

---

# Chapter 8 Optimization

## Table of Contents

8.1 Optimization Overview .....	1378
8.2 Optimizing SQL Statements .....	1380
8.2.1 Optimizing SELECT Statements .....	1380
8.2.2 Optimizing Subqueries, Derived Tables, View References, and Common Table Expressions .....	1427
8.2.3 Optimizing INFORMATION_SCHEMA Queries .....	1439
8.2.4 Optimizing Performance Schema Queries .....	1442
8.2.5 Optimizing Data Change Statements .....	1444
8.2.6 Optimizing Database Privileges .....	1446
8.2.7 Other Optimization Tips .....	1446
8.3 Optimization and Indexes .....	1446
8.3.1 How MySQL Uses Indexes .....	1447
8.3.2 Primary Key Optimization .....	1448
8.3.3 SPATIAL Index Optimization .....	1448
8.3.4 Foreign Key Optimization .....	1449
8.3.5 Column Indexes .....	1449
8.3.6 Multiple-Column Indexes .....	1450
8.3.7 Verifying Index Usage .....	1452
8.3.8 InnoDB and MyISAM Index Statistics Collection .....	1452
8.3.9 Comparison of B-Tree and Hash Indexes .....	1454
8.3.10 Use of Index Extensions .....	1455
8.3.11 Optimizer Use of Generated Column Indexes .....	1458
8.3.12 Invisible Indexes .....	1459
8.3.13 Descending Indexes .....	1461
8.3.14 Indexed Lookups from TIMESTAMP Columns .....	1462
8.4 Optimizing Database Structure .....	1464
8.4.1 Optimizing Data Size .....	1464
8.4.2 Optimizing MySQL Data Types .....	1466
8.4.3 Optimizing for Many Tables .....	1467
8.4.4 Internal Temporary Table Use in MySQL .....	1469
8.5 Optimizing for InnoDB Tables .....	1472
8.5.1 Optimizing Storage Layout for InnoDB Tables .....	1472
8.5.2 Optimizing InnoDB Transaction Management .....	1473
8.5.3 Optimizing InnoDB Read-Only Transactions .....	1474
8.5.4 Optimizing InnoDB Redo Logging .....	1475
8.5.5 Bulk Data Loading for InnoDB Tables .....	1476
8.5.6 Optimizing InnoDB Queries .....	1477
8.5.7 Optimizing InnoDB DDL Operations .....	1478
8.5.8 Optimizing InnoDB Disk I/O .....	1478
8.5.9 Optimizing InnoDB Configuration Variables .....	1481
8.5.10 Optimizing InnoDB for Systems with Many Tables .....	1483
8.6 Optimizing for MyISAM Tables .....	1483
8.6.1 Optimizing MyISAM Queries .....	1483
8.6.2 Bulk Data Loading for MyISAM Tables .....	1484
8.6.3 Optimizing REPAIR TABLE Statements .....	1485
8.7 Optimizing for MEMORY Tables .....	1487
8.8 Understanding the Query Execution Plan .....	1487
8.8.1 Optimizing Queries with EXPLAIN .....	1487

8.8.2 EXPLAIN Output Format .....	1488
8.8.3 Extended EXPLAIN Output Format .....	1502
8.8.4 Obtaining Execution Plan Information for a Named Connection .....	1504
8.8.5 Estimating Query Performance .....	1505
8.9 Controlling the Query Optimizer .....	1505
8.9.1 Controlling Query Plan Evaluation .....	1505
8.9.2 Switchable Optimizations .....	1506
8.9.3 Optimizer Hints .....	1510
8.9.4 Index Hints .....	1524
8.9.5 The Optimizer Cost Model .....	1527
8.9.6 Optimizer Statistics .....	1530
8.10 Buffering and Caching .....	1533
8.10.1 InnoDB Buffer Pool Optimization .....	1533
8.10.2 The MyISAM Key Cache .....	1534
8.10.3 Caching of Prepared Statements and Stored Programs .....	1538
8.11 Optimizing Locking Operations .....	1540
8.11.1 Internal Locking Methods .....	1540
8.11.2 Table Locking Issues .....	1542
8.11.3 Concurrent Inserts .....	1544
8.11.4 Metadata Locking .....	1544
8.11.5 External Locking .....	1548
8.12 Optimizing the MySQL Server .....	1549
8.12.1 Optimizing Disk I/O .....	1549
8.12.2 Using Symbolic Links .....	1551
8.12.3 Optimizing Memory Use .....	1553
8.12.4 Optimizing Network Use .....	1559
8.12.5 Resource Groups .....	1564
8.13 Measuring Performance (Benchmarking) .....	1569
8.13.1 Measuring the Speed of Expressions and Functions .....	1570
8.13.2 Using Your Own Benchmarks .....	1570
8.13.3 Measuring Performance with <code>performance_schema</code> .....	1570
8.14 Examining Thread Information .....	1571
8.14.1 Thread Command Values .....	1572
8.14.2 General Thread States .....	1574
8.14.3 Replication Master Thread States .....	1580
8.14.4 Replication Slave I/O Thread States .....	1581
8.14.5 Replication Slave SQL Thread States .....	1582
8.14.6 Replication Slave Connection Thread States .....	1583
8.14.7 NDB Cluster Thread States .....	1583
8.14.8 Event Scheduler Thread States .....	1584

This chapter explains how to optimize MySQL performance and provides examples. Optimization involves configuring, tuning, and measuring performance, at several levels. Depending on your job role (developer, DBA, or a combination of both), you might optimize at the level of individual SQL statements, entire applications, a single database server, or multiple networked database servers. Sometimes you can be proactive and plan in advance for performance, while other times you might troubleshoot a configuration or code issue after a problem occurs. Optimizing CPU and memory usage can also improve scalability, allowing the database to handle more load without slowing down.

## 8.1 Optimization Overview

Database performance depends on several factors at the database level, such as tables, queries, and configuration settings. These software constructs result in CPU and I/O operations at the hardware level,

which you must minimize and make as efficient as possible. As you work on database performance, you start by learning the high-level rules and guidelines for the software side, and measuring performance using wall-clock time. As you become an expert, you learn more about what happens internally, and start measuring things such as CPU cycles and I/O operations.

Typical users aim to get the best database performance out of their existing software and hardware configurations. Advanced users look for opportunities to improve the MySQL software itself, or develop their own storage engines and hardware appliances to expand the MySQL ecosystem.

- [Optimizing at the Database Level](#)
- [Optimizing at the Hardware Level](#)
- [Balancing Portability and Performance](#)

## Optimizing at the Database Level

The most important factor in making a database application fast is its basic design:

- Are the tables structured properly? In particular, do the columns have the right data types, and does each table have the appropriate columns for the type of work? For example, applications that perform frequent updates often have many tables with few columns, while applications that analyze large amounts of data often have few tables with many columns.
- Are the right [indexes](#) in place to make queries efficient?
- Are you using the appropriate storage engine for each table, and taking advantage of the strengths and features of each storage engine you use? In particular, the choice of a transactional storage engine such as [InnoDB](#) or a nontransactional one such as [MyISAM](#) can be very important for performance and scalability.



### Note

[InnoDB](#) is the default storage engine for new tables. In practice, the advanced [InnoDB](#) performance features mean that [InnoDB](#) tables often outperform the simpler [MyISAM](#) tables, especially for a busy database.

- Does each table use an appropriate row format? This choice also depends on the storage engine used for the table. In particular, compressed tables use less disk space and so require less disk I/O to read and write the data. Compression is available for all kinds of workloads with [InnoDB](#) tables, and for read-only [MyISAM](#) tables.
- Does the application use an appropriate [locking strategy](#)? For example, by allowing shared access when possible so that database operations can run concurrently, and requesting exclusive access when appropriate so that critical operations get top priority. Again, the choice of storage engine is significant. The [InnoDB](#) storage engine handles most locking issues without involvement from you, allowing for better concurrency in the database and reducing the amount of experimentation and tuning for your code.
- Are all [memory areas used for caching](#) sized correctly? That is, large enough to hold frequently accessed data, but not so large that they overload physical memory and cause paging. The main memory areas to configure are the [InnoDB](#) buffer pool and the [MyISAM](#) key cache.

## Optimizing at the Hardware Level

Any database application eventually hits hardware limits as the database becomes more and more busy. A DBA must evaluate whether it is possible to tune the application or reconfigure the server to avoid these

bottlenecks, or whether more hardware resources are required. System bottlenecks typically arise from these sources:

- Disk seeks. It takes time for the disk to find a piece of data. With modern disks, the mean time for this is usually lower than 10ms, so we can in theory do about 100 seeks a second. This time improves slowly with new disks and is very hard to optimize for a single table. The way to optimize seek time is to distribute the data onto more than one disk.
- Disk reading and writing. When the disk is at the correct position, we need to read or write the data. With modern disks, one disk delivers at least 10–20MB/s throughput. This is easier to optimize than seeks because you can read in parallel from multiple disks.
- CPU cycles. When the data is in main memory, we must process it to get our result. Having large tables compared to the amount of memory is the most common limiting factor. But with small tables, speed is usually not the problem.
- Memory bandwidth. When the CPU needs more data than can fit in the CPU cache, main memory bandwidth becomes a bottleneck. This is an uncommon bottleneck for most systems, but one to be aware of.

## Balancing Portability and Performance

To use performance-oriented SQL extensions in a portable MySQL program, you can wrap MySQL-specific keywords in a statement within `/*! */` comment delimiters. Other SQL servers ignore the commented keywords. For information about writing comments, see [Section 9.6, “Comment Syntax”](#).

## 8.2 Optimizing SQL Statements

The core logic of a database application is performed through SQL statements, whether issued directly through an interpreter or submitted behind the scenes through an API. The tuning guidelines in this section help to speed up all kinds of MySQL applications. The guidelines cover SQL operations that read and write data, the behind-the-scenes overhead for SQL operations in general, and operations used in specific scenarios such as database monitoring.

### 8.2.1 Optimizing SELECT Statements

Queries, in the form of `SELECT` statements, perform all the lookup operations in the database. Tuning these statements is a top priority, whether to achieve sub-second response times for dynamic web pages, or to chop hours off the time to generate huge overnight reports.

Besides `SELECT` statements, the tuning techniques for queries also apply to constructs such as `CREATE TABLE...AS SELECT`, `INSERT INTO...SELECT`, and `WHERE` clauses in `DELETE` statements. Those statements have additional performance considerations because they combine write operations with the read-oriented query operations.

NDB Cluster supports a join pushdown optimization whereby a qualifying join is sent in its entirety to NDB Cluster data nodes, where it can be distributed among them and executed in parallel. For more information about this optimization, see [Conditions for NDB pushdown joins](#),

The main considerations for optimizing queries are:

- To make a slow `SELECT ... WHERE` query faster, the first thing to check is whether you can add an `index`. Set up indexes on columns used in the `WHERE` clause, to speed up evaluation, filtering, and the final retrieval of results. To avoid wasted disk space, construct a small set of indexes that speed up many related queries used in your application.

Indexes are especially important for queries that reference different tables, using features such as [joins](#) and [foreign keys](#). You can use the [EXPLAIN](#) statement to determine which indexes are used for a [SELECT](#). See [Section 8.3.1, “How MySQL Uses Indexes”](#) and [Section 8.8.1, “Optimizing Queries with EXPLAIN”](#).

- Isolate and tune any part of the query, such as a function call, that takes excessive time. Depending on how the query is structured, a function could be called once for every row in the result set, or even once for every row in the table, greatly magnifying any inefficiency.
- Minimize the number of [full table scans](#) in your queries, particularly for big tables.
- Keep table statistics up to date by using the [ANALYZE TABLE](#) statement periodically, so the optimizer has the information needed to construct an efficient execution plan.
- Learn the tuning techniques, indexing techniques, and configuration parameters that are specific to the storage engine for each table. Both [InnoDB](#) and [MyISAM](#) have sets of guidelines for enabling and sustaining high performance in queries. For details, see [Section 8.5.6, “Optimizing InnoDB Queries”](#) and [Section 8.6.1, “Optimizing MyISAM Queries”](#).
- You can optimize single-query transactions for [InnoDB](#) tables, using the technique in [Section 8.5.3, “Optimizing InnoDB Read-Only Transactions”](#).
- Avoid transforming the query in ways that make it hard to understand, especially if the optimizer does some of the same transformations automatically.
- If a performance issue is not easily solved by one of the basic guidelines, investigate the internal details of the specific query by reading the [EXPLAIN](#) plan and adjusting your indexes, [WHERE](#) clauses, join clauses, and so on. (When you reach a certain level of expertise, reading the [EXPLAIN](#) plan might be your first step for every query.)
- Adjust the size and properties of the memory areas that MySQL uses for caching. With efficient use of the [InnoDB buffer pool](#), [MyISAM](#) key cache, and the MySQL query cache, repeated queries run faster because the results are retrieved from memory the second and subsequent times.
- Even for a query that runs fast using the cache memory areas, you might still optimize further so that they require less cache memory, making your application more scalable. Scalability means that your application can handle more simultaneous users, larger requests, and so on without experiencing a big drop in performance.
- Deal with locking issues, where the speed of your query might be affected by other sessions accessing the tables at the same time.

### 8.2.1.1 WHERE Clause Optimization

This section discusses optimizations that can be made for processing [WHERE](#) clauses. The examples use [SELECT](#) statements, but the same optimizations apply for [WHERE](#) clauses in [DELETE](#) and [UPDATE](#) statements.



#### Note

Because work on the MySQL optimizer is ongoing, not all of the optimizations that MySQL performs are documented here.

You might be tempted to rewrite your queries to make arithmetic operations faster, while sacrificing readability. Because MySQL does similar optimizations automatically, you can often avoid this work, and leave the query in a more understandable and maintainable form. Some of the optimizations performed by MySQL follow:

- Removal of unnecessary parentheses:

```
((a AND b) AND c OR (((a AND b) AND (c AND d))))
-> (a AND b AND c) OR (a AND b AND c AND d)
```

- Constant folding:

```
(a<b AND b=c) AND a=5
-> b>5 AND b=c AND a=5
```

- Constant condition removal:

```
(b>=5 AND b=5) OR (b=6 AND 5=5) OR (b=7 AND 5=6)
-> b=5 OR b=6
```

In MySQL 8.0.14 and later, this takes place during preparation rather than during the optimization phase, which helps in simplification of joins. See [Section 8.2.1.8, “Outer Join Optimization”](#), for further information and examples.

