of a hexadecimal literal, use it in numeric context. Ways to do this include adding 0 or using `CAST(... AS UNSIGNED)`. For example, a hexadecimal literal assigned to a user-defined variable is a binary string by default. To assign the value as a number, use it in numeric context:

```
mysql> SET @v1 = X'41';
mysql> SET @v2 = X'41'+0;
mysql> SET @v3 = CAST(X'41' AS UNSIGNED);
mysql> SELECT @v1, @v2, @v3;
+------+------+------+
| @v1  | @v2  | @v3  |
+------+------+------+
| A    |   65 |   65 |
+------+------+------+
```

An empty hexadecimal value (`X''`) evaluates to a zero-length binary string. Converted to a number, it produces 0:

```
mysql> SELECT CHARSET(X''), LENGTH(X'');
+--------------+-------------+
| CHARSET(X'') | LENGTH(X'') |
+--------------+-------------+
| binary       |           0 |
+--------------+-------------+
mysql> SELECT X''+0;
+-------+
| X''+0 |
+-------+
|     0 |
+-------+
```

The `X'val'` notation is based on standard SQL. The `0x` notation is based on ODBC, for which hexadecimal strings are often used to supply values for `BLOB` columns.

To convert a string or a number to a string in hexadecimal format, use the `HEX()` function:

```
mysql> SELECT HEX('cat');
+------------+
| HEX('cat') |
+------------+
| 636174     |
+------------+
mysql> SELECT X'636174';
+-----------+
| X'636174' |
+-----------+
| cat       |
+-----------+
```

For hexadecimal literals, bit operations are considered numeric context, but bit operations permit numeric or binary string arguments in MySQL 8.0 and higher. To explicitly specify binary string context for hexadecimal literals, use a `_binary` introducer for at least one of the arguments:

```
mysql> SET @v1 = X'000D' | X'0BC0';
mysql> SET @v2 = _binary X'000D' | X'0BC0';
mysql> SELECT HEX(@v1), HEX(@v2);
+----------+----------+
| HEX(@v1) | HEX(@v2) |
+----------+----------+
| BCD      | 0BCD     |
+----------+----------+
```

The displayed result appears similar for both bit operations, but the result without `_binary` is a `BIGINT` value, whereas the result with `_binary` is a binary string. Due to the difference in result types, the displayed values differ: High-order 0 digits are not displayed for the numeric result.

9.1.5 Bit-Value Literals

Bit-value literals are written using `b'val'` or `0bval` notation. *`val`* is a binary value written using zeros and ones. Lettercase of any leading `b` does not matter. A leading `0b` is case-sensitive and cannot be written as `0B`.

Legal bit-value literals:

```
b'01'
B'01'
0b01
```

Illegal bit-value literals:

```
b'2'    (2 is not a binary digit)
0B01    (0B must be written as 0b)
```

By default, a bit-value literal is a binary string:

```
mysql> SELECT b'1000001', CHARSET(b'1000001');
+------------+---------------------+
| b'1000001' | CHARSET(b'1000001') |
+------------+---------------------+
| A          | binary              |
+------------+---------------------+
mysql> SELECT 0b1100001, CHARSET(0b1100001);
+-----------+--------------------+
| 0b1100001 | CHARSET(0b1100001) |
+-----------+--------------------+
| a         | binary             |
+-----------+--------------------+
```

A bit-value literal may have an optional character set introducer and `COLLATE` clause, to designate it as a string that uses a particular character set and collation:

```
[_charset_name] b'val' [COLLATE collation_name]
```

Examples:

```
SELECT _latin1 b'1000001';
SELECT _utf8mb4 0b1000001 COLLATE utf8mb4_danish_ci;
```

The examples use `b'val'` notation, but `0bval` notation permits introducers as well. For information about introducers, see Section 10.3.8, “Character Set Introducers”.

In numeric contexts, MySQL treats a bit literal like an integer. To ensure numeric treatment of a bit literal, use it in numeric context. Ways to do this include adding 0 or using `CAST(... AS UNSIGNED)`. For example, a bit literal assigned to a user-defined variable is a binary string by default. To assign the value as a number, use it in numeric context:

```
mysql> SET @v1 = b'1100001';
mysql> SET @v2 = b'1100001'+0;
mysql> SET @v3 = CAST(b'1100001' AS UNSIGNED);
mysql> SELECT @v1, @v2, @v3;
+------+------+------+
| @v1  | @v2  | @v3  |
+------+------+------+
| a    |   97 |   97 |
+------+------+------+
```

An empty bit value (`b''`) evaluates to a zero-length binary string. Converted to a number, it produces 0:

```
mysql> SELECT CHARSET(b''), LENGTH(b'');
+--------------+-------------+
| CHARSET(b'') | LENGTH(b'') |
+--------------+-------------+
| binary       |           0 |
+--------------+-------------+
mysql> SELECT b''+0;
+-------+
| b''+0 |
+-------+
|     0 |
+-------+
```

Bit-value notation is convenient for specifying values to be assigned to `BIT` columns:

```
mysql> CREATE TABLE t (b BIT(8));
mysql> INSERT INTO t SET b = b'11111111';
mysql> INSERT INTO t SET b = b'1010';
mysql> INSERT INTO t SET b = b'0101';
```

Bit values in result sets are returned as binary values, which may not display well. To convert a bit value to printable form, use it in numeric context or use a conversion function such as `BIN()` or `HEX()`. High-order 0 digits are not displayed in the converted value.

```
mysql> SELECT b+0, BIN(b), OCT(b), HEX(b) FROM t;
+------+----------+--------+--------+
| b+0  | BIN(b)   | OCT(b) | HEX(b) |
+------+----------+--------+--------+
|  255 | 11111111 | 377    | FF     |
|   10 | 1010     | 12     | A      |
|    5 | 101      | 5      | 5      |
+------+----------+--------+--------+
```

For bit literals, bit operations are considered numeric context, but bit operations permit numeric or binary string arguments in MySQL 8.0 and higher. To explicitly specify binary string context for bit literals, use a `_binary` introducer for at least one of the arguments:

```
mysql> SET @v1 = b'000010101' | b'000101010';
mysql> SET @v2 = _binary b'000010101' | _binary b'000101010';
mysql> SELECT HEX(@v1), HEX(@v2);
+----------+----------+
| HEX(@v1) | HEX(@v2) |
+----------+----------+
| 3F       | 003F     |
+----------+----------+
```

The displayed result appears similar for both bit operations, but the result without `_binary` is a `BIGINT` value, whereas the result with `_binary` is a binary string. Due to the difference in result types, the displayed values differ: High-order 0 digits are not displayed for the numeric result.

9.1.6 Boolean Literals

The constants `TRUE` and `FALSE` evaluate to `1` and `0`, respectively. The constant names can be written in any lettercase.

```
mysql> SELECT TRUE, true, FALSE, false;
        -> 1, 1, 0, 0
```

9.1.7 NULL Values

The `NULL` value means “no data.” `NULL` can be written in any lettercase.

Be aware that the `NULL` value is different from values such as `0` for numeric types or the empty string for string types. For more information, see Section B.3.4.3, “Problems with NULL Values”.

For text file import or export operations performed with `LOAD DATA` or `SELECT ... INTO OUTFILE`, `NULL` is represented by the `\N` sequence. See Section 13.2.9, “LOAD DATA Statement”.

For sorting with `ORDER BY`, `NULL` values sort before other values for ascending sorts, after other values for descending sorts.

9.2 Schema Object Names

Certain objects within MySQL, including database, table, index, column, alias, view, stored procedure, partition, tablespace, resource group and other object names are known as identifiers. This section describes the permissible syntax for identifiers in MySQL. Section 9.2.1, “Identifier Length Limits”, indicates the maximum length of each type of identifier. Section 9.2.3, “Identifier Case Sensitivity”, describes which types of identifiers are case-sensitive and under what conditions.

An identifier may be quoted or unquoted. If an identifier contains special characters or is a reserved word, you *must* quote it whenever you refer to it. (Exception: A reserved word that follows a period in a qualified name must be an identifier, so it need not be quoted.) Reserved words are listed at Section 9.3, “Keywords and Reserved Words”.

Internally, identifiers are converted to and are stored as Unicode (UTF-8). The permissible Unicode characters in identifiers are those in the Basic Multilingual Plane (BMP). Supplementary characters are not permitted. Identifiers thus may contain these characters:

- Permitted characters in unquoted identifiers:
  - ASCII: [0-9,a-z,A-Z$_] (basic Latin letters, digits 0-9, dollar, underscore)
  - Extended: U+0080 .. U+FFFF
- Permitted characters in quoted identifiers include the full Unicode Basic Multilingual Plane (BMP), except U+0000:
  - ASCII: U+0001 .. U+007F
  - Extended: U+0080 .. U+FFFF
- ASCII NUL (U+0000) and supplementary characters (U+10000 and higher) are not permitted in quoted or unquoted identifiers.
- Identifiers may begin with a digit but unless quoted may not consist solely of digits.
- Database, table, and column names cannot end with space characters.
- Beginning with MySQL 8.0.32, use of the dollar sign as the first character in the unquoted name of a database, table, view, column, stored program, or alias is deprecated and produces a warning. This includes such names used with qualifiers (see Section 9.2.2, “Identifier Qualifiers”). The dollar sign can still be used as the leading character of such an identifier when it is quoted according to the rules given later in this section.

The identifier quote character is the backtick (`):

```
mysql> SELECT * FROM `select` WHERE `select`.id > 100;
```

If the ANSI_QUOTES SQL mode is enabled, it is also permissible to quote identifiers within double quotation marks:

```
mysql> CREATE TABLE "test" (col INT);
ERROR 1064: You have an error in your SQL syntax...
mysql> SET sql_mode='ANSI_QUOTES';
mysql> CREATE TABLE "test" (col INT);
Query OK, 0 rows affected (0.00 sec)
```

The ANSI_QUOTES mode causes the server to interpret double-quoted strings as identifiers. Consequently, when this mode is enabled, string literals must be enclosed within single quotation marks. They cannot be enclosed within double quotation marks. The server SQL mode is controlled as described in Section 5.1.11, “Server SQL Modes”.

Identifier quote characters can be included within an identifier if you quote the identifier. If the character to be included within the identifier is the same as that used to quote the identifier itself, then you need to double the character. The following statement creates a table named a`b that contains a column named c"d:

```
mysql> CREATE TABLE `a``b` (`c"d` INT);
```

In the select list of a query, a quoted column alias can be specified using identifier or string quoting characters:

```
mysql> SELECT 1 AS `one`, 2 AS 'two';
+-----+-----+
| one | two |
+-----+-----+
|   1 |   2 |
+-----+-----+
```

Elsewhere in the statement, quoted references to the alias must use identifier quoting or the reference is treated as a string literal.

It is recommended that you do not use names that begin with `Me` or `MeN`, where `M` and `N` are integers. For example, avoid using `1e` as an identifier, because an expression such as `1e+3` is ambiguous. Depending on context, it might be interpreted as the expression `1e + 3` or as the number `1e+3`.

Be careful when using `MD5()` to produce table names because it can produce names in illegal or ambiguous formats such as those just described.

It is also recommended that you do not use column names that begin with `!hidden!` to ensure that new names do not collide with names used by existing hidden columns for functional indexes.

A user variable cannot be used directly in an SQL statement as an identifier or as part of an identifier. See Section 9.4, “User-Defined Variables”, for more information and examples of workarounds.

Special characters in database and table names are encoded in the corresponding file system names as described in Section 9.2.4, “Mapping of Identifiers to File Names”.

9.2.1 Identifier Length Limits

The following table describes the maximum length for each type of identifier.