- Constant expressions used by indexes are evaluated only once.
- `COUNT(*)` on a single table without a `WHERE` is retrieved directly from the table information for `MyISAM` and `MEMORY` tables. This is also done for any `NOT NULL` expression when used with only one table.
- Early detection of invalid constant expressions. MySQL quickly detects that some `SELECT` statements are impossible and returns no rows.
- `HAVING` is merged with `WHERE` if you do not use `GROUP BY` or aggregate functions (`COUNT()`, `MIN()`, and so on).
- For each table in a join, a simpler `WHERE` is constructed to get a fast `WHERE` evaluation for the table and also to skip rows as soon as possible.
- All constant tables are read first before any other tables in the query. A constant table is any of the following:
  - An empty table or a table with one row.
  - A table that is used with a `WHERE` clause on a `PRIMARY KEY` or a `UNIQUE` index, where all index parts are compared to constant expressions and are defined as `NOT NULL`.

All of the following tables are used as constant tables:

```
SELECT * FROM t WHERE primary_key=1;
SELECT * FROM t1,t2
  WHERE t1.primary_key=1 AND t2.primary_key=t1.id;
```

- The best join combination for joining the tables is found by trying all possibilities. If all columns in `ORDER BY` and `GROUP BY` clauses come from the same table, that table is preferred first when joining.
- If there is an `ORDER BY` clause and a different `GROUP BY` clause, or if the `ORDER BY` or `GROUP BY` contains columns from tables other than the first table in the join queue, a temporary table is created.
- If you use the `SQL_SMALL_RESULT` modifier, MySQL uses an in-memory temporary table.
- Each table index is queried, and the best index is used unless the optimizer believes that it is more efficient to use a table scan. At one time, a scan was used based on whether the best index spanned

more than 30% of the table, but a fixed percentage no longer determines the choice between using an index or a scan. The optimizer now is more complex and bases its estimate on additional factors such as table size, number of rows, and I/O block size.

- In some cases, MySQL can read rows from the index without even consulting the data file. If all columns used from the index are numeric, only the index tree is used to resolve the query.
- Before each row is output, those that do not match the `HAVING` clause are skipped.

Some examples of queries that are very fast:

```
SELECT COUNT(*) FROM tbl_name;
SELECT MIN(key_part1),MAX(key_part1) FROM tbl_name;
SELECT MAX(key_part2) FROM tbl_name
  WHERE key_part1=constant;
SELECT ... FROM tbl_name
  ORDER BY key_part1,key_part2,... LIMIT 10;
SELECT ... FROM tbl_name
  ORDER BY key_part1 DESC, key_part2 DESC, ... LIMIT 10;
```

MySQL resolves the following queries using only the index tree, assuming that the indexed columns are numeric:

```
SELECT key_part1,key_part2 FROM tbl_name WHERE key_part1=val;
SELECT COUNT(*) FROM tbl_name
  WHERE key_part1=val1 AND key_part2=val2;
SELECT key_part2 FROM tbl_name GROUP BY key_part1;
```

The following queries use indexing to retrieve the rows in sorted order without a separate sorting pass:

```
SELECT ... FROM tbl_name
  ORDER BY key_part1,key_part2,... ;
SELECT ... FROM tbl_name
  ORDER BY key_part1 DESC, key_part2 DESC, ... ;
```

### 8.2.1.2 Range Optimization

The `range` access method uses a single index to retrieve a subset of table rows that are contained within one or several index value intervals. It can be used for a single-part or multiple-part index. The following sections describe conditions under which the optimizer uses range access.

- [Range Access Method for Single-Part Indexes](#)
- [Range Access Method for Multiple-Part Indexes](#)
- [Equality Range Optimization of Many-Valued Comparisons](#)
- [Skip Scan Range Access Method](#)
- [Range Optimization of Row Constructor Expressions](#)
- [Limiting Memory Use for Range Optimization](#)

## Range Access Method for Single-Part Indexes

For a single-part index, index value intervals can be conveniently represented by corresponding conditions in the `WHERE` clause, denoted as *range conditions* rather than “intervals.”

The definition of a range condition for a single-part index is as follows:

- For both `BTREE` and `HASH` indexes, comparison of a key part with a constant value is a range condition when using the `=`, `<=`, `IN()`, `IS NULL`, or `IS NOT NULL` operators.
- Additionally, for `BTREE` indexes, comparison of a key part with a constant value is a range condition when using the `>`, `<`, `>=`, `<=`, `BETWEEN`, `!=`, or `<>` operators, or `LIKE` comparisons if the argument to `LIKE` is a constant string that does not start with a wildcard character.
- For all index types, multiple range conditions combined with `OR` or `AND` form a range condition.

“Constant value” in the preceding descriptions means one of the following:

- A constant from the query string
- A column of a `const` or `system` table from the same join
- The result of an uncorrelated subquery
- Any expression composed entirely from subexpressions of the preceding types

Here are some examples of queries with range conditions in the `WHERE` clause:

```
SELECT * FROM t1
  WHERE key_col > 1
    AND key_col < 10;

SELECT * FROM t1
  WHERE key_col = 1
    OR key_col IN (15,18,20);

SELECT * FROM t1
  WHERE key_col LIKE 'ab%'
    OR key_col BETWEEN 'bar' AND 'foo';
```

Some nonconstant values may be converted to constants during the optimizer constant propagation phase.

MySQL tries to extract range conditions from the `WHERE` clause for each of the possible indexes. During the extraction process, conditions that cannot be used for constructing the range condition are dropped, conditions that produce overlapping ranges are combined, and conditions that produce empty ranges are removed.

Consider the following statement, where `key1` is an indexed column and `nonkey` is not indexed:

```
SELECT * FROM t1 WHERE
  (key1 < 'abc' AND (key1 LIKE 'abcde%' OR key1 LIKE '%b')) OR
  (key1 < 'bar' AND nonkey = 4) OR
  (key1 < 'uux' AND key1 > 'z');
```

The extraction process for key `key1` is as follows:

1. Start with original `WHERE` clause:

```
(key1 < 'abc' AND (key1 LIKE 'abcde%' OR key1 LIKE '%b')) OR
(key1 < 'bar' AND nonkey = 4) OR
(key1 < 'uux' AND key1 > 'z')
```

2. Remove `nonkey = 4` and `key1 LIKE '%b'` because they cannot be used for a range scan. The correct way to remove them is to replace them with `TRUE`, so that we do not miss any matching rows when doing the range scan. Replacing them with `TRUE` yields:

```
(key1 < 'abc' AND (key1 LIKE 'abcde%' OR TRUE)) OR
(key1 < 'bar' AND TRUE) OR
(key1 < 'uux' AND key1 > 'z')
```

3. Collapse conditions that are always true or false:

- `(key1 LIKE 'abcde%' OR TRUE)` is always true
- `(key1 < 'uux' AND key1 > 'z')` is always false

Replacing these conditions with constants yields:

```
(key1 < 'abc' AND TRUE) OR (key1 < 'bar' AND TRUE) OR (FALSE)
```

Removing unnecessary `TRUE` and `FALSE` constants yields:

```
(key1 < 'abc') OR (key1 < 'bar')
```

4. Combining overlapping intervals into one yields the final condition to be used for the range scan:

```
(key1 < 'bar')
```

In general (and as demonstrated by the preceding example), the condition used for a range scan is less restrictive than the `WHERE` clause. MySQL performs an additional check to filter out rows that satisfy the range condition but not the full `WHERE` clause.

The range condition extraction algorithm can handle nested `AND/OR` constructs of arbitrary depth, and its output does not depend on the order in which conditions appear in `WHERE` clause.

MySQL does not support merging multiple ranges for the `range` access method for spatial indexes. To work around this limitation, you can use a `UNION` with identical `SELECT` statements, except that you put each spatial predicate in a different `SELECT`.

## Range Access Method for Multiple-Part Indexes

Range conditions on a multiple-part index are an extension of range conditions for a single-part index. A range condition on a multiple-part index restricts index rows to lie within one or several key tuple intervals. Key tuple intervals are defined over a set of key tuples, using ordering from the index.

For example, consider a multiple-part index defined as `key1(key_part1, key_part2, key_part3)`, and the following set of key tuples listed in key order:

<code>key_part1</code>	<code>key_part2</code>	<code>key_part3</code>
NULL	1	'abc'
NULL	1	'xyz'
NULL	2	'foo'
1	1	'abc'
1	1	'xyz'

1	2	'abc'
2	1	'aaa'

The condition `key_part1 = 1` defines this interval:

```
(1,-inf,-inf) <= (key_part1,key_part2,key_part3) < (1,+inf,+inf)
```

The interval covers the 4th, 5th, and 6th tuples in the preceding data set and can be used by the range access method.

By contrast, the condition `key_part3 = 'abc'` does not define a single interval and cannot be used by the range access method.

The following descriptions indicate how range conditions work for multiple-part indexes in greater detail.

- For `HASH` indexes, each interval containing identical values can be used. This means that the interval can be produced only for conditions in the following form:

```
key_part1 cmp const1
AND key_part2 cmp const2
AND ...
AND key_partN cmp constN;
```

Here, `const1`, `const2`, ... are constants, `cmp` is one of the `=`, `<=`, or `IS NULL` comparison operators, and the conditions cover all index parts. (That is, there are `N` conditions, one for each part of an `N`-part index.) For example, the following is a range condition for a three-part `HASH` index:

```
key_part1 = 1 AND key_part2 IS NULL AND key_part3 = 'foo'
```

For the definition of what is considered to be a constant, see [Range Access Method for Single-Part Indexes](#).

- For a `BTREE` index, an interval might be usable for conditions combined with `AND`, where each condition compares a key part with a constant value using `=`, `<=`, `IS NULL`, `>`, `<`, `>=`, `<=`, `!=`, `<>`, `BETWEEN`, or `LIKE 'pattern'` (where `'pattern'` does not start with a wildcard). An interval can be used as long as it is possible to determine a single key tuple containing all rows that match the condition (or two intervals if `<>` or `!=` is used).

The optimizer attempts to use additional key parts to determine the interval as long as the comparison operator is `=`, `<=`, or `IS NULL`. If the operator is `>`, `<`, `>=`, `<=`, `!=`, `<>`, `BETWEEN`, or `LIKE`, the optimizer uses it but considers no more key parts. For the following expression, the optimizer uses `=` from the first comparison. It also uses `>=` from the second comparison but considers no further key parts and does not use the third comparison for interval construction:

```
key_part1 = 'foo' AND key_part2 >= 10 AND key_part3 > 10
```

The single interval is:

```
('foo',10,-inf) < (key_part1,key_part2,key_part3) < ('foo',+inf,+inf)
```

It is possible that the created interval contains more rows than the initial condition. For example, the preceding interval includes the value `('foo', 11, 0)`, which does not satisfy the original condition.

- If conditions that cover sets of rows contained within intervals are combined with `OR`, they form a condition that covers a set of rows contained within the union of their intervals. If the conditions are

combined with `AND`, they form a condition that covers a set of rows contained within the intersection of their intervals. For example, for this condition on a two-part index:

```
(key_part1 = 1 AND key_part2 < 2) OR (key_part1 > 5)
```

The intervals are:

```
(1,-inf) < (key_part1,key_part2) < (1,2)
(5,-inf) < (key_part1,key_part2)
```

In this example, the interval on the first line uses one key part for the left bound and two key parts for the right bound. The interval on the second line uses only one key part. The `key_len` column in the `EXPLAIN` output indicates the maximum length of the key prefix used.

In some cases, `key_len` may indicate that a key part was used, but that might be not what you would expect. Suppose that `key_part1` and `key_part2` can be `NULL`. Then the `key_len` column displays two key part lengths for the following condition:

```
key_part1 >= 1 AND key_part2 < 2
```

But, in fact, the condition is converted to this:

```
key_part1 >= 1 AND key_part2 IS NOT NULL
```

For a description of how optimizations are performed to combine or eliminate intervals for range conditions on a single-part index, see [Range Access Method for Single-Part Indexes](#). Analogous steps are performed for range conditions on multiple-part indexes.

## Equality Range Optimization of Many-Valued Comparisons

Consider these expressions, where `col_name` is an indexed column:

```
col_name IN(val1, ..., valN)
col_name = val1 OR ... OR col_name = valN
```

Each expression is true if `col_name` is equal to any of several values. These comparisons are equality range comparisons (where the “range” is a single value). The optimizer estimates the cost of reading qualifying rows for equality range comparisons as follows:

- If there is a unique index on `col_name`, the row estimate for each range is 1 because at most one row can have the given value.
- Otherwise, any index on `col_name` is nonunique and the optimizer can estimate the row count for each range using dives into the index or index statistics.

With index dives, the optimizer makes a dive at each end of a range and uses the number of rows in the range as the estimate. For example, the expression `col_name IN (10, 20, 30)` has three equality ranges and the optimizer makes two dives per range to generate a row estimate. Each pair of dives yields an estimate of the number of rows that have the given value.

Index dives provide accurate row estimates, but as the number of comparison values in the expression increases, the optimizer takes longer to generate a row estimate. Use of index statistics is less accurate than index dives but permits faster row estimation for large value lists.

The `eq_range_index_dive_limit` system variable enables you to configure the number of values at which the optimizer switches from one row estimation strategy to the other. To permit use of index dives for comparisons of up to  $N$  equality ranges, set `eq_range_index_dive_limit` to  $N + 1$ . To disable use of statistics and always use index dives regardless of  $N$ , set `eq_range_index_dive_limit` to 0.

To update table index statistics for best estimates, use `ANALYZE TABLE`.

Prior to MySQL 8.0, there is no way of skipping the use of index dives to estimate index usefulness, except by using the `eq_range_index_dive_limit` system variable. In MySQL 8.0, index dive skipping is possible for queries that satisfy all these conditions:

- The query is for a single table, not a join on multiple tables.
- A single-index `FORCE INDEX` index hint is present. The idea is that if index use is forced, there is nothing to be gained from the additional overhead of performing dives into the index.
- The index is nonunique and not a `FULLTEXT` index.
- No subquery is present.
- No `DISTINCT`, `GROUP BY`, or `ORDER BY` clause is present.

For `EXPLAIN FOR CONNECTION`, the output changes as follows if index dives are skipped:

- For traditional output, the `rows` and `filtered` values are `NULL`.
- For JSON output, `rows_examined_per_scan` and `rows_produced_per_join` do not appear, `skip_index_dive_due_to_force` is `true`, and cost calculations are not accurate.

Without `FOR CONNECTION`, `EXPLAIN` output does not change when index dives are skipped.