Identifier Type	Maximum Length (characters)
Database	64 (includes NDB Cluster 8.0.18 and later)
Table	64 (includes NDB Cluster 8.0.18 and later)
Column	64
Index	64
Constraint	64
Stored Program	64
View	64
Tablespace	64
Server	64
Log File Group	64
Alias	256 (see exception following table)
Compound Statement Label	16
User-Defined Variable	64
Resource Group	64

Aliases for column names in `CREATE VIEW` statements are checked against the maximum column length of 64 characters (not the maximum alias length of 256 characters).

For constraint definitions that include no constraint name, the server internally generates a name derived from the associated table name. For example, internally generated foreign key and `CHECK` constraint names consist of the table name plus `_ibfk_` or `_chk_` and a number. If the table name is close to the length limit for constraint names, the additional characters required for the constraint name may cause that name to exceed the limit, resulting in an error.

Identifiers are stored using Unicode (UTF-8). This applies to identifiers in table definitions and to identifiers stored in the grant tables in the `mysql` database. The sizes of the identifier string columns in the grant tables are measured in characters. You can use multibyte characters without reducing the number of characters permitted for values stored in these columns.

Prior to NDB 8.0.18, NDB Cluster imposed a maximum length of 63 characters for names of databases and tables. As of NDB 8.0.18, this limitation is removed. See Section 23.2.7.11, “Previous NDB Cluster Issues Resolved in NDB Cluster 8.0”.

Values such as user name and host names in MySQL account names are strings rather than identifiers. For information about the maximum length of such values as stored in grant tables, see Grant Table Scope Column Properties.

9.2.2 Identifier Qualifiers

Object names may be unqualified or qualified. An unqualified name is permitted in contexts where interpretation of the name is unambiguous. A qualified name includes at least one qualifier to clarify the interpretive context by overriding a default context or providing missing context.

For example, this statement creates a table using the unqualified name `t1`:

```
CREATE TABLE t1 (i INT);
```

Because `t1` includes no qualifier to specify a database, the statement creates the table in the default database. If there is no default database, an error occurs.

This statement creates a table using the qualified name `db1.t1`:

```
CREATE TABLE db1.t1 (i INT);
```

Because `db1.t1` includes a database qualifier `db1`, the statement creates `t1` in the database named `db1`, regardless of the default database. The qualifier *must* be specified if there is no default database. The qualifier *may* be specified if there is a default database, to specify a database different from the default, or to make the database explicit if the default is the same as the one specified.

Qualifiers have these characteristics:

- An unqualified name consists of a single identifier. A qualified name consists of multiple identifiers.
- The components of a multiple-part name must be separated by period (`.`) characters. The initial parts of a multiple-part name act as qualifiers that affect the context within which to interpret the final identifier.
- The qualifier character is a separate token and need not be contiguous with the associated identifiers. For example, *`tbl_name.col_name`* and *`tbl_name . col_name`* are equivalent.
- If any components of a multiple-part name require quoting, quote them individually rather than quoting the name as a whole. For example, write `` `my-table`.`my-column` ``, not `` `my-table.my-column` ``.
- A reserved word that follows a period in a qualified name must be an identifier, so in that context it need not be quoted.

The permitted qualifiers for object names depend on the object type:

- A database name is fully qualified and takes no qualifier:

  ```
  CREATE DATABASE db1;
  ```

- A table, view, or stored program name may be given a database-name qualifier. Examples of unqualified and qualified names in `CREATE` statements:

  ```
  CREATE TABLE mytable ...;
  CREATE VIEW myview ...;
  CREATE PROCEDURE myproc ...;
  CREATE FUNCTION myfunc ...;
  CREATE EVENT myevent ...;

  CREATE TABLE mydb.mytable ...;
  CREATE VIEW mydb.myview ...;
  CREATE PROCEDURE mydb.myproc ...;
  CREATE FUNCTION mydb.myfunc ...;
  CREATE EVENT mydb.myevent ...;
  ```

- A trigger is associated with a table, so any qualifier applies to the table name:

```
CREATE TRIGGER mytrigger ... ON mytable ...;

CREATE TRIGGER mytrigger ... ON mydb.mytable ...;
```

- A column name may be given multiple qualifiers to indicate context in statements that reference it, as shown in the following table.

Column Reference	Meaning
col_name	Column *col_name* from whichever table used in the statement contains a column of that name
tbl_name.col_name	Column *col_name* from table *tbl_name* of the default database
db_name.tbl_name.col_name	Column *col_name* from table *tbl_name* of the database *db_name*

  In other words, a column name may be given a table-name qualifier, which itself may be given a database-name qualifier. Examples of unqualified and qualified column references in `SELECT` statements:

```
SELECT c1 FROM mytable
WHERE c2 > 100;

SELECT mytable.c1 FROM mytable
WHERE mytable.c2 > 100;

SELECT mydb.mytable.c1 FROM mydb.mytable
WHERE mydb.mytable.c2 > 100;
```

You need not specify a qualifier for an object reference in a statement unless the unqualified reference is ambiguous. Suppose that column `c1` occurs only in table `t1`, `c2` only in `t2`, and `c` in both `t1` and `t2`. Any unqualified reference to `c` is ambiguous in a statement that refers to both tables and must be qualified as `t1.c` or `t2.c` to indicate which table you mean:

```
SELECT c1, c2, t1.c FROM t1 INNER JOIN t2
WHERE t2.c > 100;
```

Similarly, to retrieve from a table `t` in database `db1` and from a table `t` in database `db2` in the same statement, you must qualify the table references: For references to columns in those tables, qualifiers are required only for column names that appear in both tables. Suppose that column `c1` occurs only in table `db1.t`, `c2` only in `db2.t`, and `c` in both `db1.t` and `db2.t`. In this case, `c` is ambiguous and must be qualified but `c1` and `c2` need not be:

```
SELECT c1, c2, db1.t.c FROM db1.t INNER JOIN db2.t
WHERE db2.t.c > 100;
```

Table aliases enable qualified column references to be written more simply:

```
SELECT c1, c2, t1.c FROM db1.t AS t1 INNER JOIN db2.t AS t2
WHERE t2.c > 100;
```

9.2.3 Identifier Case Sensitivity

In MySQL, databases correspond to directories within the data directory. Each table within a database corresponds to at least one file within the database directory (and possibly more, depending on the storage engine). Triggers also correspond to files. Consequently, the case sensitivity of the underlying operating system plays a part in the case sensitivity of database, table, and trigger names. This means such names are not case-sensitive in Windows, but are case-sensitive in most varieties of Unix. One notable exception is macOS, which is Unix-based but uses a default file system type (HFS+) that is not case-sensitive. However, macOS also supports UFS volumes, which are case-sensitive just as on any Unix. See Section 1.6.1, “MySQL Extensions to Standard SQL”. The `lower_case_table_names`

system variable also affects how the server handles identifier case sensitivity, as described later in this section.

Note

Although database, table, and trigger names are not case-sensitive on some platforms, you should not refer to one of these using different cases within the same statement. The following statement would not work because it refers to a table both as `my_table` and as `MY_TABLE`:

```
mysql> SELECT * FROM my_table WHERE MY_TABLE.col=1;
```

Partition, subpartition, column, index, stored routine, event, and resource group names are not case-sensitive on any platform, nor are column aliases.

However, names of logfile groups are case-sensitive. This differs from standard SQL.

By default, table aliases are case-sensitive on Unix, but not so on Windows or macOS. The following statement would not work on Unix, because it refers to the alias both as `a` and as `A`:

```
mysql> SELECT col_name FROM tbl_name AS a
       WHERE a.col_name = 1 OR A.col_name = 2;
```

However, this same statement is permitted on Windows. To avoid problems caused by such differences, it is best to adopt a consistent convention, such as always creating and referring to databases and tables using lowercase names. This convention is recommended for maximum portability and ease of use.

How table and database names are stored on disk and used in MySQL is affected by the `lower_case_table_names` system variable. `lower_case_table_names` can take the values shown in the following table. This variable does *not* affect case sensitivity of trigger identifiers. On Unix, the default value of `lower_case_table_names` is 0. On Windows, the default value is 1. On macOS, the default value is 2.

`lower_case_table_names` can only be configured when initializing the server. Changing the `lower_case_table_names` setting after the server is initialized is prohibited.

Value	Meaning
`0`	Table and database names are stored on disk using the lettercase specified in the `CREATE TABLE` or `CREATE DATABASE` statement. Name comparisons are case-sensitive. You should *not* set this variable to 0 if you are running MySQL on a system that has case-insensitive file names (such as Windows or macOS). If you force this variable to 0 with `--lower-case-table-names=0` on a case-insensitive file system and access `MyISAM` tablenames using different lettercases, index corruption may result.
`1`	Table names are stored in lowercase on disk and name comparisons are not case-sensitive. MySQL converts all table names to lowercase on storage and lookup. This behavior also applies to database names and table aliases.
`2`	Table and database names are stored on disk using the lettercase specified in the `CREATE TABLE` or `CREATE DATABASE` statement, but MySQL converts them to lowercase on lookup. Name comparisons are not case-sensitive. This works *only* on file systems that are not

Value	Meaning
	case-sensitive! `InnoDB` table names and view names are stored in lowercase, as for `lower_case_table_names=1`.

If you are using MySQL on only one platform, you do not normally have to use a `lower_case_table_names` setting other than the default. However, you may encounter difficulties if you want to transfer tables between platforms that differ in file system case sensitivity. For example, on Unix, you can have two different tables named `my_table` and `MY_TABLE`, but on Windows these two names are considered identical. To avoid data transfer problems arising from lettercase of database or table names, you have two options:

- Use `lower_case_table_names=1` on all systems. The main disadvantage with this is that when you use `SHOW TABLES` or `SHOW DATABASES`, you do not see the names in their original lettercase.
- Use `lower_case_table_names=0` on Unix and `lower_case_table_names=2` on Windows. This preserves the lettercase of database and table names. The disadvantage of this is that you must ensure that your statements always refer to your database and table names with the correct lettercase on Windows. If you transfer your statements to Unix, where lettercase is significant, they do not work if the lettercase is incorrect.

  Exception: If you are using `InnoDB` tables and you are trying to avoid these data transfer problems, you should use `lower_case_table_names=1` on all platforms to force names to be converted to lowercase.

Object names may be considered duplicates if their uppercase forms are equal according to a binary collation. That is true for names of cursors, conditions, procedures, functions, savepoints, stored routine parameters, stored program local variables, and plugins. It is not true for names of columns, constraints, databases, partitions, statements prepared with `PREPARE`, tables, triggers, users, and user-defined variables.

File system case sensitivity can affect searches in string columns of `INFORMATION_SCHEMA` tables. For more information, see Section 10.8.7, “Using Collation in INFORMATION_SCHEMA Searches”.

9.2.4 Mapping of Identifiers to File Names

There is a correspondence between database and table identifiers and names in the file system. For the basic structure, MySQL represents each database as a directory in the data directory, and depending upon the storage engine, each table may be represented by one or more files in the appropriate database directory.

For the data and index files, the exact representation on disk is storage engine specific. These files may be stored in the database directory, or the information may be stored in a separate file. `InnoDB` data is stored in the InnoDB data files. If you are using tablespaces with `InnoDB`, then the specific tablespace files you create are used instead.

Any character is legal in database or table identifiers except ASCII NUL (`X'00'`). MySQL encodes any characters that are problematic in the corresponding file system objects when it creates database directories or table files:

- Basic Latin letters (`a..zA..Z`), digits (`0..9`) and underscore (_) are encoded as is. Consequently, their case sensitivity directly depends on file system features.
- All other national letters from alphabets that have uppercase/lowercase mapping are encoded as shown in the following table. Values in the Code Range column are UCS-2 values.

Code Range	Pattern	Number	Used	Unused	Blocks
00C0..017F	[@][0..4][g..z]	5*20= 100	97	3	Latin-1 Supplement +

Code Range	Pattern	Number	Used	Unused	Blocks
					Latin Extended-A
0370..03FF	[@][5..9][g..z]	5*20= 100	88	12	Greek and Coptic
0400..052F	[@][g..z][0..6]	20*7= 140	137	3	Cyrillic + Cyrillic Supplement
0530..058F	[@][g..z][7..8]	20*2= 40	38	2	Armenian
2160..217F	[@][g..z][9]	20*1= 20	16	4	Number Forms
0180..02AF	[@][g..z][a..k]	20*11=220	203	17	Latin Extended-B + IPA Extensions
1E00..1EFF	[@][g..z][l..r]	20*7= 140	136	4	Latin Extended Additional
1F00..1FFF	[@][g..z][s..z]	20*8= 160	144	16	Greek Extended
....	[@][a..f][g..z]	6*20= 120	0	120	RESERVED
24B6..24E9	[@][@][a..z]	26	26	0	Enclosed Alphanumerics
FF21..FF5A	[@][a..z][@]	26	26	0	Halfwidth and Fullwidth forms

One of the bytes in the sequence encodes lettercase. For example: `LATIN CAPITAL LETTER A WITH GRAVE` is encoded as `@0G`, whereas `LATIN SMALL LETTER A WITH GRAVE` is encoded as `@0g`. Here the third byte (`G` or `g`) indicates lettercase. (On a case-insensitive file system, both letters are treated as the same.)

For some blocks, such as Cyrillic, the second byte determines lettercase. For other blocks, such as Latin1 Supplement, the third byte determines lettercase. If two bytes in the sequence are letters (as in Greek Extended), the leftmost letter character stands for lettercase. All other letter bytes must be in lowercase.

- All nonletter characters except underscore (_), as well as letters from alphabets that do not have uppercase/lowercase mapping (such as Hebrew) are encoded using hexadecimal representation using lowercase letters for hexadecimal digits `a..f`:

```
0x003F -> @003f
0xFFFF -> @ffff
```

The hexadecimal values correspond to character values in the `ucs2` double-byte character set.

On Windows, some names such as `nul`, `prn`, and `aux` are encoded by appending `@@@` to the name when the server creates the corresponding file or directory. This occurs on all platforms for portability of the corresponding database object between platforms.

The following names are reserved and appended with `@@@` if used in schema or table names:

- CON
- PRN
- AUX
- NUL
- COM1 through COM9

- LPT1 through LPT9

CLOCK$ is also a member of this group of reserved names, but is not appended with `@@@`, but `@0024` instead. That is, if CLOCK$ is used as a schema or table name, it is written to the file system as `CLOCK@0024`. The same is true for any use of $ (dollar sign) in a schema or table name; it is replaced with `@0024` on the filesystem.

Note

These names are also written to `INNODB_TABLES` in their appended forms, but are written to `TABLES` in their unappended form, as entered by the user.

9.2.5 Function Name Parsing and Resolution

MySQL supports built-in (native) functions, loadable functions, and stored functions. This section describes how the server recognizes whether the name of a built-in function is used as a function call or as an identifier, and how the server determines which function to use in cases when functions of different types exist with a given name.

- Built-In Function Name Parsing
- Function Name Resolution

Built-In Function Name Parsing

The parser uses default rules for parsing names of built-in functions. These rules can be changed by enabling the `IGNORE_SPACE` SQL mode.

When the parser encounters a word that is the name of a built-in function, it must determine whether the name signifies a function call or is instead a nonexpression reference to an identifier such as a table or column name. For example, in the following statements, the first reference to `count` is a function call, whereas the second reference is a table name:

```
SELECT COUNT(*) FROM mytable;
CREATE TABLE count (i INT);
```

The parser should recognize the name of a built-in function as indicating a function call only when parsing what is expected to be an expression. That is, in nonexpression context, function names are permitted as identifiers.

However, some built-in functions have special parsing or implementation considerations, so the parser uses the following rules by default to distinguish whether their names are being used as function calls or as identifiers in nonexpression context:

- To use the name as a function call in an expression, there must be no whitespace between the name and the following `(` parenthesis character.
- Conversely, to use the function name as an identifier, it must not be followed immediately by a parenthesis.

The requirement that function calls be written with no whitespace between the name and the parenthesis applies only to the built-in functions that have special considerations. `COUNT` is one such name. The `sql/lex.h` source file lists the names of these special functions for which following whitespace determines their interpretation: names defined by the `SYM_FN()` macro in the `symbols[]` array.

The following list names the functions in MySQL 8.0 that are affected by the `IGNORE_SPACE` setting and listed as special in the `sql/lex.h` source file. You may find it easiest to treat the no-whitespace requirement as applying to all function calls.

- `ADDDATE`

- BIT_AND
- BIT_OR
- BIT_XOR
- CAST
- COUNT
- CURDATE
- CURTIME
- DATE_ADD
- DATE_SUB
- EXTRACT
- GROUP_CONCAT
- MAX
- MID
- MIN
- NOW
- POSITION
- SESSION_USER
- STD
- STDDEV
- STDDEV_POP
- STDDEV_SAMP
- SUBDATE
- SUBSTR
- SUBSTRING
- SUM
- SYSDATE
- SYSTEM_USER
- TRIM
- VARIANCE
- VAR_POP
- VAR_SAMP

For functions not listed as special in `sql/lex.h`, whitespace does not matter. They are interpreted as function calls only when used in expression context and may be used freely as identifiers otherwise. `ASCII` is one such name. However, for these nonaffected function names, interpretation may vary in

expression context: *`func_name ()`* is interpreted as a built-in function if there is one with the given name; if not, *`func_name ()`* is interpreted as a loadable function or stored function if one exists with that name.

The `IGNORE_SPACE` SQL mode can be used to modify how the parser treats function names that are whitespace-sensitive:

- With `IGNORE_SPACE` disabled, the parser interprets the name as a function call when there is no whitespace between the name and the following parenthesis. This occurs even when the function name is used in nonexpression context:

  ```
  mysql> CREATE TABLE count(i INT);
  ERROR 1064 (42000): You have an error in your SQL syntax ...
  near 'count(i INT)'
  ```

  To eliminate the error and cause the name to be treated as an identifier, either use whitespace following the name or write it as a quoted identifier (or both):

  ```
  CREATE TABLE count (i INT);
  CREATE TABLE `count`(i INT);
  CREATE TABLE `count` (i INT);
  ```

- With `IGNORE_SPACE` enabled, the parser loosens the requirement that there be no whitespace between the function name and the following parenthesis. This provides more flexibility in writing function calls. For example, either of the following function calls are legal:

  ```
  SELECT COUNT(*) FROM mytable;
  SELECT COUNT (*) FROM mytable;
  ```

  However, enabling `IGNORE_SPACE` also has the side effect that the parser treats the affected function names as reserved words (see Section 9.3, “Keywords and Reserved Words”). This means that a space following the name no longer signifies its use as an identifier. The name can be used in function calls with or without following whitespace, but causes a syntax error in nonexpression context unless it is quoted. For example, with `IGNORE_SPACE` enabled, both of the following statements fail with a syntax error because the parser interprets `count` as a reserved word:

  ```
  CREATE TABLE count(i INT);
  CREATE TABLE count (i INT);
  ```

  To use the function name in nonexpression context, write it as a quoted identifier:

  ```
  CREATE TABLE `count`(i INT);
  CREATE TABLE `count` (i INT);
  ```

To enable the `IGNORE_SPACE` SQL mode, use this statement:

```
SET sql_mode = 'IGNORE_SPACE';
```

`IGNORE_SPACE` is also enabled by certain other composite modes such as `ANSI` that include it in their value:

```
SET sql_mode = 'ANSI';
```

Check Section 5.1.11, “Server SQL Modes”, to see which composite modes enable `IGNORE_SPACE`.

To minimize the dependency of SQL code on the `IGNORE_SPACE` setting, use these guidelines:

- Avoid creating loadable functions or stored functions that have the same name as a built-in function.
- Avoid using function names in nonexpression context. For example, these statements use `count` (one of the affected function names affected by `IGNORE_SPACE`), so they fail with or without whitespace following the name if `IGNORE_SPACE` is enabled:

  ```
  CREATE TABLE count(i INT);
  CREATE TABLE count (i INT);
  ```

If you must use a function name in nonexpression context, write it as a quoted identifier:

```
CREATE TABLE `count`(i INT);
CREATE TABLE `count` (i INT);
```

Function Name Resolution

The following rules describe how the server resolves references to function names for function creation and invocation:

- Built-in functions and loadable functions

  An error occurs if you try to create a loadable function with the same name as a built-in function.

  `IF NOT EXISTS` (available beginning with MySQL 8.0.29) has no effect in such cases. See Section 13.7.4.1, “CREATE FUNCTION Statement for Loadable Functions”, for more information.

- Built-in functions and stored functions

  It is possible to create a stored function with the same name as a built-in function, but to invoke the stored function it is necessary to qualify it with a schema name. For example, if you create a stored function named `PI` in the `test` schema, invoke it as `test.PI()` because the server resolves `PI()` without a qualifier as a reference to the built-in function. The server generates a warning if the stored function name collides with a built-in function name. The warning can be displayed with `SHOW WARNINGS`.

  `IF NOT EXISTS` (MySQL 8.0.29 and later) has no effect in such cases; see Section 13.1.17, “CREATE PROCEDURE and CREATE FUNCTION Statements”.

- Loadable functions and stored functions

  It is possible to create a stored function with the same name as an existing loadable function, or the other way around. The server generates a warning if a proposed stored function name collides with an existing loadable function name, or if a proposed loadable function name would be the same as that of an existing stored function. In either case, once both functions exist, it is necessary thereafter to qualify the stored function with a schema name when invoking it; the server assumes in such cases that the unqualified name refers to the loadable function.

  Beginning with MySQL 8.0.29, `IF NOT EXISTS` is supported with `CREATE FUNCTION` statements, but has no effect in such cases.

  Prior to MySQL 8.0.28, it was possible to create a stored function with the same name as an existing loadable function, but not the other way around (Bug #33301931).

The preceding function name resolution rules have implications for upgrading to versions of MySQL that implement new built-in functions:

- If you have already created a loadable function with a given name and upgrade MySQL to a version that implements a new built-in function with the same name, the loadable function becomes inaccessible. To correct this, use `DROP FUNCTION` to drop the loadable function and `CREATE FUNCTION` to re-create the loadable function with a different nonconflicting name. Then modify any affected code to use the new name.

- If a new version of MySQL implements a built-in function or loadable function with the same name as an existing stored function, you have two choices: Rename the stored function to use a nonconflicting name, or change any calls to the function that do not do so already to use a schema qualifier (*`schema_name.func_name()`* syntax). In either case, modify any affected code accordingly.

9.3 Keywords and Reserved Words

Keywords are words that have significance in SQL. Certain keywords, such as `SELECT`, `DELETE`, or `BIGINT`, are reserved and require special treatment for use as identifiers such as table and column names. This may also be true for the names of built-in functions.

Nonreserved keywords are permitted as identifiers without quoting. Reserved words are permitted as identifiers if you quote them as described in Section 9.2, “Schema Object Names”:

```
mysql> CREATE TABLE interval (begin INT, end INT);
ERROR 1064 (42000): You have an error in your SQL syntax ...
near 'interval (begin INT, end INT)'
```

`BEGIN` and `END` are keywords but not reserved, so their use as identifiers does not require quoting. `INTERVAL` is a reserved keyword and must be quoted to be used as an identifier:

```
mysql> CREATE TABLE `interval` (begin INT, end INT);
Query OK, 0 rows affected (0.01 sec)
```

Exception: A word that follows a period in a qualified name must be an identifier, so it need not be quoted even if it is reserved:

```
mysql> CREATE TABLE mydb.interval (begin INT, end INT);
Query OK, 0 rows affected (0.01 sec)
```

Names of built-in functions are permitted as identifiers but may require care to be used as such. For example, `COUNT` is acceptable as a column name. However, by default, no whitespace is permitted in function invocations between the function name and the following `(` character. This requirement enables the parser to distinguish whether the name is used in a function call or in nonfunction context. For further details on recognition of function names, see Section 9.2.5, “Function Name Parsing and Resolution”.