After execution of a query for which index dives are skipped, the corresponding row in the `INFORMATION_SCHEMA.OPTIMIZER_TRACE` table contains an `index_dives_for_range_access` value of `skipped_due_to_force_index`.

## Skip Scan Range Access Method

Consider the following scenario:

```
CREATE TABLE t1 (f1 INT NOT NULL, f2 INT NOT NULL, PRIMARY KEY(f1, f2));
INSERT INTO t1 VALUES
  (1,1), (1,2), (1,3), (1,4), (1,5),
  (2,1), (2,2), (2,3), (2,4), (2,5);
INSERT INTO t1 SELECT f1, f2 + 5 FROM t1;
INSERT INTO t1 SELECT f1, f2 + 10 FROM t1;
INSERT INTO t1 SELECT f1, f2 + 20 FROM t1;
INSERT INTO t1 SELECT f1, f2 + 40 FROM t1;
ANALYZE TABLE t1;

EXPLAIN SELECT f1, f2 FROM t1 WHERE f2 > 40;
```

To execute this query, MySQL can choose an index scan to fetch all rows (the index includes all columns to be selected), then apply the `f2 > 40` condition from the `WHERE` clause to produce the final result set.

A range scan is more efficient than a full index scan, but cannot be used in this case because there is no condition on `f1`, the first index column. However, as of MySQL 8.0.13, the optimizer can perform multiple range scans, one for each value of `f1`, using a method called Skip Scan that is similar to Loose Index Scan (see [Section 8.2.1.15, “GROUP BY Optimization”](#)):

1. Skip between distinct values of the first index part, `f1` (the index prefix).
2. Perform a subrange scan on each distinct prefix value for the `f2 > 40` condition on the remaining index part.

For the data set shown earlier, the algorithm operates like this:

1. Get the first distinct value of the first key part (`f1 = 1`).
2. Construct the range based on the first and second key parts (`f1 = 1 AND f2 > 40`).
3. Perform a range scan.
4. Get the next distinct value of the first key part (`f1 = 2`).
5. Construct the range based on the first and second key parts (`f1 = 2 AND f2 > 40`).
6. Perform a range scan.

Using this strategy decreases the number of accessed rows because MySQL skips the rows that do not qualify for each constructed range. This Skip Scan access method is applicable under the following conditions:

- Table T has at least one compound index with key parts of the form `([A_1, ..., A_k], B_1, ..., B_m, C, D_1, ..., D_n)`. Key parts A and D may be empty, but B and C must be nonempty.
- The query references only one table.
- The query does not use `GROUP BY` or `DISTINCT`.
- The query references only columns in the index.
- The predicates on `A_1, ..., A_k` must be equality predicates and they must be constants. This includes the `IN()` operator.
- The query must be a conjunctive query; that is, an `AND` of `OR` conditions: `(cond1(key_part1) OR cond2(key_part1)) AND (cond1(key_part2) OR ...) AND ...`
- There must be a range condition on C.
- Conditions on D columns are permitted. Conditions on D must be in conjunction with the range condition on C.

Use of Skip Scan is indicated in `EXPLAIN` output as follows:

- `Using index for skip scan` in the `Extra` column indicates that the loose index Skip Scan access method is used.
- If the index can be used for Skip Scan, the index should be visible in the `possible_keys` column.

Use of Skip Scan is indicated in optimizer trace output by a "`skip scan`" element of this form:

```
"skip_scan_range": {
  "type": "skip_scan",
  "index": "index_used_for_skip_scan",
  "key_parts_used_for_access": [key_parts_used_for_access],
  "range": [range]
}
```

You may also see a `"best_skip_scan_summary"` element. If Skip Scan is chosen as the best range access variant, a `"chosen_range_access_summary"` is written. If Skip Scan is chosen as the overall best access method, a `"best_access_path"` element is present.

Use of Skip Scan is subject to the value of the `skip_scan` flag of the `optimizer_switch` system variable. See [Section 8.9.2, “Switchable Optimizations”](#). By default, this flag is `on`. To disable it, set `skip_scan` to `off`.

In addition to using the `optimizer_switch` system variable to control optimizer use of Skip Scan session-wide, MySQL supports optimizer hints to influence the optimizer on a per-statement basis. See [Section 8.9.3, “Optimizer Hints”](#).

## Range Optimization of Row Constructor Expressions

The optimizer is able to apply the range scan access method to queries of this form:

```
SELECT ... FROM t1 WHERE ( col_1, col_2 ) IN (( 'a', 'b' ), ( 'c', 'd' ));
```

Previously, for range scans to be used, it was necessary to write the query as:

```
SELECT ... FROM t1 WHERE ( col_1 = 'a' AND col_2 = 'b' )
OR ( col_1 = 'c' AND col_2 = 'd' );
```

For the optimizer to use a range scan, queries must satisfy these conditions:

- Only `IN()` predicates are used, not `NOT IN()`.
- On the left side of the `IN()` predicate, the row constructor contains only column references.
- On the right side of the `IN()` predicate, row constructors contain only runtime constants, which are either literals or local column references that are bound to constants during execution.
- On the right side of the `IN()` predicate, there is more than one row constructor.

For more information about the optimizer and row constructors, see [Section 8.2.1.20, “Row Constructor Expression Optimization”](#)

## Limiting Memory Use for Range Optimization

To control the memory available to the range optimizer, use the `range_optimizer_max_mem_size` system variable:

- A value of 0 means “no limit.”
- With a value greater than 0, the optimizer tracks the memory consumed when considering the range access method. If the specified limit is about to be exceeded, the range access method is abandoned and other methods, including a full table scan, are considered instead. This could be less optimal. If this happens, the following warning occurs (where `N` is the current `range_optimizer_max_mem_size` value):

```
Warning      3170      Memory capacity of N bytes for
                  'range_optimizer_max_mem_size' exceeded. Range
                  optimization was not done for this query.
```

- For `UPDATE` and `DELETE` statements, if the optimizer falls back to a full table scan and the `sql_safe_updates` system variable is enabled, an error occurs rather than a warning because, in

effect, no key is used to determine which rows to modify. For more information, see [Using Safe-Updates Mode \(--safe-updates\)](#).

For individual queries that exceed the available range optimization memory and for which the optimizer falls back to less optimal plans, increasing the `range_optimizer_max_mem_size` value may improve performance.

To estimate the amount of memory needed to process a range expression, use these guidelines:

- For a simple query such as the following, where there is one candidate key for the range access method, each predicate combined with `OR` uses approximately 230 bytes:

```
SELECT COUNT(*) FROM t
WHERE a=1 OR a=2 OR a=3 OR ... a=N;
```

- Similarly for a query such as the following, each predicate combined with `AND` uses approximately 125 bytes:

```
SELECT COUNT(*) FROM t
WHERE a=1 AND b=1 AND c=1 ... N;
```

- For a query with `IN( )` predicates:

```
SELECT COUNT(*) FROM t
WHERE a IN (1,2, ..., M) AND b IN (1,2, ..., N);
```

Each literal value in an `IN( )` list counts as a predicate combined with `OR`. If there are two `IN( )` lists, the number of predicates combined with `OR` is the product of the number of literal values in each list. Thus, the number of predicates combined with `OR` in the preceding case is  $M \times N$ .

### 8.2.1.3 Index Merge Optimization

The *Index Merge* access method retrieves rows with multiple `range` scans and merges their results into one. This access method merges index scans from a single table only, not scans across multiple tables. The merge can produce unions, intersections, or unions-of-intersections of its underlying scans.

Example queries for which Index Merge may be used:

```
SELECT * FROM tbl_name WHERE key1 = 10 OR key2 = 20;

SELECT * FROM tbl_name
  WHERE (key1 = 10 OR key2 = 20) AND non_key = 30;

SELECT * FROM t1, t2
  WHERE (t1.key1 IN (1,2) OR t1.key2 LIKE 'value%')
    AND t2.key1 = t1.some_col;

SELECT * FROM t1, t2
  WHERE t1.key1 = 1
    AND (t2.key1 = t1.some_col OR t2.key2 = t1.some_col2);
```



#### Note

The Index Merge optimization algorithm has the following known limitations:

- If your query has a complex `WHERE` clause with deep `AND/OR` nesting and MySQL does not choose the optimal plan, try distributing terms using the following identity transformations:

```
(x AND y) OR z => (x OR z) AND (y OR z)
(x OR y) AND z => (x AND z) OR (y AND z)
```

- Index Merge is not applicable to full-text indexes.

In `EXPLAIN` output, the Index Merge method appears as `index_merge` in the `type` column. In this case, the `key` column contains a list of indexes used, and `key_len` contains a list of the longest key parts for those indexes.

The Index Merge access method has several algorithms, which are displayed in the `Extra` field of `EXPLAIN` output:

- `Using intersect(...)`
- `Using union(...)`
- `Using sort_union(...)`

The following sections describe these algorithms in greater detail. The optimizer chooses between different possible Index Merge algorithms and other access methods based on cost estimates of the various available options.

- [Index Merge Intersection Access Algorithm](#)
- [Index Merge Union Access Algorithm](#)
- [Index Merge Sort-Union Access Algorithm](#)
- [Influencing Index Merge Optimization](#)

## Index Merge Intersection Access Algorithm

This access algorithm is applicable when a `WHERE` clause is converted to several range conditions on different keys combined with `AND`, and each condition is one of the following:

- An `N`-part expression of this form, where the index has exactly `N` parts (that is, all index parts are covered):

```
key_part1 = const1 AND key_part2 = const2 ... AND key_partN = constN
```

- Any range condition over the primary key of an `InnoDB` table.

Examples:

```
SELECT * FROM innodb_table
  WHERE primary_key < 10 AND key_col1 = 20;

SELECT * FROM tbl_name
  WHERE key1_part1 = 1 AND key1_part2 = 2 AND key2 = 2;
```

The Index Merge intersection algorithm performs simultaneous scans on all used indexes and produces the intersection of row sequences that it receives from the merged index scans.

If all columns used in the query are covered by the used indexes, full table rows are not retrieved (`EXPLAIN` output contains `Using index` in `Extra` field in this case). Here is an example of such a query:

```
SELECT COUNT(*) FROM t1 WHERE key1 = 1 AND key2 = 1;
```

If the used indexes do not cover all columns used in the query, full rows are retrieved only when the range conditions for all used keys are satisfied.

If one of the merged conditions is a condition over the primary key of an `InnoDB` table, it is not used for row retrieval, but is used to filter out rows retrieved using other conditions.

## Index Merge Union Access Algorithm

The criteria for this algorithm are similar to those for the Index Merge intersection algorithm. The algorithm is applicable when the table's `WHERE` clause is converted to several range conditions on different keys combined with `OR`, and each condition is one of the following:

- An `N`-part expression of this form, where the index has exactly `N` parts (that is, all index parts are covered):

```
key_part1 = const1 AND key_part2 = const2 ... AND key_partN = constN
```

- Any range condition over a primary key of an `InnoDB` table.
- A condition for which the Index Merge intersection algorithm is applicable.

Examples:

```
SELECT * FROM t1
  WHERE key1 = 1 OR key2 = 2 OR key3 = 3;

SELECT * FROM innodb_table
  WHERE (key1 = 1 AND key2 = 2)
    OR (key3 = 'foo' AND key4 = 'bar') AND key5 = 5;
```

## Index Merge Sort-Union Access Algorithm

This access algorithm is applicable when the `WHERE` clause is converted to several range conditions combined by `OR`, but the Index Merge union algorithm is not applicable.

Examples:

```
SELECT * FROM tbl_name
  WHERE key_col1 < 10 OR key_col2 < 20;

SELECT * FROM tbl_name
  WHERE (key_col1 > 10 OR key_col2 = 20) AND nonkey_col = 30;
```

The difference between the sort-union algorithm and the union algorithm is that the sort-union algorithm must first fetch row IDs for all rows and sort them before returning any rows.

## Influencing Index Merge Optimization

Use of Index Merge is subject to the value of the `index_merge`, `index_merge_intersection`, `index_merge_union`, and `index_merge_sort_union` flags of the `optimizer_switch` system variable. See [Section 8.9.2, “Switchable Optimizations”](#). By default, all those flags are `on`. To enable only certain algorithms, set `index_merge` to `off`, and enable only such of the others as should be permitted.

In addition to using the `optimizer_switch` system variable to control optimizer use of the Index Merge algorithms session-wide, MySQL supports optimizer hints to influence the optimizer on a per-statement basis. See [Section 8.9.3, “Optimizer Hints”](#).

#### 8.2.1.4 Engine Condition Pushdown Optimization

This optimization improves the efficiency of direct comparisons between a nonindexed column and a constant. In such cases, the condition is “pushed down” to the storage engine for evaluation. This optimization can be used only by the [NDB](#) storage engine.



##### Note

The [NDB](#) storage engine is currently not available in MySQL 8.0. If you are interested in using NDB Cluster, see [MySQL NDB Cluster 7.5 and NDB Cluster 7.6](#), which provides information about MySQL Cluster NDB 7.5 (based on MySQL 5.7 but containing the latest improvements and fixes for the [NDBCLUSTER](#) storage engine).

#### 8.2.1.5 Index Condition Pushdown Optimization

Index Condition Pushdown (ICP) is an optimization for the case where MySQL retrieves rows from a table using an index. Without ICP, the storage engine traverses the index to locate rows in the base table and returns them to the MySQL server which evaluates the [WHERE](#) condition for the rows. With ICP enabled, and if parts of the [WHERE](#) condition can be evaluated by using only columns from the index, the MySQL server pushes this part of the [WHERE](#) condition down to the storage engine. The storage engine then evaluates the pushed index condition by using the index entry and only if this is satisfied is the row read from the table. ICP can reduce the number of times the storage engine must access the base table and the number of times the MySQL server must access the storage engine.

Applicability of the Index Condition Pushdown optimization is subject to these conditions:

- ICP is used for the [range](#), [ref](#), [eq\\_ref](#), and [ref\\_or\\_null](#) access methods when there is a need to access full table rows.
- ICP can be used for [InnoDB](#) and [MyISAM](#) tables, including partitioned [InnoDB](#) and [MyISAM](#) tables.
- For [InnoDB](#) tables, ICP is used only for secondary indexes. The goal of ICP is to reduce the number of full-row reads and thereby reduce I/O operations. For [InnoDB](#) clustered indexes, the complete record is already read into the [InnoDB](#) buffer. Using ICP in this case does not reduce I/O.
- ICP is not supported with secondary indexes created on virtual generated columns. [InnoDB](#) supports secondary indexes on virtual generated columns.
- Conditions that refer to subqueries cannot be pushed down.
- Conditions that refer to stored functions cannot be pushed down. Storage engines cannot invoke stored functions.
- Triggered conditions cannot be pushed down. (For information about triggered conditions, see [Section 8.2.2.3, “Optimizing Subqueries with the EXISTS Strategy”](#).)