The `INFORMATION_SCHEMA.KEYWORDS` table lists the words considered keywords by MySQL and indicates whether they are reserved. See Section 26.3.17, “The INFORMATION_SCHEMA KEYWORDS Table”.

- MySQL 8.0 Keywords and Reserved Words
- MySQL 8.0 New Keywords and Reserved Words
- MySQL 8.0 Removed Keywords and Reserved Words

MySQL 8.0 Keywords and Reserved Words

The following list shows the keywords and reserved words in MySQL 8.0, along with changes to individual words from version to version. Reserved keywords are marked with (R). In addition, `_FILENAME` is reserved.

At some point, you might upgrade to a higher version, so it is a good idea to have a look at future reserved words, too. You can find these in the manuals that cover higher versions of MySQL. Most of the reserved words in the list are forbidden by standard SQL as column or table names (for example, `GROUP`). A few are reserved because MySQL needs them and uses a `yacc` parser.

A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z

A

- `ACCESSIBLE` (R)
- `ACCOUNT`
- `ACTION`
- `ACTIVE`; added in 8.0.14 (nonreserved)

- ADD (R)
- ADMIN; became nonreserved in 8.0.12
- AFTER
- AGAINST
- AGGREGATE
- ALGORITHM
- ALL (R)
- ALTER (R)
- ALWAYS
- ANALYSE; removed in 8.0.1
- ANALYZE (R)
- AND (R)
- ANY
- ARRAY; added in 8.0.17 (reserved); became nonreserved in 8.0.19
- AS (R)
- ASC (R)
- ASCII
- ASENSITIVE (R)
- AT
- ATTRIBUTE; added in 8.0.21 (nonreserved)
- AUTHENTICATION; added in 8.0.27 (nonreserved)
- AUTOEXTEND_SIZE
- AUTO_INCREMENT
- AVG
- AVG_ROW_LENGTH

B

- BACKUP
- BEFORE (R)
- BEGIN
- BETWEEN (R)
- BIGINT (R)
- BINARY (R)
- BINLOG

- BIT
- BLOB (R)
- BLOCK
- BOOL
- BOOLEAN
- BOTH (R)
- BTREE
- BUCKETS; added in 8.0.2 (nonreserved)
- BULK; added in 8.0.32 (nonreserved)
- BY (R)
- BYTE

C

- CACHE
- CALL (R)
- CASCADE (R)
- CASCADED
- CASE (R)
- CATALOG_NAME
- CHAIN
- CHALLENGE_RESPONSE; added in 8.0.27 (nonreserved)
- CHANGE (R)
- CHANGED
- CHANNEL
- CHAR (R)
- CHARACTER (R)
- CHARSET
- CHECK (R)
- CHECKSUM
- CIPHER
- CLASS_ORIGIN
- CLIENT
- CLONE; added in 8.0.3 (nonreserved)
- CLOSE

- COALESCE
- CODE
- COLLATE (R)
- COLLATION
- COLUMN (R)
- COLUMNS
- COLUMN_FORMAT
- COLUMN_NAME
- COMMENT
- COMMIT
- COMMITTED
- COMPACT
- COMPLETION
- COMPONENT
- COMPRESSED
- COMPRESSION
- CONCURRENT
- CONDITION (R)
- CONNECTION
- CONSISTENT
- CONSTRAINT (R)
- CONSTRAINT_CATALOG
- CONSTRAINT_NAME
- CONSTRAINT_SCHEMA
- CONTAINS
- CONTEXT
- CONTINUE (R)
- CONVERT (R)
- CPU
- CREATE (R)
- CROSS (R)
- CUBE (R); became reserved in 8.0.1
- CUME_DIST (R); added in 8.0.2 (reserved)

- CURRENT
- CURRENT_DATE (R)
- CURRENT_TIME (R)
- CURRENT_TIMESTAMP (R)
- CURRENT_USER (R)
- CURSOR (R)
- CURSOR_NAME

D

- DATA
- DATABASE (R)
- DATABASES (R)
- DATAFILE
- DATE
- DATETIME
- DAY
- DAY_HOUR (R)
- DAY_MICROSECOND (R)
- DAY_MINUTE (R)
- DAY_SECOND (R)
- DEALLOCATE
- DEC (R)
- DECIMAL (R)
- DECLARE (R)
- DEFAULT (R)
- DEFAULT_AUTH
- DEFINER
- DEFINITION; added in 8.0.4 (nonreserved)
- DELAYED (R)
- DELAY_KEY_WRITE
- DELETE (R)
- DENSE_RANK (R); added in 8.0.2 (reserved)
- DESC (R)
- DESCRIBE (R)

- DESCRIPTION; added in 8.0.4 (nonreserved)
- DES_KEY_FILE; removed in 8.0.3
- DETERMINISTIC (R)
- DIAGNOSTICS
- DIRECTORY
- DISABLE
- DISCARD
- DISK
- DISTINCT (R)
- DISTINCTROW (R)
- DIV (R)
- DO
- DOUBLE (R)
- DROP (R)
- DUAL (R)
- DUMPFILE
- DUPLICATE
- DYNAMIC

E

- EACH (R)
- ELSE (R)
- ELSEIF (R)
- EMPTY (R); added in 8.0.4 (reserved)
- ENABLE
- ENCLOSED (R)
- ENCRYPTION
- END
- ENDS
- ENFORCED; added in 8.0.16 (nonreserved)
- ENGINE
- ENGINES
- ENGINE_ATTRIBUTE; added in 8.0.21 (nonreserved)
- ENUM

- ERROR
- ERRORS
- ESCAPE
- ESCAPED (R)
- EVENT
- EVENTS
- EVERY
- EXCEPT (R)
- EXCHANGE
- EXCLUDE; added in 8.0.2 (nonreserved)
- EXECUTE
- EXISTS (R)
- EXIT (R)
- EXPANSION
- EXPIRE
- EXPLAIN (R)
- EXPORT
- EXTENDED
- EXTENT_SIZE

F

- FACTOR; added in 8.0.27 (nonreserved)
- FAILED_LOGIN_ATTEMPTS; added in 8.0.19 (nonreserved)
- FALSE (R)
- FAST
- FAULTS
- FETCH (R)
- FIELDS
- FILE
- FILE_BLOCK_SIZE
- FILTER
- FINISH; added in 8.0.27 (nonreserved)
- FIRST
- FIRST_VALUE (R); added in 8.0.2 (reserved)

- FIXED
- FLOAT (R)
- FLOAT4 (R)
- FLOAT8 (R)
- FLUSH
- FOLLOWING; added in 8.0.2 (nonreserved)
- FOLLOWS
- FOR (R)
- FORCE (R)
- FOREIGN (R)
- FORMAT
- FOUND
- FROM (R)
- FULL
- FULLTEXT (R)
- FUNCTION (R); became reserved in 8.0.1

G

- GENERAL
- GENERATE; added in 8.0.32 (nonreserved)
- GENERATED (R)
- GEOMCOLLECTION; added in 8.0.11 (nonreserved)
- GEOMETRY
- GEOMETRYCOLLECTION
- GET (R)
- GET_FORMAT
- GET_MASTER_PUBLIC_KEY; added in 8.0.4 (reserved); became nonreserved in 8.0.11
- GET_SOURCE_PUBLIC_KEY; added in 8.0.23 (nonreserved)
- GLOBAL
- GRANT (R)
- GRANTS
- GROUP (R)
- GROUPING (R); added in 8.0.1 (reserved)
- GROUPS (R); added in 8.0.2 (reserved)

- GROUP_REPLICATION
- GTID_ONLY; added in 8.0.27 (nonreserved)

H

- HANDLER
- HASH
- HAVING (R)
- HELP
- HIGH_PRIORITY (R)
- HISTOGRAM; added in 8.0.2 (nonreserved)
- HISTORY; added in 8.0.3 (nonreserved)
- HOST
- HOSTS
- HOUR
- HOUR_MICROSECOND (R)
- HOUR_MINUTE (R)
- HOUR_SECOND (R)

I

- IDENTIFIED
- IF (R)
- IGNORE (R)
- IGNORE_SERVER_IDS
- IMPORT
- IN (R)
- INACTIVE; added in 8.0.14 (nonreserved)
- INDEX (R)
- INDEXES
- INFILE (R)
- INITIAL; added in 8.0.27 (nonreserved)
- INITIAL_SIZE
- INITIATE; added in 8.0.27 (nonreserved)
- INNER (R)
- INOUT (R)
- INSENSITIVE (R)

- INSERT (R)
- INSERT_METHOD
- INSTALL
- INSTANCE
- INT (R)
- INT1 (R)
- INT2 (R)
- INT3 (R)
- INT4 (R)
- INT8 (R)
- INTEGER (R)
- INTERSECT (R); added in 8.0.31 (reserved)
- INTERVAL (R)
- INTO (R)
- INVISIBLE
- INVOKER
- IO
- IO_AFTER_GTIDS (R)
- IO_BEFORE_GTIDS (R)
- IO_THREAD
- IPC
- IS (R)
- ISOLATION
- ISSUER
- ITERATE (R)

J

- JOIN (R)
- JSON
- JSON_TABLE (R); added in 8.0.4 (reserved)
- JSON_VALUE; added in 8.0.21 (nonreserved)

K

- KEY (R)
- KEYRING; added in 8.0.24 (nonreserved)

- KEYS (R)
- KEY_BLOCK_SIZE
- KILL (R)

L

- LAG (R); added in 8.0.2 (reserved)
- LANGUAGE
- LAST
- LAST_VALUE (R); added in 8.0.2 (reserved)
- LATERAL (R); added in 8.0.14 (reserved)
- LEAD (R); added in 8.0.2 (reserved)
- LEADING (R)
- LEAVE (R)
- LEAVES
- LEFT (R)
- LESS
- LEVEL
- LIKE (R)
- LIMIT (R)
- LINEAR (R)
- LINES (R)
- LINESTRING
- LIST
- LOAD (R)
- LOCAL
- LOCALTIME (R)
- LOCALTIMESTAMP (R)
- LOCK (R)
- LOCKED; added in 8.0.1 (nonreserved)
- LOCKS
- LOGFILE
- LOGS
- LONG (R)
- LONGBLOB (R)

- LONGTEXT (R)
- LOOP (R)
- LOW_PRIORITY (R)

M

- MASTER
- MASTER_AUTO_POSITION
- MASTER_BIND (R)
- MASTER_COMPRESSION_ALGORITHMS; added in 8.0.18 (nonreserved)
- MASTER_CONNECT_RETRY
- MASTER_DELAY
- MASTER_HEARTBEAT_PERIOD
- MASTER_HOST
- MASTER_LOG_FILE
- MASTER_LOG_POS
- MASTER_PASSWORD
- MASTER_PORT
- MASTER_PUBLIC_KEY_PATH; added in 8.0.4 (nonreserved)
- MASTER_RETRY_COUNT
- MASTER_SERVER_ID; removed in 8.0.23
- MASTER_SSL
- MASTER_SSL_CA
- MASTER_SSL_CAPATH
- MASTER_SSL_CERT
- MASTER_SSL_CIPHER
- MASTER_SSL_CRL
- MASTER_SSL_CRLPATH
- MASTER_SSL_KEY
- MASTER_SSL_VERIFY_SERVER_CERT (R)
- MASTER_TLS_CIPHERSUITES; added in 8.0.19 (nonreserved)
- MASTER_TLS_VERSION
- MASTER_USER
- MASTER_ZSTD_COMPRESSION_LEVEL; added in 8.0.18 (nonreserved)
- MATCH (R)