To understand how this optimization works, first consider how an index scan proceeds when Index Condition Pushdown is not used:

1. Get the next row, first by reading the index tuple, and then by using the index tuple to locate and read the full table row.
2. Test the part of the [WHERE](#) condition that applies to this table. Accept or reject the row based on the test result.

Using Index Condition Pushdown, the scan proceeds like this instead:

1. Get the next row's index tuple (but not the full table row).
2. Test the part of the `WHERE` condition that applies to this table and can be checked using only index columns. If the condition is not satisfied, proceed to the index tuple for the next row.
3. If the condition is satisfied, use the index tuple to locate and read the full table row.
4. Test the remaining part of the `WHERE` condition that applies to this table. Accept or reject the row based on the test result.

`EXPLAIN` output shows `Using index condition` in the `Extra` column when Index Condition Pushdown is used. It does not show `Using index` because that does not apply when full table rows must be read.

Suppose that a table contains information about people and their addresses and that the table has an index defined as `INDEX (zipcode, lastname, firstname)`. If we know a person's `zipcode` value but are not sure about the last name, we can search like this:

```
SELECT * FROM people
  WHERE zipcode='95054'
    AND lastname LIKE '%etrunia%'
    AND address LIKE '%Main Street%';
```

MySQL can use the index to scan through people with `zipcode='95054'`. The second part (`lastname LIKE '%etrunia%'`) cannot be used to limit the number of rows that must be scanned, so without Index Condition Pushdown, this query must retrieve full table rows for all people who have `zipcode='95054'`.

With Index Condition Pushdown, MySQL checks the `lastname LIKE '%etrunia%'` part before reading the full table row. This avoids reading full rows corresponding to index tuples that match the `zipcode` condition but not the `lastname` condition.

Index Condition Pushdown is enabled by default. It can be controlled with the `optimizer_switch` system variable by setting the `index_condition_pushdown` flag:

```
SET optimizer_switch = 'index_condition_pushdown=off';
SET optimizer_switch = 'index_condition_pushdown=on';
```

See [Section 8.9.2, “Switchable Optimizations”](#).

### 8.2.1.6 Nested-Loop Join Algorithms

MySQL executes joins between tables using a nested-loop algorithm or variations on it.

- [Nested-Loop Join Algorithm](#)
- [Block Nested-Loop Join Algorithm](#)

#### Nested-Loop Join Algorithm

A simple nested-loop join (NLJ) algorithm reads rows from the first table in a loop one at a time, passing each row to a nested loop that processes the next table in the join. This process is repeated as many times as there remain tables to be joined.

Assume that a join between three tables `t1`, `t2`, and `t3` is to be executed using the following join types:

Table	Join Type
t1	range
t2	ref

t3      ALL

If a simple NLJ algorithm is used, the join is processed like this:

```
for each row in t1 matching range {
    for each row in t2 matching reference key {
        for each row in t3 {
            if row satisfies join conditions, send to client
        }
    }
}
```

Because the NLJ algorithm passes rows one at a time from outer loops to inner loops, it typically reads tables processed in the inner loops many times.

## Block Nested-Loop Join Algorithm

A Block Nested-Loop (BNL) join algorithm uses buffering of rows read in outer loops to reduce the number of times that tables in inner loops must be read. For example, if 10 rows are read into a buffer and the buffer is passed to the next inner loop, each row read in the inner loop can be compared against all 10 rows in the buffer. This reduces by an order of magnitude the number of times the inner table must be read.

MySQL join buffering has these characteristics:

- Join buffering can be used when the join is of type `ALL` or `index` (in other words, when no possible keys can be used, and a full scan is done, of either the data or index rows, respectively), or `range`. Use of buffering is also applicable to outer joins, as described in [Section 8.2.1.11, “Block Nested-Loop and Batched Key Access Joins”](#).
- A join buffer is never allocated for the first nonconstant table, even if it would be of type `ALL` or `index`.
- Only columns of interest to a join are stored in its join buffer, not whole rows.
- The `join_buffer_size` system variable determines the size of each join buffer used to process a query.
- One buffer is allocated for each join that can be buffered, so a given query might be processed using multiple join buffers.
- A join buffer is allocated prior to executing the join and freed after the query is done.

For the example join described previously for the NLJ algorithm (without buffering), the join is done as follows using join buffering:

```
for each row in t1 matching range {
    for each row in t2 matching reference key {
        store used columns from t1, t2 in join buffer
        if buffer is full {
            for each row in t3 {
                for each t1, t2 combination in join buffer {
                    if row satisfies join conditions, send to client
                }
            }
            empty join buffer
        }
    }
}

if buffer is not empty {
```

```

        for each row in t3 {
            for each t1, t2 combination in join buffer {
                if row satisfies join conditions, send to client
            }
        }
    }
}

```

If  $s$  is the size of each stored  $t_1, t_2$  combination in the join buffer and  $c$  is the number of combinations in the buffer, the number of times table  $t_3$  is scanned is:

```
(s * c)/join_buffer_size + 1
```

The number of  $t_3$  scans decreases as the value of `join_buffer_size` increases, up to the point when `join_buffer_size` is large enough to hold all previous row combinations. At that point, no speed is gained by making it larger.

### 8.2.1.7 Nested Join Optimization

The syntax for expressing joins permits nested joins. The following discussion refers to the join syntax described in [Section 13.2.10.2, “JOIN Syntax”](#).

The syntax of `table_factor` is extended in comparison with the SQL Standard. The latter accepts only `table_reference`, not a list of them inside a pair of parentheses. This is a conservative extension if we consider each comma in a list of `table_reference` items as equivalent to an inner join. For example:

```
SELECT * FROM t1 LEFT JOIN (t2, t3, t4)
    ON (t2.a=t1.a AND t3.b=t1.b AND t4.c=t1.c)
```

Is equivalent to:

```
SELECT * FROM t1 LEFT JOIN (t2 CROSS JOIN t3 CROSS JOIN t4)
    ON (t2.a=t1.a AND t3.b=t1.b AND t4.c=t1.c)
```

In MySQL, `CROSS JOIN` is syntactically equivalent to `INNER JOIN`; they can replace each other. In standard SQL, they are not equivalent. `INNER JOIN` is used with an `ON` clause; `CROSS JOIN` is used otherwise.

In general, parentheses can be ignored in join expressions containing only inner join operations. Consider this join expression:

```
t1 LEFT JOIN (t2 LEFT JOIN t3 ON t2.b=t3.b OR t2.b IS NULL)
    ON t1.a=t2.a
```

After removing parentheses and grouping operations to the left, that join expression transforms into this expression:

```
(t1 LEFT JOIN t2 ON t1.a=t2.a) LEFT JOIN t3
    ON t2.b=t3.b OR t2.b IS NULL
```

Yet, the two expressions are not equivalent. To see this, suppose that the tables  $t_1$ ,  $t_2$ , and  $t_3$  have the following state:

- Table  $t_1$  contains rows `(1), (2)`
- Table  $t_2$  contains row `(1,101)`
- Table  $t_3$  contains row `(101)`

In this case, the first expression returns a result set including the rows `(1,1,101,101), (2,NULL,NULL,NULL)`, whereas the second expression returns the rows `(1,1,101,101), (2,NULL,NULL,101)`:

```
mysql> SELECT *
      FROM t1
        LEFT JOIN
          (t2 LEFT JOIN t3 ON t2.b=t3.b OR t2.b IS NULL)
        ON t1.a=t2.a;
+---+---+---+---+
| a | a | b | b |
+---+---+---+---+
| 1 | 1 | 101 | 101 |
| 2 | NULL | NULL | NULL |
+---+---+---+---+

mysql> SELECT *
      FROM (t1 LEFT JOIN t2 ON t1.a=t2.a)
        LEFT JOIN t3
        ON t2.b=t3.b OR t2.b IS NULL;
+---+---+---+---+
| a | a | b | b |
+---+---+---+---+
| 1 | 1 | 101 | 101 |
| 2 | NULL | NULL | 101 |
+---+---+---+---+
```

In the following example, an outer join operation is used together with an inner join operation:

```
t1 LEFT JOIN (t2, t3) ON t1.a=t2.a
```

That expression cannot be transformed into the following expression:

```
t1 LEFT JOIN t2 ON t1.a=t2.a, t3
```

For the given table states, the two expressions return different sets of rows:

```
mysql> SELECT *
      FROM t1 LEFT JOIN (t2, t3) ON t1.a=t2.a;
+---+---+---+---+
| a | a | b | b |
+---+---+---+---+
| 1 | 1 | 101 | 101 |
| 2 | NULL | NULL | NULL |
+---+---+---+---+

mysql> SELECT *
      FROM t1 LEFT JOIN t2 ON t1.a=t2.a, t3;
+---+---+---+---+
| a | a | b | b |
+---+---+---+---+
| 1 | 1 | 101 | 101 |
| 2 | NULL | NULL | 101 |
+---+---+---+---+
```

Therefore, if we omit parentheses in a join expression with outer join operators, we might change the result set for the original expression.

More exactly, we cannot ignore parentheses in the right operand of the left outer join operation and in the left operand of a right join operation. In other words, we cannot ignore parentheses for the inner table

expressions of outer join operations. Parentheses for the other operand (operand for the outer table) can be ignored.

The following expression:

```
(t1,t2) LEFT JOIN t3 ON P(t2.b,t3.b)
```

Is equivalent to this expression for any tables `t1, t2, t3` and any condition `P` over attributes `t2.b` and `t3.b`:

```
t1, t2 LEFT JOIN t3 ON P(t2.b,t3.b)
```

Whenever the order of execution of join operations in a join expression (*joined\_table*) is not from left to right, we talk about nested joins. Consider the following queries:

```
SELECT * FROM t1 LEFT JOIN (t2 LEFT JOIN t3 ON t2.b=t3.b) ON t1.a=t2.a
      WHERE t1.a > 1

SELECT * FROM t1 LEFT JOIN (t2, t3) ON t1.a=t2.a
      WHERE (t2.b=t3.b OR t2.b IS NULL) AND t1.a > 1
```

Those queries are considered to contain these nested joins:

```
t2 LEFT JOIN t3 ON t2.b=t3.b
t2, t3
```

In the first query, the nested join is formed with a left join operation. In the second query, it is formed with an inner join operation.

In the first query, the parentheses can be omitted: The grammatical structure of the join expression will dictate the same order of execution for join operations. For the second query, the parentheses cannot be omitted, although the join expression here can be interpreted unambiguously without them. In our extended syntax, the parentheses in `(t2, t3)` of the second query are required, although theoretically the query could be parsed without them: We still would have unambiguous syntactical structure for the query because `LEFT JOIN` and `ON` play the role of the left and right delimiters for the expression `(t2, t3)`.

The preceding examples demonstrate these points:

- For join expressions involving only inner joins (and not outer joins), parentheses can be removed and joins evaluated left to right. In fact, tables can be evaluated in any order.
- The same is not true, in general, for outer joins or for outer joins mixed with inner joins. Removal of parentheses may change the result.

Queries with nested outer joins are executed in the same pipeline manner as queries with inner joins. More exactly, a variation of the nested-loop join algorithm is exploited. Recall the algorithm by which the nested-loop join executes a query (see [Section 8.2.1.6, “Nested-Loop Join Algorithms”](#)). Suppose that a join query over 3 tables `T1, T2, T3` has this form:

```
SELECT * FROM T1 INNER JOIN T2 ON P1(T1,T2)
      INNER JOIN T3 ON P2(T2,T3)
      WHERE P(T1,T2,T3)
```

Here, `P1(T1, T2)` and `P2(T3, T3)` are some join conditions (on expressions), whereas `P(T1, T2, T3)` is a condition over columns of tables `T1, T2, T3`.

The nested-loop join algorithm would execute this query in the following manner:

```

FOR each row t1 in T1 {
    FOR each row t2 in T2 such that P1(t1,t2) {
        FOR each row t3 in T3 such that P2(t2,t3) {
            IF P(t1,t2,t3) {
                t:=t1||t2||t3; OUTPUT t;
            }
        }
    }
}
    
```

The notation  $t_1||t_2||t_3$  indicates a row constructed by concatenating the columns of rows  $t_1$ ,  $t_2$ , and  $t_3$ . In some of the following examples, `NULL` where a table name appears means a row in which `NULL` is used for each column of that table. For example,  $t_1||t_2||NULL$  indicates a row constructed by concatenating the columns of rows  $t_1$  and  $t_2$ , and `NULL` for each column of  $t_3$ . Such a row is said to be `NULL`-complemented.

Now consider a query with nested outer joins:

```

SELECT * FROM T1 LEFT JOIN
    (T2 LEFT JOIN T3 ON P2(T2,T3))
    ON P1(T1,T2)
WHERE P(T1,T2,T3)
    
```

For this query, modify the nested-loop pattern to obtain:

```

FOR each row t1 in T1 {
    BOOL f1:=FALSE;
    FOR each row t2 in T2 such that P1(t1,t2) {
        BOOL f2:=FALSE;
        FOR each row t3 in T3 such that P2(t2,t3) {
            IF P(t1,t2,t3) {
                t:=t1||t2||t3; OUTPUT t;
            }
            f2=TRUE;
            f1=TRUE;
        }
        IF (!f2) {
            IF P(t1,t2,NULL) {
                t:=t1||t2||NULL; OUTPUT t;
            }
            f1=TRUE;
        }
    }
    IF (!f1) {
        IF P(t1,NULL,NULL) {
            t:=t1||NULL||NULL; OUTPUT t;
        }
    }
}
    
```

In general, for any nested loop for the first inner table in an outer join operation, a flag is introduced that is turned off before the loop and is checked after the loop. The flag is turned on when for the current row from the outer table a match from the table representing the inner operand is found. If at the end of the loop cycle the flag is still off, no match has been found for the current row of the outer table. In this case, the row is complemented by `NULL` values for the columns of the inner tables. The result row is passed to the final check for the output or into the next nested loop, but only if the row satisfies the join condition of all embedded outer joins.

In the example, the outer join table expressed by the following expression is embedded:

```
(T2 LEFT JOIN T3 ON P2(T2,T3))
```

For the query with inner joins, the optimizer could choose a different order of nested loops, such as this one:

```
FOR each row t3 in T3 {
    FOR each row t2 in T2 such that P2(t2,t3) {
        FOR each row t1 in T1 such that P1(t1,t2) {
            IF P(t1,t2,t3) {
                t:=t1||t2||t3; OUTPUT t;
            }
        }
    }
}
```

For queries with outer joins, the optimizer can choose only such an order where loops for outer tables precede loops for inner tables. Thus, for our query with outer joins, only one nesting order is possible. For the following query, the optimizer evaluates two different nestings. In both nestings, [T1](#) must be processed in the outer loop because it is used in an outer join. [T2](#) and [T3](#) are used in an inner join, so that join must be processed in the inner loop. However, because the join is an inner join, [T2](#) and [T3](#) can be processed in either order.

```
SELECT * T1 LEFT JOIN (T2,T3) ON P1(T1,T2) AND P2(T1,T3)
      WHERE P(T1,T2,T3)
```

One nesting evaluates [T2](#), then [T3](#):

```
FOR each row t1 in T1 {
    BOOL f1:=FALSE;
    FOR each row t2 in T2 such that P1(t1,t2) {
        FOR each row t3 in T3 such that P2(t1,t3) {
            IF P(t1,t2,t3) {
                t:=t1||t2||t3; OUTPUT t;
            }
            f1:=TRUE
        }
    }
    IF (!f1) {
        IF P(t1,NULL,NULL) {
            t:=t1||NULL||NULL; OUTPUT t;
        }
    }
}
```

The other nesting evaluates [T3](#), then [T2](#):

```
FOR each row t1 in T1 {
    BOOL f1:=FALSE;
    FOR each row t3 in T3 such that P2(t1,t3) {
        FOR each row t2 in T2 such that P1(t1,t2) {
            IF P(t1,t2,t3) {
                t:=t1||t2||t3; OUTPUT t;
            }
            f1:=TRUE
        }
    }
    IF (!f1) {
        IF P(t1,NULL,NULL) {
            t:=t1||NULL||NULL; OUTPUT t;
        }
    }
}
```

```

    }
}
```

When discussing the nested-loop algorithm for inner joins, we omitted some details whose impact on the performance of query execution may be huge. We did not mention so-called “pushed-down” conditions. Suppose that our `WHERE` condition `P(T1, T2, T3)` can be represented by a conjunctive formula:

```
P(T1,T2,T3) = C1(T1) AND C2(T2) AND C3(T3).
```

In this case, MySQL actually uses the following nested-loop algorithm for the execution of the query with inner joins:

```

FOR each row t1 in T1 such that C1(t1) {
    FOR each row t2 in T2 such that P1(t1,t2) AND C2(t2) {
        FOR each row t3 in T3 such that P2(t2,t3) AND C3(t3) {
            IF P(t1,t2,t3) {
                t:=t1||t2||t3; OUTPUT t;
            }
        }
    }
}
```

You see that each of the conjuncts `C1(T1)`, `C2(T2)`, `C3(T3)` are pushed out of the most inner loop to the most outer loop where it can be evaluated. If `C1(T1)` is a very restrictive condition, this condition pushdown may greatly reduce the number of rows from table `T1` passed to the inner loops. As a result, the execution time for the query may improve immensely.

For a query with outer joins, the `WHERE` condition is to be checked only after it has been found that the current row from the outer table has a match in the inner tables. Thus, the optimization of pushing conditions out of the inner nested loops cannot be applied directly to queries with outer joins. Here we must introduce conditional pushed-down predicates guarded by the flags that are turned on when a match has been encountered.

Recall this example with outer joins:

```
P(T1,T2,T3)=C1(T1) AND C(T2) AND C3(T3)
```

For that example, the nested-loop algorithm using guarded pushed-down conditions looks like this:

```

FOR each row t1 in T1 such that C1(t1) {
    BOOL f1:=FALSE;
    FOR each row t2 in T2
        such that P1(t1,t2) AND (f1?C2(t2):TRUE) {
            BOOL f2:=FALSE;
            FOR each row t3 in T3
                such that P2(t2,t3) AND (f1&&f2?C3(t3):TRUE) {
                    IF (f1&&f2?TRUE:(C2(t2) AND C3(t3))) {
                        t:=t1||t2||t3; OUTPUT t;
                    }
                    f2=TRUE;
                    f1=TRUE;
                }
                IF (!f2) {
                    IF (f1?TRUE:C2(t2) && P(t1,t2,NULL)) {
                        t:=t1||t2||NULL; OUTPUT t;
                    }
                    f1=TRUE;
                }
            }
        }
}
```

```

IF (!f1 && P(t1,NULL,NULL)) {
    t:=t1||NULL||NULL; OUTPUT t;
}

```

In general, pushed-down predicates can be extracted from join conditions such as `P1(T1,T2)` and `P(T2,T3)`. In this case, a pushed-down predicate is guarded also by a flag that prevents checking the predicate for the `NULL`-complemented row generated by the corresponding outer join operation.

Access by key from one inner table to another in the same nested join is prohibited if it is induced by a predicate from the `WHERE` condition.

### 8.2.1.8 Outer Join Optimization

Outer joins include `LEFT JOIN` and `RIGHT JOIN`.

MySQL implements an `A LEFT JOIN B join_specification` as follows:

- Table `B` is set to depend on table `A` and all tables on which `A` depends.
- Table `A` is set to depend on all tables (except `B`) that are used in the `LEFT JOIN` condition.
- The `LEFT JOIN` condition is used to decide how to retrieve rows from table `B`. (In other words, any condition in the `WHERE` clause is not used.)
- All standard join optimizations are performed, with the exception that a table is always read after all tables on which it depends. If there is a circular dependency, an error occurs.
- All standard `WHERE` optimizations are performed.
- If there is a row in `A` that matches the `WHERE` clause, but there is no row in `B` that matches the `ON` condition, an extra `B` row is generated with all columns set to `NULL`.
- If you use `LEFT JOIN` to find rows that do not exist in some table and you have the following test: `col_name IS NULL` in the `WHERE` part, where `col_name` is a column that is declared as `NOT NULL`, MySQL stops searching for more rows (for a particular key combination) after it has found one row that matches the `LEFT JOIN` condition.

The `RIGHT JOIN` implementation is analogous to that of `LEFT JOIN` with the table roles reversed. Right joins are converted to equivalent left joins, as described in [Section 8.2.1.9, “Outer Join Simplification”](#).

For a `LEFT JOIN`, if the `WHERE` condition is always false for the generated `NULL` row, the `LEFT JOIN` is changed to an inner join. For example, the `WHERE` clause would be false in the following query if `t2.column1` were `NULL`:

```
SELECT * FROM t1 LEFT JOIN t2 ON (column1) WHERE t2.column2=5;
```

Therefore, it is safe to convert the query to an inner join:

```
SELECT * FROM t1, t2 WHERE t2.column2=5 AND t1.column1=t2.column1;
```

In MySQL 8.0.14 and later, trivial `WHERE` conditions arising from constant literal expressions are removed during preparation, rather than at a later stage in optimization, by which time joins have already been simplified. Earlier removal of trivial conditions allows the optimizer to convert outer joins to inner joins; this can result in improved plans for queries with outer joins containing trivial conditions in the `WHERE` clause, such as this one:

```
SELECT * FROM t1 LEFT JOIN t2 ON condition_1 WHERE condition_2 OR 0 = 1
```

The optimizer now sees during preparation that  $0 = 1$  is always false, making `OR 0 = 1` redundant, and removes it, leaving this:

```
SELECT * FROM t1 LEFT JOIN t2 ON condition_1 WHERE condition_2
```

Now the optimizer can rewrite the query as an inner join, like this:

```
SELECT * FROM t1 JOIN t2 WHERE condition_1 AND condition_2
```

Now the optimizer can use table `t2` before table `t1` if doing so would result in a better query plan.

To provide a hint about the table join order, use optimizer hints; see [Section 8.9.3, “Optimizer Hints”](#).

Alternatively, use `STRAIGHT_JOIN`; see [Section 13.2.10, “SELECT Syntax”](#). However, `STRAIGHT_JOIN` may prevent indexes from being used because it disables semi-join transformations; see [Section 8.2.2.1, “Optimizing Subqueries, Derived Tables, View References, and Common Table Expressions with Semi-Join Transformations”](#).

### 8.2.1.9 Outer Join Simplification

Table expressions in the `FROM` clause of a query are simplified in many cases.

At the parser stage, queries with right outer join operations are converted to equivalent queries containing only left join operations. In the general case, the conversion is performed such that this right join:

```
(T1, ...) RIGHT JOIN (T2, ...) ON P(T1, ..., T2, ...)
```

Becomes this equivalent left join:

```
(T2, ...) LEFT JOIN (T1, ...) ON P(T1, ..., T2, ...)
```

All inner join expressions of the form `T1 INNER JOIN T2 ON P(T1,T2)` are replaced by the list `T1, T2, P(T1, T2)` being joined as a conjunct to the `WHERE` condition (or to the join condition of the embedding join, if there is any).

When the optimizer evaluates plans for outer join operations, it takes into consideration only plans where, for each such operation, the outer tables are accessed before the inner tables. The optimizer choices are limited because only such plans enable outer joins to be executed using the nested-loop algorithm.

Consider a query of this form, where `R(T2)` greatly narrows the number of matching rows from table `T2`:

```
SELECT * T1 LEFT JOIN T2 ON P1(T1,T2)
  WHERE P(T1,T2) AND R(T2)
```

If the query is executed as written, the optimizer has no choice but to access the less-restricted table `T1` before the more-restricted table `T2`, which may produce a very inefficient execution plan.

Instead, MySQL converts the query to a query with no outer join operation if the `WHERE` condition is null-rejected. (That is, it converts the outer join to an inner join.) A condition is said to be null-rejected for an outer join operation if it evaluates to `FALSE` or `UNKNOWN` for any `NULL`-complemented row generated for the operation.

Thus, for this outer join:

```
T1 LEFT JOIN T2 ON T1.A=T2.A
```

Conditions such as these are null-rejected because they cannot be true for any `NULL`-complemented row (with `T2` columns set to `NULL`):

```
T2.B IS NOT NULL
T2.B > 3
T2.C <= T1.C
T2.B < 2 OR T2.C > 1
```

Conditions such as these are not null-rejected because they might be true for a `NULL`-complemented row:

```
T2.B IS NULL
T1.B < 3 OR T2.B IS NOT NULL
T1.B < 3 OR T2.B > 3
```

The general rules for checking whether a condition is null-rejected for an outer join operation are simple:

- It is of the form `A IS NOT NULL`, where `A` is an attribute of any of the inner tables
- It is a predicate containing a reference to an inner table that evaluates to `UNKNOWN` when one of its arguments is `NULL`
- It is a conjunction containing a null-rejected condition as a conjunct
- It is a disjunction of null-rejected conditions

A condition can be null-rejected for one outer join operation in a query and not null-rejected for another. In this query, the `WHERE` condition is null-rejected for the second outer join operation but is not null-rejected for the first one:

```
SELECT * FROM T1 LEFT JOIN T2 ON T2.A=T1.A
    LEFT JOIN T3 ON T3.B=T1.B
WHERE T3.C > 0
```

If the `WHERE` condition is null-rejected for an outer join operation in a query, the outer join operation is replaced by an inner join operation.

For example, in the preceding query, the second outer join is null-rejected and can be replaced by an inner join:

```
SELECT * FROM T1 LEFT JOIN T2 ON T2.A=T1.A
    INNER JOIN T3 ON T3.B=T1.B
WHERE T3.C > 0
```

For the original query, the optimizer evaluates only plans compatible with the single table-access order `T1, T2, T3`. For the rewritten query, it additionally considers the access order `T3, T1, T2`.

A conversion of one outer join operation may trigger a conversion of another. Thus, the query:

```
SELECT * FROM T1 LEFT JOIN T2 ON T2.A=T1.A
    LEFT JOIN T3 ON T3.B=T2.B
WHERE T3.C > 0
```

Is first converted to the query:

```
SELECT * FROM T1 LEFT JOIN T2 ON T2.A=T1.A
```

```
    INNER JOIN T3 ON T3.B=T2.B
WHERE T3.C > 0
```

Which is equivalent to the query:

```
SELECT * FROM (T1 LEFT JOIN T2 ON T2.A=T1.A), T3
WHERE T3.C > 0 AND T3.B=T2.B
```

The remaining outer join operation can also be replaced by an inner join because the condition `T3.B=T2.B` is null-rejected. This results in a query with no outer joins at all:

```
SELECT * FROM (T1 INNER JOIN T2 ON T2.A=T1.A), T3
WHERE T3.C > 0 AND T3.B=T2.B
```

Sometimes the optimizer succeeds in replacing an embedded outer join operation, but cannot convert the embedding outer join. The following query:

```
SELECT * FROM T1 LEFT JOIN
    (T2 LEFT JOIN T3 ON T3.B=T2.B)
    ON T2.A=T1.A
WHERE T3.C > 0
```

Is converted to:

```
SELECT * FROM T1 LEFT JOIN
    (T2 INNER JOIN T3 ON T3.B=T2.B)
    ON T2.A=T1.A
WHERE T3.C > 0
```

That can be rewritten only to the form still containing the embedding outer join operation:

```
SELECT * FROM T1 LEFT JOIN
    (T2,T3)
    ON (T2.A=T1.A AND T3.B=T2.B)
WHERE T3.C > 0
```

Any attempt to convert an embedded outer join operation in a query must take into account the join condition for the embedding outer join together with the `WHERE` condition. In this query, the `WHERE` condition is not null-rejected for the embedded outer join, but the join condition of the embedding outer join `T2.A=T1.A AND T3.C=T1.C` is null-rejected:

```
SELECT * FROM T1 LEFT JOIN
    (T2 LEFT JOIN T3 ON T3.B=T2.B)
    ON T2.A=T1.A AND T3.C=T1.C
WHERE T3.D > 0 OR T1.D > 0
```

Consequently, the query can be converted to:

```
SELECT * FROM T1 LEFT JOIN
    (T2, T3)
    ON T2.A=T1.A AND T3.C=T1.C AND T3.B=T2.B
WHERE T3.D > 0 OR T1.D > 0
```

### 8.2.1.10 Multi-Range Read Optimization

Reading rows using a range scan on a secondary index can result in many random disk accesses to the base table when the table is large and not stored in the storage engine's cache. With the Disk-Sweep

Multi-Range Read (MRR) optimization, MySQL tries to reduce the number of random disk access for range scans by first scanning the index only and collecting the keys for the relevant rows. Then the keys are sorted and finally the rows are retrieved from the base table using the order of the primary key. The motivation for Disk-sweep MRR is to reduce the number of random disk accesses and instead achieve a more sequential scan of the base table data.