- MAXVALUE (R)
- MAX_CONNECTIONS_PER_HOUR
- MAX_QUERIES_PER_HOUR
- MAX_ROWS
- MAX_SIZE
- MAX_UPDATES_PER_HOUR
- MAX_USER_CONNECTIONS
- MEDIUM
- MEDIUMBLOB (R)
- MEDIUMINT (R)
- MEDIUMTEXT (R)
- MEMBER; added in 8.0.17 (reserved); became nonreserved in 8.0.19
- MEMORY
- MERGE
- MESSAGE_TEXT
- MICROSECOND
- MIDDLEINT (R)
- MIGRATE
- MINUTE
- MINUTE_MICROSECOND (R)
- MINUTE_SECOND (R)
- MIN_ROWS
- MOD (R)
- MODE
- MODIFIES (R)
- MODIFY
- MONTH
- MULTILINESTRING
- MULTIPOINT
- MULTIPOLYGON
- MUTEX
- MYSQL_ERRNO

N

- NAME
- NAMES
- NATIONAL
- NATURAL (R)
- NCHAR
- NDB
- NDBCLUSTER
- NESTED; added in 8.0.4 (nonreserved)
- NETWORK_NAMESPACE; added in 8.0.16 (nonreserved)
- NEVER
- NEW
- NEXT
- NO
- NODEGROUP
- NONE
- NOT (R)
- NOWAIT; added in 8.0.1 (nonreserved)
- NO_WAIT
- NO_WRITE_TO_BINLOG (R)
- NTH_VALUE (R); added in 8.0.2 (reserved)
- NTILE (R); added in 8.0.2 (reserved)
- NULL (R)
- NULLS; added in 8.0.2 (nonreserved)
- NUMBER
- NUMERIC (R)
- NVARCHAR

O

- OF (R); added in 8.0.1 (reserved)
- OFF; added in 8.0.20 (nonreserved)
- OFFSET
- OJ; added in 8.0.16 (nonreserved)
- OLD; added in 8.0.14 (nonreserved)
- ON (R)

- ONE
- ONLY
- OPEN
- OPTIMIZE (R)
- OPTIMIZER_COSTS (R)
- OPTION (R)
- OPTIONAL; added in 8.0.13 (nonreserved)
- OPTIONALLY (R)
- OPTIONS
- OR (R)
- ORDER (R)
- ORDINALITY; added in 8.0.4 (nonreserved)
- ORGANIZATION; added in 8.0.4 (nonreserved)
- OTHERS; added in 8.0.2 (nonreserved)
- OUT (R)
- OUTER (R)
- OUTFILE (R)
- OVER (R); added in 8.0.2 (reserved)
- OWNER

P

- PACK_KEYS
- PAGE
- PARSER
- PARTIAL
- PARTITION (R)
- PARTITIONING
- PARTITIONS
- PASSWORD
- PASSWORD_LOCK_TIME; added in 8.0.19 (nonreserved)
- PATH; added in 8.0.4 (nonreserved)
- PERCENT_RANK (R); added in 8.0.2 (reserved)
- PERSIST; became nonreserved in 8.0.16
- PERSIST_ONLY; added in 8.0.2 (reserved); became nonreserved in 8.0.16

- PHASE
- PLUGIN
- PLUGINS
- PLUGIN_DIR
- POINT
- POLYGON
- PORT
- PRECEDES
- PRECEDING; added in 8.0.2 (nonreserved)
- PRECISION (R)
- PREPARE
- PRESERVE
- PREV
- PRIMARY (R)
- PRIVILEGES
- PRIVILEGE_CHECKS_USER; added in 8.0.18 (nonreserved)
- PROCEDURE (R)
- PROCESS; added in 8.0.11 (nonreserved)
- PROCESSLIST
- PROFILE
- PROFILES
- PROXY
- PURGE (R)

Q

- QUARTER
- QUERY
- QUICK

R

- RANDOM; added in 8.0.18 (nonreserved)
- RANGE (R)
- RANK (R); added in 8.0.2 (reserved)
- READ (R)
- READS (R)

- READ_ONLY
- READ_WRITE (R)
- REAL (R)
- REBUILD
- RECOVER
- RECURSIVE (R); added in 8.0.1 (reserved)
- REDOFILE; removed in 8.0.3
- REDO_BUFFER_SIZE
- REDUNDANT
- REFERENCE; added in 8.0.4 (nonreserved)
- REFERENCES (R)
- REGEXP (R)
- REGISTRATION; added in 8.0.27 (nonreserved)
- RELAY
- RELAYLOG
- RELAY_LOG_FILE
- RELAY_LOG_POS
- RELAY_THREAD
- RELEASE (R)
- RELOAD
- REMOTE; added in 8.0.3 (nonreserved); removed in 8.0.14
- REMOVE
- RENAME (R)
- REORGANIZE
- REPAIR
- REPEAT (R)
- REPEATABLE
- REPLACE (R)
- REPLICA; added in 8.0.22 (nonreserved)
- REPLICAS; added in 8.0.22 (nonreserved)
- REPLICATE_DO_DB
- REPLICATE_DO_TABLE
- REPLICATE_IGNORE_DB

- REPLICATE_IGNORE_TABLE
- REPLICATE_REWRITE_DB
- REPLICATE_WILD_DO_TABLE
- REPLICATE_WILD_IGNORE_TABLE
- REPLICATION
- REQUIRE (R)
- REQUIRE_ROW_FORMAT; added in 8.0.19 (nonreserved)
- RESET
- RESIGNAL (R)
- RESOURCE; added in 8.0.3 (nonreserved)
- RESPECT; added in 8.0.2 (nonreserved)
- RESTART; added in 8.0.4 (nonreserved)
- RESTORE
- RESTRICT (R)
- RESUME
- RETAIN; added in 8.0.14 (nonreserved)
- RETURN (R)
- RETURNED_SQLSTATE
- RETURNING; added in 8.0.21 (nonreserved)
- RETURNS
- REUSE; added in 8.0.3 (nonreserved)
- REVERSE
- REVOKE (R)
- RIGHT (R)
- RLIKE (R)
- ROLE; became nonreserved in 8.0.1
- ROLLBACK
- ROLLUP
- ROTATE
- ROUTINE
- ROW (R); became reserved in 8.0.2
- ROWS (R); became reserved in 8.0.2
- ROW_COUNT

- ROW_FORMAT
- ROW_NUMBER (R); added in 8.0.2 (reserved)
- RTREE

S

- SAVEPOINT
- SCHEDULE
- SCHEMA (R)
- SCHEMAS (R)
- SCHEMA_NAME
- SECOND
- SECONDARY; added in 8.0.16 (nonreserved)
- SECONDARY_ENGINE; added in 8.0.13 (nonreserved)
- SECONDARY_ENGINE_ATTRIBUTE; added in 8.0.21 (nonreserved)
- SECONDARY_LOAD; added in 8.0.13 (nonreserved)
- SECONDARY_UNLOAD; added in 8.0.13 (nonreserved)
- SECOND_MICROSECOND (R)
- SECURITY
- SELECT (R)
- SENSITIVE (R)
- SEPARATOR (R)
- SERIAL
- SERIALIZABLE
- SERVER
- SESSION
- SET (R)
- SHARE
- SHOW (R)
- SHUTDOWN
- SIGNAL (R)
- SIGNED
- SIMPLE
- SKIP; added in 8.0.1 (nonreserved)
- SLAVE

- SLOW
- SMALLINT (R)
- SNAPSHOT
- SOCKET
- SOME
- SONAME
- SOUNDS
- SOURCE
- SOURCE_AUTO_POSITION; added in 8.0.23 (nonreserved)
- SOURCE_BIND; added in 8.0.23 (nonreserved)
- SOURCE_COMPRESSION_ALGORITHMS; added in 8.0.23 (nonreserved)
- SOURCE_CONNECT_RETRY; added in 8.0.23 (nonreserved)
- SOURCE_DELAY; added in 8.0.23 (nonreserved)
- SOURCE_HEARTBEAT_PERIOD; added in 8.0.23 (nonreserved)
- SOURCE_HOST; added in 8.0.23 (nonreserved)
- SOURCE_LOG_FILE; added in 8.0.23 (nonreserved)
- SOURCE_LOG_POS; added in 8.0.23 (nonreserved)
- SOURCE_PASSWORD; added in 8.0.23 (nonreserved)
- SOURCE_PORT; added in 8.0.23 (nonreserved)
- SOURCE_PUBLIC_KEY_PATH; added in 8.0.23 (nonreserved)
- SOURCE_RETRY_COUNT; added in 8.0.23 (nonreserved)
- SOURCE_SSL; added in 8.0.23 (nonreserved)
- SOURCE_SSL_CA; added in 8.0.23 (nonreserved)
- SOURCE_SSL_CAPATH; added in 8.0.23 (nonreserved)
- SOURCE_SSL_CERT; added in 8.0.23 (nonreserved)
- SOURCE_SSL_CIPHER; added in 8.0.23 (nonreserved)
- SOURCE_SSL_CRL; added in 8.0.23 (nonreserved)
- SOURCE_SSL_CRLPATH; added in 8.0.23 (nonreserved)
- SOURCE_SSL_KEY; added in 8.0.23 (nonreserved)
- SOURCE_SSL_VERIFY_SERVER_CERT; added in 8.0.23 (nonreserved)
- SOURCE_TLS_CIPHERSUITES; added in 8.0.23 (nonreserved)
- SOURCE_TLS_VERSION; added in 8.0.23 (nonreserved)
- SOURCE_USER; added in 8.0.23 (nonreserved)

- SOURCE_ZSTD_COMPRESSION_LEVEL; added in 8.0.23 (nonreserved)
- SPATIAL (R)
- SPECIFIC (R)
- SQL (R)
- SQLEXCEPTION (R)
- SQLSTATE (R)
- SQLWARNING (R)
- SQL_AFTER_GTIDS
- SQL_AFTER_MTS_GAPS
- SQL_BEFORE_GTIDS
- SQL_BIG_RESULT (R)
- SQL_BUFFER_RESULT
- SQL_CACHE; removed in 8.0.3
- SQL_CALC_FOUND_ROWS (R)
- SQL_NO_CACHE
- SQL_SMALL_RESULT (R)
- SQL_THREAD
- SQL_TSI_DAY
- SQL_TSI_HOUR
- SQL_TSI_MINUTE
- SQL_TSI_MONTH
- SQL_TSI_QUARTER
- SQL_TSI_SECOND
- SQL_TSI_WEEK
- SQL_TSI_YEAR
- SRID; added in 8.0.3 (nonreserved)
- SSL (R)
- STACKED
- START
- STARTING (R)
- STARTS
- STATS_AUTO_RECALC
- STATS_PERSISTENT

- STATS_SAMPLE_PAGES
- STATUS
- STOP
- STORAGE
- STORED (R)
- STRAIGHT_JOIN (R)
- STREAM; added in 8.0.20 (nonreserved)
- STRING
- SUBCLASS_ORIGIN
- SUBJECT
- SUBPARTITION
- SUBPARTITIONS
- SUPER
- SUSPEND
- SWAPS
- SWITCHES
- SYSTEM (R); added in 8.0.3 (reserved)