The Multi-Range Read optimization provides these benefits:

- MRR enables data rows to be accessed sequentially rather than in random order, based on index tuples. The server obtains a set of index tuples that satisfy the query conditions, sorts them according to data row ID order, and uses the sorted tuples to retrieve data rows in order. This makes data access more efficient and less expensive.
- MRR enables batch processing of requests for key access for operations that require access to data rows through index tuples, such as range index scans and equi-joins that use an index for the join attribute. MRR iterates over a sequence of index ranges to obtain qualifying index tuples. As these results accumulate, they are used to access the corresponding data rows. It is not necessary to acquire all index tuples before starting to read data rows.

The MRR optimization is not supported with secondary indexes created on virtual generated columns. [InnoDB](#) supports secondary indexes on virtual generated columns.

The following scenarios illustrate when MRR optimization can be advantageous:

Scenario A: MRR can be used for [InnoDB](#) and [MyISAM](#) tables for index range scans and equi-join operations.

1. A portion of the index tuples are accumulated in a buffer.
2. The tuples in the buffer are sorted by their data row ID.
3. Data rows are accessed according to the sorted index tuple sequence.

Scenario B: MRR can be used for [NDB](#) tables for multiple-range index scans or when performing an equi-join by an attribute.

1. A portion of ranges, possibly single-key ranges, is accumulated in a buffer on the central node where the query is submitted.
2. The ranges are sent to the execution nodes that access data rows.
3. The accessed rows are packed into packages and sent back to the central node.
4. The received packages with data rows are placed in a buffer.
5. Data rows are read from the buffer.

When MRR is used, the [Extra](#) column in [EXPLAIN](#) output shows [Using MRR](#).

[InnoDB](#) and [MyISAM](#) do not use MRR if full table rows need not be accessed to produce the query result. This is the case if results can be produced entirely on the basis on information in the index tuples (through a [covering index](#)); MRR provides no benefit.

Two [optimizer\\_switch](#) system variable flags provide an interface to the use of MRR optimization. The [mrr](#) flag controls whether MRR is enabled. If [mrr](#) is enabled ([on](#)), the [mrr\\_cost\\_based](#) flag controls whether the optimizer attempts to make a cost-based choice between using and not using MRR ([on](#)) or uses MRR whenever possible ([off](#)). By default, [mrr](#) is [on](#) and [mrr\\_cost\\_based](#) is [on](#). See [Section 8.9.2, “Switchable Optimizations”](#).

For MRR, a storage engine uses the value of the `read_rnd_buffer_size` system variable as a guideline for how much memory it can allocate for its buffer. The engine uses up to `read_rnd_buffer_size` bytes and determines the number of ranges to process in a single pass.

### 8.2.1.11 Block Nested-Loop and Batched Key Access Joins

In MySQL, a Batched Key Access (BKA) Join algorithm is available that uses both index access to the joined table and a join buffer. The BKA algorithm supports inner join, outer join, and semi-join operations, including nested outer joins. Benefits of BKA include improved join performance due to more efficient table scanning. Also, the Block Nested-Loop (BNL) Join algorithm previously used only for inner joins is extended and can be employed for outer join and semi-join operations, including nested outer joins.

The following sections discuss the join buffer management that underlies the extension of the original BNL algorithm, the extended BNL algorithm, and the BKA algorithm. For information about semi-join strategies, see [Section 8.2.2.1, “Optimizing Subqueries, Derived Tables, View References, and Common Table Expressions with Semi-Join Transformations”](#)

- [Join Buffer Management for Block Nested-Loop and Batched Key Access Algorithms](#)
- [Block Nested-Loop Algorithm for Outer Joins and Semi-Joins](#)
- [Batched Key Access Joins](#)
- [Optimizer Hints for Block Nested-Loop and Batched Key Access Algorithms](#)

#### Join Buffer Management for Block Nested-Loop and Batched Key Access Algorithms

MySQL can employ join buffers to execute not only inner joins without index access to the inner table, but also outer joins and semi-joins that appear after subquery flattening. Moreover, a join buffer can be effectively used when there is an index access to the inner table.

The join buffer management code slightly more efficiently utilizes join buffer space when storing the values of the interesting row columns: No additional bytes are allocated in buffers for a row column if its value is `NULL`, and the minimum number of bytes is allocated for any value of the `VARCHAR` type.

The code supports two types of buffers, regular and incremental. Suppose that join buffer `B1` is employed to join tables `t1` and `t2` and the result of this operation is joined with table `t3` using join buffer `B2`:

- A regular join buffer contains columns from each join operand. If `B2` is a regular join buffer, each row `r` put into `B2` is composed of the columns of a row `r1` from `B1` and the interesting columns of a matching row `r2` from table `t3`.
- An incremental join buffer contains only columns from rows of the table produced by the second join operand. That is, it is incremental to a row from the first operand buffer. If `B2` is an incremental join buffer, it contains the interesting columns of the row `r2` together with a link to the row `r1` from `B1`.

Incremental join buffers are always incremental relative to a join buffer from an earlier join operation, so the buffer from the first join operation is always a regular buffer. In the example just given, the buffer `B1` used to join tables `t1` and `t2` must be a regular buffer.

Each row of the incremental buffer used for a join operation contains only the interesting columns of a row from the table to be joined. These columns are augmented with a reference to the interesting columns of the matched row from the table produced by the first join operand. Several rows in the incremental buffer can refer to the same row `r` whose columns are stored in the previous join buffers insofar as all these rows match row `r`.

Incremental buffers enable less frequent copying of columns from buffers used for previous join operations. This provides a savings in buffer space because in the general case a row produced by the first join

operand can be matched by several rows produced by the second join operand. It is unnecessary to make several copies of a row from the first operand. Incremental buffers also provide a savings in processing time due to the reduction in copying time.

The `block_nested_loop` and `batched_key_access` flags of the `optimizer_switch` system variable control how the optimizer uses the Block Nested-Loop and Batched Key Access join algorithms. By default, `block_nested_loop` is `on` and `batched_key_access` is `off`. See [Section 8.9.2, “Switchable Optimizations”](#). Optimizer hints may also be applied; see [Optimizer Hints for Block Nested-Loop and Batched Key Access Algorithms](#).

For information about semi-join strategies, see [Section 8.2.2.1, “Optimizing Subqueries, Derived Tables, View References, and Common Table Expressions with Semi-Join Transformations”](#)

## Block Nested-Loop Algorithm for Outer Joins and Semi-Joins

The original implementation of the MySQL BNL algorithm is extended to support outer join and semi-join operations.

When these operations are executed with a join buffer, each row put into the buffer is supplied with a match flag.

If an outer join operation is executed using a join buffer, each row of the table produced by the second operand is checked for a match against each row in the join buffer. When a match is found, a new extended row is formed (the original row plus columns from the second operand) and sent for further extensions by the remaining join operations. In addition, the match flag of the matched row in the buffer is enabled. After all rows of the table to be joined have been examined, the join buffer is scanned. Each row from the buffer that does not have its match flag enabled is extended by `NULL` complements (`NULL` values for each column in the second operand) and sent for further extensions by the remaining join operations.

The `block_nested_loop` flag of the `optimizer_switch` system variable controls how the optimizer uses the Block Nested-Loop algorithm. By default, `block_nested_loop` is `on`. See [Section 8.9.2, “Switchable Optimizations”](#). Optimizer hints may also be applied; see [Optimizer Hints for Block Nested-Loop and Batched Key Access Algorithms](#).

In `EXPLAIN` output, use of BNL for a table is signified when the `Extra` value contains `Using join buffer (Block Nested Loop)` and the `type` value is `ALL`, `index`, or `range`.

For information about semi-join strategies, see [Section 8.2.2.1, “Optimizing Subqueries, Derived Tables, View References, and Common Table Expressions with Semi-Join Transformations”](#)

## Batched Key Access Joins

MySQL implements a method of joining tables called the Batched Key Access (BKA) join algorithm. BKA can be applied when there is an index access to the table produced by the second join operand. Like the BNL join algorithm, the BKA join algorithm employs a join buffer to accumulate the interesting columns of the rows produced by the first operand of the join operation. Then the BKA algorithm builds keys to access the table to be joined for all rows in the buffer and submits these keys in a batch to the database engine for index lookups. The keys are submitted to the engine through the Multi-Range Read (MRR) interface (see [Section 8.2.1.10, “Multi-Range Read Optimization”](#)). After submission of the keys, the MRR engine functions perform lookups in the index in an optimal way, fetching the rows of the joined table found by these keys, and starts feeding the BKA join algorithm with matching rows. Each matching row is coupled with a reference to a row in the join buffer.

When BKA is used, the value of `join_buffer_size` defines how large the batch of keys is in each request to the storage engine. The larger the buffer, the more sequential access will be to the right hand table of a join operation, which can significantly improve performance.

For BKA to be used, the `batched_key_access` flag of the `optimizer_switch` system variable must be set to `on`. BKA uses MRR, so 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. The following setting enables BKA:

```
mysql> SET optimizer_switch='mrr=on,mrr_cost_based=off,batched_key_access=on';
```

There are two scenarios by which MRR functions execute:

- The first scenario is used for conventional disk-based storage engines such as `InnoDB` and `MyISAM`. For these engines, usually the keys for all rows from the join buffer are submitted to the MRR interface at once. Engine-specific MRR functions perform index lookups for the submitted keys, get row IDs (or primary keys) from them, and then fetch rows for all these selected row IDs one by one by request from BKA algorithm. Every row is returned with an association reference that enables access to the matched row in the join buffer. The rows are fetched by the MRR functions in an optimal way: They are fetched in the row ID (primary key) order. This improves performance because reads are in disk order rather than random order.
- The second scenario is used for remote storage engines such as `NDB`. A package of keys for a portion of rows from the join buffer, together with their associations, is sent by a MySQL Server (SQL node) to MySQL Cluster data nodes. In return, the SQL node receives a package (or several packages) of matching rows coupled with corresponding associations. The BKA join algorithm takes these rows and builds new joined rows. Then a new set of keys is sent to the data nodes and the rows from the returned packages are used to build new joined rows. The process continues until the last keys from the join buffer are sent to the data nodes, and the SQL node has received and joined all rows matching these keys. This improves performance because fewer key-bearing packages sent by the SQL node to the data nodes means fewer round trips between it and the data nodes to perform the join operation.

With the first scenario, a portion of the join buffer is reserved to store row IDs (primary keys) selected by index lookups and passed as a parameter to the MRR functions.

There is no special buffer to store keys built for rows from the join buffer. Instead, a function that builds the key for the next row in the buffer is passed as a parameter to the MRR functions.

In `EXPLAIN` output, use of BKA for a table is signified when the `Extra` value contains `Using join buffer (Batched Key Access)` and the `type` value is `ref` or `eq_ref`.

## Optimizer Hints for Block Nested-Loop and Batched Key Access Algorithms

In addition to using the `optimizer_switch` system variable to control optimizer use of the BNL and BKA algorithms session-wide, MySQL supports optimizer hints to influence the optimizer on a per-statement basis. See [Section 8.9.3, “Optimizer Hints”](#).

To use a BNL or BKA 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.

### 8.2.1.12 Condition Filtering

In join processing, prefix rows are those rows passed from one table in a join to the next. In general, the optimizer attempts to put tables with low prefix counts early in the join order to keep the number of row combinations from increasing rapidly. To the extent that the optimizer can use information about conditions on rows selected from one table and passed to the next, the more accurately it can compute row estimates and choose the best execution plan.

Without condition filtering, the prefix row count for a table is based on the estimated number of rows selected by the `WHERE` clause according to whichever access method the optimizer chooses. Condition

filtering enables the optimizer to use other relevant conditions in the `WHERE` clause not taken into account by the access method, and thus improve its prefix row count estimates. For example, even though there might be an index-based access method that can be used to select rows from the current table in a join, there might also be additional conditions for the table in the `WHERE` clause that can filter (further restrict) the estimate for qualifying rows passed to the next table.

A condition contributes to the filtering estimate only if:

- It refers to the current table.
- It depends on a constant value or values from earlier tables in the join sequence.
- It was not already taken into account by the access method.

In `EXPLAIN` output, the `rows` column indicates the row estimate for the chosen access method, and the `filtered` column reflects the effect of condition filtering. `filtered` values are expressed as percentages. The maximum value is 100, which means no filtering of rows occurred. Values decreasing from 100 indicate increasing amounts of filtering.

The prefix row count (the number of rows estimated to be passed from the current table in a join to the next) is the product of the `rows` and `filtered` values. That is, the prefix row count is the estimated row count, reduced by the estimated filtering effect. For example, if `rows` is 1000 and `filtered` is 20%, condition filtering reduces the estimated row count of 1000 to a prefix row count of  $1000 \times .20 = 1000 \times .2 = 200$ .

Consider the following query:

```
SELECT *
  FROM employee JOIN department ON employee.dept_no = department.dept_no
 WHERE employee.first_name = 'John'
   AND employee.hire_date BETWEEN '2018-01-01' AND '2018-06-01';
```

Suppose that the data set has these characteristics:

- The `employee` table has 1024 rows.
- The `department` table has 12 rows.
- Both tables have an index on `dept_no`.
- The `employee` table has an index on `first_name`.
- 8 rows satisfy this condition on `employee.first_name`:

```
employee.first_name = 'John'
```

- 150 rows satisfy this condition on `employee.hire_date`:

```
employee.hire_date BETWEEN '2018-01-01' AND '2018-06-01'
```

- 1 row satisfies both conditions:

```
employee.first_name = 'John'
AND employee.hire_date BETWEEN '2018-01-01' AND '2018-06-01'
```

Without condition filtering, `EXPLAIN` produces output like this:

id	table	type	possible_keys	key	ref	rows	filtered
1	employee	ref	name,h_date,dept	name	const	8	100.00
1	department	eq_ref	PRIMARY	PRIMARY	dept_no	1	100.00

For `employee`, the access method on the `name` index picks up the 8 rows that match a name of '`John`'. No filtering is done (`filtered` is 100%), so all rows are prefix rows for the next table: The prefix row count is `rows × filtered` =  $8 \times 100\% = 8$ .

With condition filtering, the optimizer additionally takes into account conditions from the `WHERE` clause not taken into account by the access method. In this case, the optimizer uses heuristics to estimate a filtering effect of 16.31% for the `BETWEEN` condition on `employee.hire_date`. As a result, `EXPLAIN` produces output like this:

id	table	type	possible_keys	key	ref	rows	filtered
1	employee	ref	name,h_date,dept	name	const	8	16.31
1	department	eq_ref	PRIMARY	PRIMARY	dept_no	1	100.00

Now the prefix row count is `rows × filtered` =  $8 \times 16.31\% = 1.3$ , which more closely reflects actual data set.

Normally, the optimizer does not calculate the condition filtering effect (prefix row count reduction) for the last joined table because there is no next table to pass rows to. An exception occurs for `EXPLAIN`: To provide more information, the filtering effect is calculated for all joined tables, including the last one.

To control whether the optimizer considers additional filtering conditions, use the `condition_fanout_filter` flag of the `optimizer_switch` system variable (see [Section 8.9.2, “Switchable Optimizations”](#)). This flag is enabled by default but can be disabled to suppress condition filtering (for example, if a particular query is found to yield better performance without it).

If the optimizer overestimates the effect of condition filtering, performance may be worse than if condition filtering is not used. In such cases, these techniques may help:

- If a column is not indexed, index it so that the optimizer has some information about the distribution of column values and can improve its row estimates.
- Similarly, if no column histogram information is available, generate a histogram (see [Section 8.9.6, “Optimizer Statistics”](#)).
- Change the join order. Ways to accomplish this include join-order optimizer hints (see [Section 8.9.3, “Optimizer Hints”](#)), `STRAIGHT_JOIN` immediately following the `SELECT`, and the `STRAIGHT_JOIN` join operator.
- Disable condition filtering for the session:

```
SET optimizer_switch = 'condition_fanout_filter=off';
```

Or, for a given query, using an optimizer hint:

```
SELECT /*+ SET_VAR(optimizer_switch = 'condition_fanout_filter=off') */ ...
```

### 8.2.1.13 IS NULL Optimization

MySQL can perform the same optimization on `col_name IS NULL` that it can use for `col_name = constant_value`. For example, MySQL can use indexes and ranges to search for `NULL` with `IS NULL`.

Examples:

```
SELECT * FROM tbl_name WHERE key_col IS NULL;
SELECT * FROM tbl_name WHERE key_col <=> NULL;
SELECT * FROM tbl_name
  WHERE key_col=const1 OR key_col=const2 OR key_col IS NULL;
```

If a `WHERE` clause includes a `col_name IS NULL` condition for a column that is declared as `NOT NULL`, that expression is optimized away. This optimization does not occur in cases when the column might produce `NULL` anyway; for example, if it comes from a table on the right side of a `LEFT JOIN`.

MySQL can also optimize the combination `col_name = expr OR col_name IS NULL`, a form that is common in resolved subqueries. `EXPLAIN` shows `ref_or_null` when this optimization is used.

This optimization can handle one `IS NULL` for any key part.

Some examples of queries that are optimized, assuming that there is an index on columns `a` and `b` of table `t2`:

```
SELECT * FROM t1 WHERE t1.a=expr OR t1.a IS NULL;
SELECT * FROM t1, t2 WHERE t1.a=t2.a OR t2.a IS NULL;
SELECT * FROM t1, t2
  WHERE (t1.a=t2.a OR t2.a IS NULL) AND t2.b=t1.b;
SELECT * FROM t1, t2
  WHERE t1.a=t2.a AND (t2.b=t1.b OR t2.b IS NULL);
SELECT * FROM t1, t2
  WHERE (t1.a=t2.a AND t2.a IS NULL AND ...)
    OR (t1.a=t2.a AND t2.a IS NULL AND ...);
```

`ref_or_null` works by first doing a read on the reference key, and then a separate search for rows with a `NULL` key value.

The optimization can handle only one `IS NULL` level. In the following query, MySQL uses key lookups only on the expression `(t1.a=t2.a AND t2.a IS NULL)` and is not able to use the key part on `b`:

```
SELECT * FROM t1, t2
  WHERE (t1.a=t2.a AND t2.a IS NULL)
    OR (t1.b=t2.b AND t2.b IS NULL);
```

### 8.2.1.14 ORDER BY Optimization

This section describes when MySQL can use an index to satisfy an `ORDER BY` clause, the `filesort` operation used when an index cannot be used, and execution plan information available from the optimizer about `ORDER BY`.

An `ORDER BY` with and without `LIMIT` may return rows in different orders, as discussed in Section 8.2.1.17, “[LIMIT Query Optimization](#)”.

- Use of Indexes to Satisfy ORDER BY
- Use of filesort to Satisfy ORDER BY
- Influencing ORDER BY Optimization
- ORDER BY Execution Plan Information Available

## Use of Indexes to Satisfy ORDER BY

In some cases, MySQL may use an index to satisfy an `ORDER BY` clause and avoid the extra sorting involved in performing a `filesort` operation.

The index may also be used even if the `ORDER BY` does not match the index exactly, as long as all unused portions of the index and all extra `ORDER BY` columns are constants in the `WHERE` clause. If the index does not contain all columns accessed by the query, the index is used only if index access is cheaper than other access methods.

Assuming that there is an index on `(key_part1, key_part2)`, the following queries may use the index to resolve the `ORDER BY` part. Whether the optimizer actually does so depends on whether reading the index is more efficient than a table scan if columns not in the index must also be read.

- In this query, the index on `(key_part1, key_part2)` enables the optimizer to avoid sorting:

```
SELECT * FROM t1
  ORDER BY key_part1, key_part2;
```

However, the query uses `SELECT *`, which may select more columns than `key_part1` and `key_part2`. In that case, scanning an entire index and looking up table rows to find columns not in the index may be more expensive than scanning the table and sorting the results. If so, the optimizer probably will not use the index. If `SELECT *` selects only the index columns, the index will be used and sorting avoided.

If `t1` is an `InnoDB` table, the table primary key is implicitly part of the index, and the index can be used to resolve the `ORDER BY` for this query:

```
SELECT pk, key_part1, key_part2 FROM t1
  ORDER BY key_part1, key_part2;
```

- In this query, `key_part1` is constant, so all rows accessed through the index are in `key_part2` order, and an index on `(key_part1, key_part2)` avoids sorting if the `WHERE` clause is selective enough to make an index range scan cheaper than a table scan:

```
SELECT * FROM t1
  WHERE key_part1 = constant
  ORDER BY key_part2;
```

- In the next two queries, whether the index is used is similar to the same queries without `DESC` shown previously:

```
SELECT * FROM t1
  ORDER BY key_part1 DESC, key_part2 DESC;

SELECT * FROM t1
  WHERE key_part1 = constant
  ORDER BY key_part2 DESC;
```

- Two columns in an `ORDER BY` can sort in the same direction (both `ASC`, or both `DESC`) or in opposite directions (one `ASC`, one `DESC`). A condition for index use is that the index must have the same homogeneity, but need not have the same actual direction.

If a query mixes `ASC` and `DESC`, the optimizer can use an index on the columns if the index also uses corresponding mixed ascending and descending columns:

```
SELECT * FROM t1
  ORDER BY key_part1 DESC, key_part2 ASC;
```

The optimizer can use an index on `(key_part1, key_part2)` if `key_part1` is descending and `key_part2` is ascending. It can also use an index on those columns (with a backward scan) if `key_part1` is ascending and `key_part2` is descending. See [Section 8.3.13, “Descending Indexes”](#)

- In the next two queries, `key_part1` is compared to a constant. The index will be used if the `WHERE` clause is selective enough to make an index range scan cheaper than a table scan:

```
SELECT * FROM t1
  WHERE key_part1 > constant
  ORDER BY key_part1 ASC;

SELECT * FROM t1
  WHERE key_part1 < constant
  ORDER BY key_part1 DESC;
```

- In the next query, the `ORDER BY` does not name `key_part1`, but all rows selected have a constant `key_part1` value, so the index can still be used:

```
SELECT * FROM t1
  WHERE key_part1 = constant1 AND key_part2 > constant2
  ORDER BY key_part2;
```

In some cases, MySQL *cannot* use indexes to resolve the `ORDER BY`, although it may still use indexes to find the rows that match the `WHERE` clause. Examples:

- The query uses `ORDER BY` on different indexes:

```
SELECT * FROM t1 ORDER BY key1, key2;
```

- The query uses `ORDER BY` on nonconsecutive parts of an index:

```
SELECT * FROM t1 WHERE key2=constant ORDER BY key1_part1, key1_part3;
```

- The index used to fetch the rows differs from the one used in the `ORDER BY`:

```
SELECT * FROM t1 WHERE key2=constant ORDER BY key1;
```

- The query uses `ORDER BY` with an expression that includes terms other than the index column name:

```
SELECT * FROM t1 ORDER BY ABS(key);
SELECT * FROM t1 ORDER BY -key;
```

- The query joins many tables, and the columns in the `ORDER BY` are not all from the first nonconstant table that is used to retrieve rows. (This is the first table in the `EXPLAIN` output that does not have a `const` join type.)

- The query has different `ORDER BY` and `GROUP BY` expressions.
- There is an index on only a prefix of a column named in the `ORDER BY` clause. In this case, the index cannot be used to fully resolve the sort order. For example, if only the first 10 bytes of a `CHAR(20)` column are indexed, the index cannot distinguish values past the 10th byte and a `filesort` is needed.
- The index does not store rows in order. For example, this is true for a `HASH` index in a `MEMORY` table.

Availability of an index for sorting may be affected by the use of column aliases. Suppose that the column `t1.a` is indexed. In this statement, the name of the column in the select list is `a`. It refers to `t1.a`, as does the reference to `a` in the `ORDER BY`, so the index on `t1.a` can be used:

```
SELECT a FROM t1 ORDER BY a;
```

In this statement, the name of the column in the select list is also `a`, but it is the alias name. It refers to `ABS(a)`, as does the reference to `a` in the `ORDER BY`, so the index on `t1.a` cannot be used:

```
SELECT ABS(a) AS a FROM t1 ORDER BY a;
```

In the following statement, the `ORDER BY` refers to a name that is not the name of a column in the select list. But there is a column in `t1` named `a`, so the `ORDER BY` refers to `t1.a` and the index on `t1.a` can be used. (The resulting sort order may be completely different from the order for `ABS(a)`, of course.)

```
SELECT ABS(a) AS b FROM t1 ORDER BY a;
```

Previously (MySQL 5.7 and lower), `GROUP BY` sorted implicitly under certain conditions. In MySQL 8.0, that no longer occurs, so specifying `ORDER BY NULL` at the end to suppress implicit sorting (as was done previously) is no longer necessary. However, query results may differ from previous MySQL versions. To produce a given sort order, provide an `ORDER BY` clause.

## Use of `filesort` to Satisfy `ORDER BY`

If an index cannot be used to satisfy an `ORDER BY` clause, MySQL performs a `filesort` operation that reads table rows and sorts them. A `filesort` constitutes an extra sorting phase in query execution.

To obtain memory for `filesort` operations, as of MySQL 8.0.12, the optimizer allocates memory buffers incrementally as needed, up to the size indicated by the `sort_buffer_size` system variable, rather than allocating a fixed amount of `sort_buffer_size` bytes up front, as was done prior to MySQL 8.0.12. This enables users to set `sort_buffer_size` to larger values to speed up larger sorts, without concern for excessive memory use for small sorts. (This benefit may not occur for multiple concurrent sorts on Windows, which has a weak multithreaded `malloc`.)

A `filesort` operation uses temporary disk files as necessary if the result set is too large to fit in memory. Some types of queries are particularly suited to completely in-memory `filesort` operations. For example, the optimizer can use `filesort` to efficiently handle in memory, without temporary files, the `ORDER BY` operation for queries (and subqueries) of the following form:

```
SELECT ... FROM single_table ... ORDER BY non_index_column [DESC] LIMIT [M,]N;
```

Such queries are common in web applications that display only a few rows from a larger result set. Examples:

```
SELECT col1, ... FROM t1 ... ORDER BY name LIMIT 10;
SELECT col1, ... FROM t1 ... ORDER BY RAND() LIMIT 15;
```

## Influencing ORDER BY Optimization

For slow `ORDER BY` queries for which `filesort` is not used, try lowering the `max_length_for_sort_data` system variable to a value that is appropriate to trigger a `filesort`. (A symptom of setting the value of this variable too high is a combination of high disk activity and low CPU activity.)

To increase `ORDER BY` speed, check whether you can get MySQL to use indexes rather than an extra sorting phase. If this is not possible, try the following strategies:

- Increase the `sort_buffer_size` variable value. Ideally, the value should be large enough for the entire result set to fit in the sort buffer (to avoid writes to disk and merge passes).

Take into account that the size of column values stored in the sort buffer is affected by the `max_sort_length` system variable value. For example, if tuples store values of long string columns and you increase the value of `max_sort_length`, the size of sort buffer tuples increases as well and may require you to increase `sort_buffer_size`.

To monitor the number of merge passes (to merge temporary files), check the `Sort_merge_passes` status variable.

- Increase the `read_rnd_buffer_size` variable value so that more rows are read at a time.
- Change the `tmpdir` system variable to point to a dedicated file system with large amounts of free space. The variable value can list several paths that are used in round-robin fashion; you can use this feature to spread the load across several directories. Separate the paths by colon characters (:) on Unix and semicolon characters (;) on Windows. The paths should name directories in file systems located on different *physical* disks, not different partitions on the same disk.

## ORDER BY Execution Plan Information Available

With `EXPLAIN` (see [Section 8.8.1, “Optimizing Queries with EXPLAIN”](#)), you can check whether MySQL can use indexes to resolve an `ORDER BY` clause:

- If the `Extra` column of `EXPLAIN` output does not contain `Using filesort`, the index is used and a `filesort` is not performed.
- If the `Extra` column of `EXPLAIN` output contains `Using filesort`, the index is not used and a `filesort` is performed.

In addition, if a `filesort` is performed, optimizer trace output includes a `filesort_summary` block. For example:

```
"filesort_summary": {
  "rows": 100,
  "examined_rows": 100,
  "number_of_tmp_files": 0,
  "peak_memory_used": 25192,
  "sort_mode": "<sort_key, packed_additional_fields>"
}
```

`peak_memory_used` indicates the maximum memory used at any one time during the sort. This is a value up to but not necessarily as large as the value of the `sort_buffer_size` system variable. Prior to MySQL 8.0.12, the output shows `sort_buffer_size` instead, indicating the value of

`sort_buffer_size`. (Prior to MySQL 8.0.12, the optimizer always allocates `sort_buffer_size` bytes for the sort buffer. As of 8.0.12, the optimizer allocates sort-buffer memory incrementally, beginning with a small amount and adding more as necessary, up to `sort_buffer_size` bytes.)