T

- TABLE (R)
- TABLES
- TABLESPACE
- TABLE_CHECKSUM
- TABLE_NAME
- TEMPORARY
- TEMPTABLE
- TERMINATED (R)
- TEXT
- THAN
- THEN (R)
- THREAD_PRIORITY; added in 8.0.3 (nonreserved)
- TIES; added in 8.0.2 (nonreserved)
- TIME
- TIMESTAMP

- TIMESTAMPADD
- TIMESTAMPDIFF
- TINYBLOB (R)
- TINYINT (R)
- TINYTEXT (R)
- TLS; added in 8.0.21 (nonreserved)
- TO (R)
- TRAILING (R)
- TRANSACTION
- TRIGGER (R)
- TRIGGERS
- TRUE (R)
- TRUNCATE
- TYPE
- TYPES

U

- UNBOUNDED; added in 8.0.2 (nonreserved)
- UNCOMMITTED
- UNDEFINED
- UNDO (R)
- UNDOFILE
- UNDO_BUFFER_SIZE
- UNICODE
- UNINSTALL
- UNION (R)
- UNIQUE (R)
- UNKNOWN
- UNLOCK (R)
- UNREGISTER; added in 8.0.27 (nonreserved)
- UNSIGNED (R)
- UNTIL
- UPDATE (R)
- UPGRADE

- URL; added in 8.0.32 (nonreserved)
- USAGE (R)
- USE (R)
- USER
- USER_RESOURCES
- USE_FRM
- USING (R)
- UTC_DATE (R)
- UTC_TIME (R)
- UTC_TIMESTAMP (R)

V

- VALIDATION
- VALUE
- VALUES (R)
- VARBINARY (R)
- VARCHAR (R)
- VARCHARACTER (R)
- VARIABLES
- VARYING (R)
- VCPU; added in 8.0.3 (nonreserved)
- VIEW
- VIRTUAL (R)
- VISIBLE

W

- WAIT
- WARNINGS
- WEEK
- WEIGHT_STRING
- WHEN (R)
- WHERE (R)
- WHILE (R)
- WINDOW (R); added in 8.0.2 (reserved)
- WITH (R)

- WITHOUT
- WORK
- WRAPPER
- WRITE (R)

X

- X509
- XA
- XID
- XML
- XOR (R)

Y

- YEAR
- YEAR_MONTH (R)

Z

- ZEROFILL (R)
- ZONE; added in 8.0.22 (nonreserved)

MySQL 8.0 New Keywords and Reserved Words

The following list shows the keywords and reserved words that are added in MySQL 8.0, compared to MySQL 5.7. Reserved keywords are marked with (R).

A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | R | S | T | U | V | W | Z

A

- ACTIVE
- ADMIN
- ARRAY
- ATTRIBUTE
- AUTHENTICATION

B

- BUCKETS
- BULK

C

- CHALLENGE_RESPONSE
- CLONE
- COMPONENT

- CUME_DIST (R)

D

- DEFINITION
- DENSE_RANK (R)
- DESCRIPTION

E

- EMPTY (R)
- ENFORCED
- ENGINE_ATTRIBUTE
- EXCEPT (R)
- EXCLUDE

F

- FACTOR
- FAILED_LOGIN_ATTEMPTS
- FINISH
- FIRST_VALUE (R)
- FOLLOWING

G

- GENERATE
- GEOMCOLLECTION
- GET_MASTER_PUBLIC_KEY
- GET_SOURCE_PUBLIC_KEY
- GROUPING (R)
- GROUPS (R)
- GTID_ONLY

H

- HISTOGRAM
- HISTORY

I

- INACTIVE
- INITIAL
- INITIATE
- INTERSECT (R)

- INVISIBLE

J

- JSON_TABLE (R)
- JSON_VALUE

K

- KEYRING

L

- LAG (R)
- LAST_VALUE (R)
- LATERAL (R)
- LEAD (R)
- LOCKED

M

- MASTER_COMPRESSION_ALGORITHMS
- MASTER_PUBLIC_KEY_PATH
- MASTER_TLS_CIPHERSUITES
- MASTER_ZSTD_COMPRESSION_LEVEL
- MEMBER

N

- NESTED
- NETWORK_NAMESPACE
- NOWAIT
- NTH_VALUE (R)
- NTILE (R)
- NULLS

O

- OF (R)
- OFF
- OJ
- OLD
- OPTIONAL
- ORDINALITY
- ORGANIZATION

- OTHERS
- OVER (R)

P

- PASSWORD_LOCK_TIME
- PATH
- PERCENT_RANK (R)
- PERSIST
- PERSIST_ONLY
- PRECEDING
- PRIVILEGE_CHECKS_USER
- PROCESS

R

- RANDOM
- RANK (R)
- RECURSIVE (R)
- REFERENCE
- REGISTRATION
- REPLICA
- REPLICAS
- REQUIRE_ROW_FORMAT
- RESOURCE
- RESPECT
- RESTART
- RETAIN
- RETURNING
- REUSE
- ROLE
- ROW_NUMBER (R)

S

- SECONDARY
- SECONDARY_ENGINE
- SECONDARY_ENGINE_ATTRIBUTE
- SECONDARY_LOAD

- SECONDARY_UNLOAD
- SKIP
- SOURCE_AUTO_POSITION
- SOURCE_BIND
- SOURCE_COMPRESSION_ALGORITHMS
- SOURCE_CONNECT_RETRY
- SOURCE_DELAY
- SOURCE_HEARTBEAT_PERIOD
- SOURCE_HOST
- SOURCE_LOG_FILE
- SOURCE_LOG_POS
- SOURCE_PASSWORD
- SOURCE_PORT
- SOURCE_PUBLIC_KEY_PATH
- SOURCE_RETRY_COUNT
- SOURCE_SSL
- SOURCE_SSL_CA
- SOURCE_SSL_CAPATH
- SOURCE_SSL_CERT
- SOURCE_SSL_CIPHER
- SOURCE_SSL_CRL
- SOURCE_SSL_CRLPATH
- SOURCE_SSL_KEY
- SOURCE_SSL_VERIFY_SERVER_CERT
- SOURCE_TLS_CIPHERSUITES
- SOURCE_TLS_VERSION
- SOURCE_USER
- SOURCE_ZSTD_COMPRESSION_LEVEL
- SRID
- STREAM
- SYSTEM (R)

T

- THREAD_PRIORITY

- TIES
- TLS

U

- UNBOUNDED
- UNREGISTER
- URL

V

- VCPU
- VISIBLE

W

- WINDOW (R)

Z

- ZONE

MySQL 8.0 Removed Keywords and Reserved Words

The following list shows the keywords and reserved words that are removed in MySQL 8.0, compared to MySQL 5.7. Reserved keywords are marked with (R).

- ANALYSE
- DES_KEY_FILE
- MASTER_SERVER_ID
- PARSE_GCOL_EXPR
- REDOFILE
- SQL_CACHE

9.4 User-Defined Variables

You can store a value in a user-defined variable in one statement and refer to it later in another statement. This enables you to pass values from one statement to another.

User variables are written as *@var_name*, where the variable name *var_name* consists of alphanumeric characters, ., _, and $. A user variable name can contain other characters if you quote it as a string or identifier (for example, @'my-var', @"my-var", or @`my-var`).

User-defined variables are session specific. A user variable defined by one client cannot be seen or used by other clients. (Exception: A user with access to the Performance Schema user_variables_by_thread table can see all user variables for all sessions.) All variables for a given client session are automatically freed when that client exits.

User variable names are not case-sensitive. Names have a maximum length of 64 characters.

One way to set a user-defined variable is by issuing a SET statement:

```
SET @var_name = expr [, @var_name = expr] ...
```

For SET, either = or := can be used as the assignment operator.

User variables can be assigned a value from a limited set of data types: integer, decimal, floating-point, binary or nonbinary string, or NULL value. Assignment of decimal and real values does not preserve the precision or scale of the value. A value of a type other than one of the permissible types is converted to a permissible type. For example, a value having a temporal or spatial data type is converted to a binary string. A value having the JSON data type is converted to a string with a character set of utf8mb4 and a collation of utf8mb4_bin.

If a user variable is assigned a nonbinary (character) string value, it has the same character set and collation as the string. The coercibility of user variables is implicit. (This is the same coercibility as for table column values.)

Hexadecimal or bit values assigned to user variables are treated as binary strings. To assign a hexadecimal or bit value as a number to a user variable, use it in numeric context. For example, add 0 or use CAST(... AS UNSIGNED):

```
mysql> SET @v1 = X'41';
mysql> SET @v2 = X'41'+0;
mysql> SET @v3 = CAST(X'41' AS UNSIGNED);
mysql> SELECT @v1, @v2, @v3;
+------+------+------+
| @v1  | @v2  | @v3  |
+------+------+------+
| A    |   65 |   65 |
+------+------+------+
mysql> SET @v1 = b'1000001';
mysql> SET @v2 = b'1000001'+0;
mysql> SET @v3 = CAST(b'1000001' AS UNSIGNED);
mysql> SELECT @v1, @v2, @v3;
+------+------+------+
| @v1  | @v2  | @v3  |
+------+------+------+
| A    |   65 |   65 |
+------+------+------+
```

If the value of a user variable is selected in a result set, it is returned to the client as a string.

If you refer to a variable that has not been initialized, it has a value of NULL and a type of string.

Beginning with MySQL 8.0.22, a reference to a user variable in a prepared statement has its type determined when the statement is first prepared, and retains this type each time the statement is executed thereafter. Similarly, the type of a user variable employed in a statement within a stored procedure is determined the first time the stored procedure is invoked, and retains this type with each subsequent invocation.

User variables may be used in most contexts where expressions are permitted. This does not currently include contexts that explicitly require a literal value, such as in the LIMIT clause of a SELECT statement, or the IGNORE *N* LINES clause of a LOAD DATA statement.

Previous releases of MySQL made it possible to assign a value to a user variable in statements other than SET. This functionality is supported in MySQL 8.0 for backward compatibility but is subject to removal in a future release of MySQL.

When making an assignment in this way, you must use := as the assignment operator; = is treated as the comparison operator in statements other than SET.

The order of evaluation for expressions involving user variables is undefined. For example, there is no guarantee that SELECT @a, @a:=@a+1 evaluates @a first and then performs the assignment.

In addition, the default result type of a variable is based on its type at the beginning of the statement. This may have unintended effects if a variable holds a value of one type at the beginning of a statement in which it is also assigned a new value of a different type.

To avoid problems with this behavior, either do not assign a value to and read the value of the same variable within a single statement, or else set the variable to `0`, `0.0`, or `''` to define its type before you use it.

`HAVING`, `GROUP BY`, and `ORDER BY`, when referring to a variable that is assigned a value in the select expression list do not work as expected because the expression is evaluated on the client and thus can use stale column values from a previous row.

User variables are intended to provide data values. They cannot be used directly in an SQL statement as an identifier or as part of an identifier, such as in contexts where a table or database name is expected, or as a reserved word such as `SELECT`. This is true even if the variable is quoted, as shown in the following example:

```
mysql> SELECT c1 FROM t;
+----+
| c1 |
+----+
|  0 |
+----+
|  1 |
+----+
2 rows in set (0.00 sec)

mysql> SET @col = "c1";
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT @col FROM t;
+------+
| @col |
+------+
| c1   |
+------+
1 row in set (0.00 sec)

mysql> SELECT `@col` FROM t;
ERROR 1054 (42S22): Unknown column '@col' in 'field list'

mysql> SET @col = "`c1`";
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT @col FROM t;
+------+
| @col |
+------+
| `c1` |
+------+
1 row in set (0.00 sec)
```

An exception to this principle that user variables cannot be used to provide identifiers, is when you are constructing a string for use as a prepared statement to execute later. In this case, user variables can be used to provide any part of the statement. The following example illustrates how this can be done:

```
mysql> SET @c = "c1";
Query OK, 0 rows affected (0.00 sec)

mysql> SET @s = CONCAT("SELECT ", @c, " FROM t");
Query OK, 0 rows affected (0.00 sec)

mysql> PREPARE stmt FROM @s;
Query OK, 0 rows affected (0.04 sec)
Statement prepared

mysql> EXECUTE stmt;
+----+
| c1 |
+----+
|  0 |
+----+
|  1 |
```

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

mysql> DEALLOCATE PREPARE stmt;
Query OK, 0 rows affected (0.00 sec)
```

See Section 13.5, “Prepared Statements”, for more information.

A similar technique can be used in application programs to construct SQL statements using program variables, as shown here using PHP 5:

```
<?php
  $mysqli = new mysqli("localhost", "user", "pass", "test");

  if( mysqli_connect_errno() )
    die("Connection failed: %s\n", mysqli_connect_error());

  $col = "c1";

  $query = "SELECT $col FROM t";

  $result = $mysqli->query($query);

  while($row = $result->fetch_assoc())
  {
    echo "<p>" . $row["$col"] . "</p>\n";
  }

  $result->close();

  $mysqli->close();
?>
```

Assembling an SQL statement in this fashion is sometimes known as “Dynamic SQL”.

9.5 Expressions

This section lists the grammar rules that expressions must follow in MySQL and provides additional information about the types of terms that may appear in expressions.

- Expression Syntax
- Expression Term Notes
- Temporal Intervals

Expression Syntax

The following grammar rules define expression syntax in MySQL. The grammar shown here is based on that given in the `sql/sql_yacc.yy` file of MySQL source distributions. For additional information about some of the expression terms, see Expression Term Notes.

```
expr:
    expr OR expr
  | expr || expr
  | expr XOR expr
  | expr AND expr
  | expr && expr
  | NOT expr
  | ! expr
  | boolean_primary IS [NOT] {TRUE | FALSE | UNKNOWN}
  | boolean_primary

boolean_primary:
    boolean_primary IS [NOT] NULL
  | boolean_primary <=> predicate
  | boolean_primary comparison_operator predicate
```

```
  | boolean_primary comparison_operator {ALL | ANY} (subquery)
  | predicate

comparison_operator: = | >= | > | <= | < | <> | !=

predicate:
    bit_expr [NOT] IN (subquery)
  | bit_expr [NOT] IN (expr [, expr] ...)
  | bit_expr [NOT] BETWEEN bit_expr AND predicate
  | bit_expr SOUNDS LIKE bit_expr
  | bit_expr [NOT] LIKE simple_expr [ESCAPE simple_expr]
  | bit_expr [NOT] REGEXP bit_expr
  | bit_expr

bit_expr:
    bit_expr | bit_expr
  | bit_expr & bit_expr
  | bit_expr << bit_expr
  | bit_expr >> bit_expr
  | bit_expr + bit_expr
  | bit_expr - bit_expr
  | bit_expr * bit_expr
  | bit_expr / bit_expr
  | bit_expr DIV bit_expr
  | bit_expr MOD bit_expr
  | bit_expr % bit_expr
  | bit_expr ^ bit_expr
  | bit_expr + interval_expr
  | bit_expr - interval_expr
  | simple_expr

simple_expr:
    literal
  | identifier
  | function_call
  | simple_expr COLLATE collation_name
  | param_marker
  | variable
  | simple_expr || simple_expr
  | + simple_expr
  | - simple_expr
  | ~ simple_expr
  | ! simple_expr
  | BINARY simple_expr
  | (expr [, expr] ...)
  | ROW (expr, expr [, expr] ...)
  | (subquery)
  | EXISTS (subquery)
  | {identifier expr}
  | match_expr
  | case_expr
  | interval_expr
```

For operator precedence, see Section 12.4.1, “Operator Precedence”. The precedence and meaning of some operators depends on the SQL mode:

- By default, `||` is a logical `OR` operator. With `PIPES_AS_CONCAT` enabled, `||` is string concatenation, with a precedence between `^` and the unary operators.
- By default, `!` has a higher precedence than `NOT`. With `HIGH_NOT_PRECEDENCE` enabled, `!` and `NOT` have the same precedence.

See Section 5.1.11, “Server SQL Modes”.

Expression Term Notes

For literal value syntax, see Section 9.1, “Literal Values”.

For identifier syntax, see Section 9.2, “Schema Object Names”.

Variables can be user variables, system variables, or stored program local variables or parameters:

- User variables: Section 9.4, “User-Defined Variables”
- System variables: Section 5.1.9, “Using System Variables”
- Stored program local variables: Section 13.6.4.1, “Local Variable DECLARE Statement”
- Stored program parameters: Section 13.1.17, “CREATE PROCEDURE and CREATE FUNCTION Statements”

`param_marker` is `?` as used in prepared statements for placeholders. See Section 13.5.1, “PREPARE Statement”.

`(`*`subquery`*`)` indicates a subquery that returns a single value; that is, a scalar subquery. See Section 13.2.15.1, “The Subquery as Scalar Operand”.

`{`*`identifier expr`*`}` is ODBC escape syntax and is accepted for ODBC compatibility. The value is *`expr`*. The `{` and `}` curly braces in the syntax should be written literally; they are not metasyntax as used elsewhere in syntax descriptions.

`match_expr` indicates a `MATCH` expression. See Section 12.10, “Full-Text Search Functions”.

`case_expr` indicates a `CASE` expression. See Section 12.5, “Flow Control Functions”.

`interval_expr` represents a temporal interval. See Temporal Intervals.

Temporal Intervals

`interval_expr` in expressions represents a temporal interval. Intervals have this syntax:

```
INTERVAL expr unit
```

`expr` represents a quantity. *`unit`* represents the unit for interpreting the quantity; it is a specifier such as `HOUR`, `DAY`, or `WEEK`. The `INTERVAL` keyword and the *`unit`* specifier are not case-sensitive.

The following table shows the expected form of the *`expr`* argument for each *`unit`* value.

Table 9.2 Temporal Interval Expression and Unit Arguments

***`unit`* Value**	**Expected *`expr`* Format**
`MICROSECOND`	`MICROSECONDS`
`SECOND`	`SECONDS`
`MINUTE`	`MINUTES`
`HOUR`	`HOURS`
`DAY`	`DAYS`
`WEEK`	`WEEKS`
`MONTH`	`MONTHS`
`QUARTER`	`QUARTERS`
`YEAR`	`YEARS`
`SECOND_MICROSECOND`	`'SECONDS.MICROSECONDS'`
`MINUTE_MICROSECOND`	`'MINUTES:SECONDS.MICROSECONDS'`
`MINUTE_SECOND`	`'MINUTES:SECONDS'`
`HOUR_MICROSECOND`	`'HOURS:MINUTES:SECONDS.MICROSECONDS'`
`HOUR_SECOND`	`'HOURS:MINUTES:SECONDS'`

unit Value	Expected *expr* Format
HOUR_MINUTE	'HOURS:MINUTES'
DAY_MICROSECOND	'DAYS HOURS:MINUTES:SECONDS.MICROSECONDS'
DAY_SECOND	'DAYS HOURS:MINUTES:SECONDS'
DAY_MINUTE	'DAYS HOURS:MINUTES'
DAY_HOUR	'DAYS HOURS'
YEAR_MONTH	'YEARS-MONTHS'

MySQL permits any punctuation delimiter in the *expr* format. Those shown in the table are the suggested delimiters.

Temporal intervals are used for certain functions, such as DATE_ADD() and DATE_SUB():

```
mysql> SELECT DATE_ADD('2018-05-01',INTERVAL 1 DAY);
        -> '2018-05-02'
mysql> SELECT DATE_SUB('2018-05-01',INTERVAL 1 YEAR);
        -> '2017-05-01'
mysql> SELECT DATE_ADD('2020-12-31 23:59:59',
    ->                 INTERVAL 1 SECOND);
        -> '2021-01-01 00:00:00'
mysql> SELECT DATE_ADD('2018-12-31 23:59:59',
    ->                 INTERVAL 1 DAY);
        -> '2019-01-01 23:59:59'
mysql> SELECT DATE_ADD('2100-12-31 23:59:59',
    ->                 INTERVAL '1:1' MINUTE_SECOND);
        -> '2101-01-01 00:01:00'
mysql> SELECT DATE_SUB('2025-01-01 00:00:00',
    ->                 INTERVAL '1 1:1:1' DAY_SECOND);
        -> '2024-12-30 22:58:59'
mysql> SELECT DATE_ADD('1900-01-01 00:00:00',
    ->                 INTERVAL '-1 10' DAY_HOUR);
        -> '1899-12-30 14:00:00'
mysql> SELECT DATE_SUB('1998-01-02', INTERVAL 31 DAY);
        -> '1997-12-02'
mysql> SELECT DATE_ADD('1992-12-31 23:59:59.000002',
    ->            INTERVAL '1.999999' SECOND_MICROSECOND);
        -> '1993-01-01 00:00:01.000001'
```

Temporal arithmetic also can be performed in expressions using INTERVAL together with the + or - operator:

```
date + INTERVAL expr unit
date - INTERVAL expr unit
```

INTERVAL *expr unit* is permitted on either side of the + operator if the expression on the other side is a date or datetime value. For the - operator, INTERVAL *expr unit* is permitted only on the right side, because it makes no sense to subtract a date or datetime value from an interval.

```
mysql> SELECT '2018-12-31 23:59:59' + INTERVAL 1 SECOND;
        -> '2019-01-01 00:00:00'
mysql> SELECT INTERVAL 1 DAY + '2018-12-31';
        -> '2019-01-01'
mysql> SELECT '2025-01-01' - INTERVAL 1 SECOND;
        -> '2024-12-31 23:59:59'
```

The EXTRACT() function uses the same kinds of *unit* specifiers as DATE_ADD() or DATE_SUB(), but extracts parts from the date rather than performing date arithmetic:

```
mysql> SELECT EXTRACT(YEAR FROM '2019-07-02');
        -> 2019
mysql> SELECT EXTRACT(YEAR_MONTH FROM '2019-07-02 01:02:03');
        -> 201907
```

Temporal intervals can be used in CREATE EVENT statements:

```
CREATE EVENT myevent
    ON SCHEDULE AT CURRENT_TIMESTAMP + INTERVAL 1 HOUR
    DO
      UPDATE myschema.mytable SET mycol = mycol + 1;
```

If you specify an interval value that is too short (does not include all the interval parts that would be expected from the *`unit`* keyword), MySQL assumes that you have left out the leftmost parts of the interval value. For example, if you specify a *`unit`* of `DAY_SECOND`, the value of *`expr`* is expected to have days, hours, minutes, and seconds parts. If you specify a value like `'1:10'`, MySQL assumes that the days and hours parts are missing and the value represents minutes and seconds. In other words, `'1:10' DAY_SECOND` is interpreted in such a way that it is equivalent to `'1:10' MINUTE_SECOND`. This is analogous to the way that MySQL interprets `TIME` values as representing elapsed time rather than as a time of day.

`expr` is treated as a string, so be careful if you specify a nonstring value with `INTERVAL`. For example, with an interval specifier of `HOUR_MINUTE`, '6/4' is treated as 6 hours, four minutes, whereas `6/4` evaluates to `1.5000` and is treated as 1 hour, 5000 minutes:

```
mysql> SELECT '6/4', 6/4;
        -> 1.5000
mysql> SELECT DATE_ADD('2019-01-01', INTERVAL '6/4' HOUR_MINUTE);
        -> '2019-01-01 06:04:00'
mysql> SELECT DATE_ADD('2019-01-01', INTERVAL 6/4 HOUR_MINUTE);
        -> '2019-01-04 12:20:00'
```

To ensure interpretation of the interval value as you expect, a `CAST()` operation may be used. To treat `6/4` as 1 hour, 5 minutes, cast it to a `DECIMAL` value with a single fractional digit:

```
mysql> SELECT CAST(6/4 AS DECIMAL(3,1));
        -> 1.5
mysql> SELECT DATE_ADD('1970-01-01 12:00:00',
    ->                 INTERVAL CAST(6/4 AS DECIMAL(3,1)) HOUR_MINUTE);
        -> '1970-01-01 13:05:00'
```

If you add to or subtract from a date value something that contains a time part, the result is automatically converted to a datetime value:

```
mysql> SELECT DATE_ADD('2023-01-01', INTERVAL 1 DAY);
        -> '2023-01-02'
mysql> SELECT DATE_ADD('2023-01-01', INTERVAL 1 HOUR);
        -> '2023-01-01 01:00:00'
```

If you add `MONTH`, `YEAR_MONTH`, or `YEAR` and the resulting date has a day that is larger than the maximum day for the new month, the day is adjusted to the maximum days in the new month:

```
mysql> SELECT DATE_ADD('2019-01-30', INTERVAL 1 MONTH);
        -> '2019-02-28'
```

Date arithmetic operations require complete dates and do not work with incomplete dates such as `'2016-07-00'` or badly malformed dates:

```
mysql> SELECT DATE_ADD('2016-07-00', INTERVAL 1 DAY);
        -> NULL
mysql> SELECT '2005-03-32' + INTERVAL 1 MONTH;
        -> NULL
```

9.6 Query Attributes

The most visible part of an SQL statement is the text of the statement. As of MySQL 8.0.23, clients can also define query attributes that apply to the next statement sent to the server for execution:

- Attributes are defined prior to sending the statement.
- Attributes exist until statement execution ends, at which point the attribute set is cleared.

- While attributes exist, they can be accessed on the server side.

Examples of the ways query attributes may be used:

- A web application produces pages that generate database queries, and for each query must track the URL of the page that generated it.
- An application passes extra processing information with each query, for use by a plugin such as an audit plugin or query rewrite plugin.

MySQL supports these capabilities without the use of workarounds such as specially formatted comments included in query strings. The remainder of this section describes how to use query attribute support, including the prerequisites that must be satisfied.

Defining and Accessing Query Attributes

Applications that use the MySQL C API define query attributes by calling the `mysql_bind_param()` function. See mysql_bind_param(). Other MySQL connectors may also provide query-attribute support. See the documentation for individual connectors.

The `mysql` client has a `query_attributes` command that enables defining up to 32 pairs of attribute names and values. See Section 4.5.1.2, “mysql Client Commands”.

Query attribute names are transmitted using the character set indicated by the `character_set_client` system variable.

To access query attributes within SQL statements for which attributes have been defined, install the `query_attributes` component as described in Prerequisites for Using Query Attributes. The component implements a `mysql_query_attribute_string()` loadable function that takes an attribute name argument and returns the attribute value as a string, or `NULL` if the attribute does not exist. See Query Attribute Loadable Functions.

The following examples use the `mysql` client `query_attributes` command to define attribute name/value pairs, and the `mysql_query_attribute_string()` function to access attribute values by name.

This example defines two attributes named `n1` and `n2`. The first `SELECT` shows how to retrieve those attributes, and also demonstrates that retrieving a nonexistent attribute (`n3`) returns `NULL`. The second `SELECT` shows that attributes do not persist across statements.

```
mysql> query_attributes n1 v1 n2 v2;
mysql> SELECT
         mysql_query_attribute_string('n1') AS 'attr 1',
         mysql_query_attribute_string('n2') AS 'attr 2',
         mysql_query_attribute_string('n3') AS 'attr 3';
+--------+--------+--------+
| attr 1 | attr 2 | attr 3 |
+--------+--------+--------+
| v1     | v2     | NULL   |
+--------+--------+--------+

mysql> SELECT
         mysql_query_attribute_string('n1') AS 'attr 1',
         mysql_query_attribute_string('n2') AS 'attr 2';
+--------+--------+
| attr 1 | attr 2 |
+--------+--------+
| NULL   | NULL   |
```

```
+--------+--------+
```

As shown by the second `SELECT` statement, attributes defined prior to a given statement are available only to that statement and are cleared after the statement executes. To use an attribute value across multiple statements, assign it to a variable. The following example shows how to do this, and illustrates that attribute values are available in subsequent statements by means of the variables, but not by calling `mysql_query_attribute_string()`:

```
mysql> query_attributes n1 v1 n2 v2;
mysql> SET
         @attr1 = mysql_query_attribute_string('n1'),
         @attr2 = mysql_query_attribute_string('n2');

mysql> SELECT
         @attr1, mysql_query_attribute_string('n1') AS 'attr 1',
         @attr2, mysql_query_attribute_string('n2') AS 'attr 2';
+--------+--------+--------+--------+
| @attr1 | attr 1 | @attr2 | attr 2 |
+--------+--------+--------+--------+
| v1     | NULL   | v2     | NULL   |
+--------+--------+--------+--------+
```

Attributes can also be saved for later use by storing them in a table:

```
mysql> CREATE TABLE t1 (c1 CHAR(20), c2 CHAR(20));

mysql> query_attributes n1 v1 n2 v2;
mysql> INSERT INTO t1 (c1, c2) VALUES(
         mysql_query_attribute_string('n1'),
         mysql_query_attribute_string('n2')
       );

mysql> SELECT * FROM t1;
+------+------+
| c1   | c2   |
+------+------+
| v1   | v2   |
+------+------+
```

Query attributes are subject to these limitations and restrictions:

- If multiple attribute-definition operations occur prior to sending a statement to the server for execution, the most recent definition operation applies and replaces attributes defined in earlier operations.
- If multiple attributes are defined with the same name, attempts to retrieve the attribute value have an undefined result.
- An attribute defined with an empty name cannot be retrieved by name.
- Attributes are not available to statements prepared with `PREPARE`.
- The `mysql_query_attribute_string()` function cannot be used in DDL statements.
- Attributes are not replicated. Statements that invoke the `mysql_query_attribute_string()` function will not get the same value on all servers.

Prerequisites for Using Query Attributes

To access query attributes within SQL statements for which attributes have been defined, the `query_attributes` component must be installed. Do so using this statement:

```
INSTALL COMPONENT "file://component_query_attributes";
```

Component installation is a one-time operation that need not be done per server startup. `INSTALL COMPONENT` loads the component, and also registers it in the `mysql.component` system table to cause it to be loaded during subsequent server startups.

The `query_attributes` component accesses query attributes to implement a `mysql_query_attribute_string()` function. See Section 5.5.4, “Query Attribute Components”.

To uninstall the `query_attributes` component, use this statement:

```
UNINSTALL COMPONENT "file://component_query_attributes";
```

`UNINSTALL COMPONENT` unloads the component, and unregisters it from the `mysql.component` system table to cause it not to be loaded during subsequent server startups.

Because installing and uninstalling the `query_attributes` component installs and uninstalls the `mysql_query_attribute_string()` function that the component implements, it is not necessary to use `CREATE FUNCTION` or `DROP FUNCTION` to do so.

Query Attribute Loadable Functions

- `mysql_query_attribute_string(`*`name`*`)`

  Applications can define attributes that apply to the next query sent to the server. The `mysql_query_attribute_string()` function, available as of MySQL 8.0.23, returns an attribute value as a string, given the attribute name. This function enables a query to access and incorporate values of the attributes that apply to it.

  `mysql_query_attribute_string()` is installed by installing the `query_attributes` component. See Section 9.6, “Query Attributes”, which also discusses the purpose and use of query attributes.

  Arguments:

  - *`name`*: The attribute name.

  Return value:

  Returns the attribute value as a string for success, or `NULL` if the attribute does not exist.

  Example:

  The following example uses the `mysql` client `query_attributes` command to define query attributes that can be retrieved by `mysql_query_attribute_string()`. The `SELECT` shows that retrieving a nonexistent attribute (`n3`) returns `NULL`.

```
mysql> query_attributes n1 v1 n2 v2;
mysql> SELECT
    ->   mysql_query_attribute_string('n1') AS 'attr 1',
    ->   mysql_query_attribute_string('n2') AS 'attr 2',
    ->   mysql_query_attribute_string('n3') AS 'attr 3';
+--------+--------+--------+
| attr 1 | attr 2 | attr 3 |
+--------+--------+--------+
| v1     | v2     | NULL   |
+--------+--------+--------+
```

9.7 Comments

MySQL Server supports three comment styles:

- From a `#` character to the end of the line.
- From a `--` sequence to the end of the line. In MySQL, the `--` (double-dash) comment style requires the second dash to be followed by at least one whitespace or control character (such as a space, tab, newline, and so on). This syntax differs slightly from standard SQL comment syntax, as discussed in Section 1.6.2.4, “'--' as the Start of a Comment”.

- From a `/*` sequence to the following `*/` sequence, as in the C programming language. This syntax enables a comment to extend over multiple lines because the beginning and closing sequences need not be on the same line.

The following example demonstrates all three comment styles:

```
mysql> SELECT 1+1;     # This comment continues to the end of line
mysql> SELECT 1+1;     -- This comment continues to the end of line
mysql> SELECT 1 /* this is an in-line comment */ + 1;
mysql> SELECT 1+
/*
this is a
multiple-line comment
*/
1;
```

Nested comments are not supported, and are deprecated; expect them to be removed in a future MySQL release. (Under some conditions, nested comments might be permitted, but usually are not, and users should avoid them.)

MySQL Server supports certain variants of C-style comments. These enable you to write code that includes MySQL extensions, but is still portable, by using comments of the following form:

```
/*! MySQL-specific code */
```

In this case, MySQL Server parses and executes the code within the comment as it would any other SQL statement, but other SQL servers should ignore the extensions. For example, MySQL Server recognizes the `STRAIGHT_JOIN` keyword in the following statement, but other servers should not:

```
SELECT /*! STRAIGHT_JOIN */ col1 FROM table1,table2 WHERE ...
```

If you add a version number after the `!` character, the syntax within the comment is executed only if the MySQL version is greater than or equal to the specified version number. The `KEY_BLOCK_SIZE` keyword in the following comment is executed only by servers from MySQL 5.1.10 or higher:

```
CREATE TABLE t1(a INT, KEY (a)) /*!50110 KEY_BLOCK_SIZE=1024 */;
```

The version number uses the format *`Mmmrr`*, where *`M`* is a major version, *`mm`* is a two-digit minor version, and *`rr`* is a two-digit release number. For example: In a statement to be run only by a MySQL server version 8.0.31 or later, use `80031` in the comment.

The comment syntax just described applies to how the `mysqld` server parses SQL statements. The `mysql` client program also performs some parsing of statements before sending them to the server. (It does this to determine statement boundaries within a multiple-statement input line.) For information about differences between the server and `mysql` client parsers, see Section 4.5.1.6, “mysql Client Tips”.

Comments in `/*!12345 ... */` format are not stored on the server. If this format is used to comment stored programs, the comments are not retained in the program body.

Another variant of C-style comment syntax is used to specify optimizer hints. Hint comments include a `+` character following the `/*` comment opening sequence. Example:

```
SELECT /*+ BKA(t1) */ FROM ... ;
```

For more information, see Section 8.9.3, “Optimizer Hints”.

The use of short-form `mysql` commands such as `\C` within multiple-line `/* ... */` comments is not supported. Short-form commands do work within single-line `/*! ... */` version comments, as do `/*+ ... */` optimizer-hint comments, which are stored in object definitions. If there is a concern that optimizer-hint comments may be stored in object definitions so that dump files when reloaded with `mysql` would result in execution of such commands, either invoke `mysql` with the `--binary-mode` option or use a reload client other than `mysql`.