The `sort_mode` value provides information about the contents of tuples in the sort buffer:

- `<sort_key, rowid>`: This indicates that sort buffer tuples are pairs that contain the sort key value and row ID of the original table row. Tuples are sorted by sort key value and the row ID is used to read the row from the table.
- `<sort_key, additional_fields>`: This indicates that sort buffer tuples contain the sort key value and columns referenced by the query. Tuples are sorted by sort key value and column values are read directly from the tuple.
- `<sort_key, packed_additional_fields>`: Like the previous variant, but the additional columns are packed tightly together instead of using a fixed-length encoding.

`EXPLAIN` does not distinguish whether the optimizer does or does not perform a `filesort` in memory. Use of an in-memory `filesort` can be seen in optimizer trace output. Look for `filesort_priority_queue_optimization`. For information about the optimizer trace, see [MySQL Internals: Tracing the Optimizer](#).

### 8.2.1.15 GROUP BY Optimization

The most general way to satisfy a `GROUP BY` clause is to scan the whole table and create a new temporary table where all rows from each group are consecutive, and then use this temporary table to discover groups and apply aggregate functions (if any). In some cases, MySQL is able to do much better than that and avoid creation of temporary tables by using index access.

The most important preconditions for using indexes for `GROUP BY` are that all `GROUP BY` columns reference attributes from the same index, and that the index stores its keys in order (as is true, for example, for a `BTREE` index, but not for a `HASH` index). Whether use of temporary tables can be replaced by index access also depends on which parts of an index are used in a query, the conditions specified for these parts, and the selected aggregate functions.

There are two ways to execute a `GROUP BY` query through index access, as detailed in the following sections. The first method applies the grouping operation together with all range predicates (if any). The second method first performs a range scan, and then groups the resulting tuples.

- [Loose Index Scan](#)
- [Tight Index Scan](#)

Loose Index Scan can also be used in the absence of `GROUP BY` under some conditions. See [Skip Scan Range Access Method](#).

#### Loose Index Scan

The most efficient way to process `GROUP BY` is when an index is used to directly retrieve the grouping columns. With this access method, MySQL uses the property of some index types that the keys are ordered (for example, `BTREE`). This property enables use of lookup groups in an index without having to consider all keys in the index that satisfy all `WHERE` conditions. This access method considers only a fraction of the keys in an index, so it is called a *Loose Index Scan*. When there is no `WHERE` clause, a Loose Index Scan reads as many keys as the number of groups, which may be a much smaller number than that of all keys. If the `WHERE` clause contains range predicates (see the discussion of the `range` join type in [Section 8.8.1, “Optimizing Queries with EXPLAIN”](#)), a Loose Index Scan looks up the first key of

each group that satisfies the range conditions, and again reads the smallest possible number of keys. This is possible under the following conditions:

- The query is over a single table.
- The `GROUP BY` names only columns that form a leftmost prefix of the index and no other columns. (If, instead of `GROUP BY`, the query has a `DISTINCT` clause, all distinct attributes refer to columns that form a leftmost prefix of the index.) For example, if a table `t1` has an index on `(c1, c2, c3)`, Loose Index Scan is applicable if the query has `GROUP BY c1, c2`. It is not applicable if the query has `GROUP BY c2, c3` (the columns are not a leftmost prefix) or `GROUP BY c1, c2, c4` (`c4` is not in the index).
- The only aggregate functions used in the select list (if any) are `MIN()` and `MAX()`, and all of them refer to the same column. The column must be in the index and must immediately follow the columns in the `GROUP BY`.
- Any other parts of the index than those from the `GROUP BY` referenced in the query must be constants (that is, they must be referenced in equalities with constants), except for the argument of `MIN()` or `MAX()` functions.
- For columns in the index, full column values must be indexed, not just a prefix. For example, with `c1 VARCHAR(20), INDEX (c1(10))`, the index uses only a prefix of `c1` values and cannot be used for Loose Index Scan.

If Loose Index Scan is applicable to a query, the `EXPLAIN` output shows `Using index for group-by` in the `Extra` column.

Assume that there is an index `idx(c1, c2, c3)` on table `t1(c1, c2, c3, c4)`. The Loose Index Scan access method can be used for the following queries:

```
SELECT c1, c2 FROM t1 GROUP BY c1, c2;
SELECT DISTINCT c1, c2 FROM t1;
SELECT c1, MIN(c2) FROM t1 GROUP BY c1;
SELECT c1, c2 FROM t1 WHERE c1 < const GROUP BY c1, c2;
SELECT MAX(c3), MIN(c3), c1, c2 FROM t1 WHERE c2 > const GROUP BY c1, c2;
SELECT c2 FROM t1 WHERE c1 < const GROUP BY c1, c2;
SELECT c1, c2 FROM t1 WHERE c3 = const GROUP BY c1, c2;
```

The following queries cannot be executed with this quick select method, for the reasons given:

- There are aggregate functions other than `MIN()` or `MAX()`:

```
SELECT c1, SUM(c2) FROM t1 GROUP BY c1;
```

- The columns in the `GROUP BY` clause do not form a leftmost prefix of the index:

```
SELECT c1, c2 FROM t1 GROUP BY c2, c3;
```

- The query refers to a part of a key that comes after the `GROUP BY` part, and for which there is no equality with a constant:

```
SELECT c1, c3 FROM t1 GROUP BY c1, c2;
```

Were the query to include `WHERE c3 = const`, Loose Index Scan could be used.

The Loose Index Scan access method can be applied to other forms of aggregate function references in the select list, in addition to the `MIN()` and `MAX()` references already supported:

- `AVG(DISTINCT)`, `SUM(DISTINCT)`, and `COUNT(DISTINCT)` are supported. `AVG(DISTINCT)` and `SUM(DISTINCT)` take a single argument. `COUNT(DISTINCT)` can have more than one column argument.
- There must be no `GROUP BY` or `DISTINCT` clause in the query.
- The Loose Index Scan limitations described previously still apply.

Assume that there is an index `idx(c1,c2,c3)` on table `t1(c1,c2,c3,c4)`. The Loose Index Scan access method can be used for the following queries:

```
SELECT COUNT(DISTINCT c1), SUM(DISTINCT c1) FROM t1;
SELECT COUNT(DISTINCT c1, c2), COUNT(DISTINCT c2, c1) FROM t1;
```

## Tight Index Scan

A Tight Index Scan may be either a full index scan or a range index scan, depending on the query conditions.

When the conditions for a Loose Index Scan are not met, it still may be possible to avoid creation of temporary tables for `GROUP BY` queries. If there are range conditions in the `WHERE` clause, this method reads only the keys that satisfy these conditions. Otherwise, it performs an index scan. Because this method reads all keys in each range defined by the `WHERE` clause, or scans the whole index if there are no range conditions, it is called a *Tight Index Scan*. With a Tight Index Scan, the grouping operation is performed only after all keys that satisfy the range conditions have been found.

For this method to work, it is sufficient that there be a constant equality condition for all columns in a query referring to parts of the key coming before or in between parts of the `GROUP BY` key. The constants from the equality conditions fill in any “gaps” in the search keys so that it is possible to form complete prefixes of the index. These index prefixes then can be used for index lookups. If the `GROUP BY` result requires sorting, and it is possible to form search keys that are prefixes of the index, MySQL also avoids extra sorting operations because searching with prefixes in an ordered index already retrieves all the keys in order.

Assume that there is an index `idx(c1,c2,c3)` on table `t1(c1,c2,c3,c4)`. The following queries do not work with the Loose Index Scan access method described previously, but still work with the Tight Index Scan access method.

- There is a gap in the `GROUP BY`, but it is covered by the condition `c2 = 'a'`:

```
SELECT c1, c2, c3 FROM t1 WHERE c2 = 'a' GROUP BY c1, c3;
```

- The `GROUP BY` does not begin with the first part of the key, but there is a condition that provides a constant for that part:

```
SELECT c1, c2, c3 FROM t1 WHERE c1 = 'a' GROUP BY c2, c3;
```

### 8.2.1.16 DISTINCT Optimization

`DISTINCT` combined with `ORDER BY` needs a temporary table in many cases.

Because `DISTINCT` may use `GROUP BY`, learn how MySQL works with columns in `ORDER BY` or `HAVING` clauses that are not part of the selected columns. See [Section 12.20.3, “MySQL Handling of GROUP BY”](#).

In most cases, a `DISTINCT` clause can be considered as a special case of `GROUP BY`. For example, the following two queries are equivalent:

```
SELECT DISTINCT c1, c2, c3 FROM t1
WHERE c1 > const;  

SELECT c1, c2, c3 FROM t1
WHERE c1 > const GROUP BY c1, c2, c3;
```

Due to this equivalence, the optimizations applicable to `GROUP BY` queries can be also applied to queries with a `DISTINCT` clause. Thus, for more details on the optimization possibilities for `DISTINCT` queries, see [Section 8.2.1.15, “GROUP BY Optimization”](#).

When combining `LIMIT row_count` with `DISTINCT`, MySQL stops as soon as it finds `row_count` unique rows.

If you do not use columns from all tables named in a query, MySQL stops scanning any unused tables as soon as it finds the first match. In the following case, assuming that `t1` is used before `t2` (which you can check with `EXPLAIN`), MySQL stops reading from `t2` (for any particular row in `t1`) when it finds the first row in `t2`:

```
SELECT DISTINCT t1.a FROM t1, t2 WHERE t1.a=t2.a;
```

### 8.2.1.17 LIMIT Query Optimization

If you need only a specified number of rows from a result set, use a `LIMIT` clause in the query, rather than fetching the whole result set and throwing away the extra data.

MySQL sometimes optimizes a query that has a `LIMIT row_count` clause and no `HAVING` clause:

- If you select only a few rows with `LIMIT`, MySQL uses indexes in some cases when normally it would prefer to do a full table scan.
- If you combine `LIMIT row_count` with `ORDER BY`, MySQL stops sorting as soon as it has found the first `row_count` rows of the sorted result, rather than sorting the entire result. If ordering is done by using an index, this is very fast. If a filesort must be done, all rows that match the query without the `LIMIT` clause are selected, and most or all of them are sorted, before the first `row_count` are found. After the initial rows have been found, MySQL does not sort any remainder of the result set.

One manifestation of this behavior is that an `ORDER BY` query with and without `LIMIT` may return rows in different order, as described later in this section.

- If you combine `LIMIT row_count` with `DISTINCT`, MySQL stops as soon as it finds `row_count` unique rows.
- In some cases, a `GROUP BY` can be resolved by reading the index in order (or doing a sort on the index), then calculating summaries until the index value changes. In this case, `LIMIT row_count` does not calculate any unnecessary `GROUP BY` values.
- As soon as MySQL has sent the required number of rows to the client, it aborts the query unless you are using `SQL_CALC_FOUND_ROWS`. In that case, the number of rows can be retrieved with `SELECT FOUND_ROWS()`. See [Section 12.15, “Information Functions”](#).
- `LIMIT 0` quickly returns an empty set. This can be useful for checking the validity of a query. It can also be employed to obtain the types of the result columns within applications that use a MySQL API that makes result set metadata available. With the `mysql` client program, you can use the `--column-type-info` option to display result column types.

- If the server uses temporary tables to resolve a query, it uses the `LIMIT row_count` clause to calculate how much space is required.
- If an index is not used for `ORDER BY` but a `LIMIT` clause is also present, the optimizer may be able to avoid using a merge file and sort the rows in memory using an in-memory `filesort` operation.

If multiple rows have identical values in the `ORDER BY` columns, the server is free to return those rows in any order, and may do so differently depending on the overall execution plan. In other words, the sort order of those rows is nondeterministic with respect to the nonordered columns.

One factor that affects the execution plan is `LIMIT`, so an `ORDER BY` query with and without `LIMIT` may return rows in different orders. Consider this query, which is sorted by the `category` column but nondeterministic with respect to the `id` and `rating` columns:

```
mysql> SELECT * FROM ratings ORDER BY category;
+----+-----+-----+
| id | category | rating |
+----+-----+-----+
| 1  |       1 |   4.5 |
| 5  |       1 |   3.2 |
| 3  |       2 |   3.7 |
| 4  |       2 |   3.5 |
| 6  |       2 |   3.5 |
| 2  |       3 |   5.0 |
| 7  |       3 |   2.7 |
+----+-----+-----+
```

Including `LIMIT` may affect order of rows within each `category` value. For example, this is a valid query result:

```
mysql> SELECT * FROM ratings ORDER BY category LIMIT 5;
+----+-----+-----+
| id | category | rating |
+----+-----+-----+
| 1  |       1 |   4.5 |
| 5  |       1 |   3.2 |
| 4  |       2 |   3.5 |
| 3  |       2 |   3.7 |
| 6  |       2 |   3.5 |
+----+-----+-----+
```

In each case, the rows are sorted by the `ORDER BY` column, which is all that is required by the SQL standard.

If it is important to ensure the same row order with and without `LIMIT`, include additional columns in the `ORDER BY` clause to make the order deterministic. For example, if `id` values are unique, you can make rows for a given `category` value appear in `id` order by sorting like this:

```
mysql> SELECT * FROM ratings ORDER BY category, id;
+----+-----+-----+
| id | category | rating |
+----+-----+-----+
| 1  |       1 |   4.5 |
| 5  |       1 |   3.2 |
| 3  |       2 |   3.7 |
| 4  |       2 |   3.5 |
| 6  |       2 |   3.5 |
| 2  |       3 |   5.0 |
| 7  |       3 |   2.7 |
+----+-----+-----+
```

```
mysql> SELECT * FROM ratings ORDER BY category, id LIMIT 5;
+----+-----+-----+
| id | category | rating |
+----+-----+-----+
| 1  | 1        | 4.5   |
| 5  | 1        | 3.2   |
| 3  | 2        | 3.7   |
| 4  | 2        | 3.5   |
| 6  | 2        | 3.5   |
+----+-----+-----+
```

### 8.2.1.18 Function Call Optimization

MySQL functions are tagged internally as deterministic or nondeterministic. A function is nondeterministic if, given fixed values for its arguments, it can return different results for different invocations. Examples of nondeterministic functions: `RAND()`, `UUID()`.

If a function is tagged nondeterministic, a reference to it in a `WHERE` clause is evaluated for every row (when selecting from one table) or combination of rows (when selecting from a multiple-table join).

MySQL also determines when to evaluate functions based on types of arguments, whether the arguments are table columns or constant values. A deterministic function that takes a table column as argument must be evaluated whenever that column changes value.

Nondeterministic functions may affect query performance. For example, some optimizations may not be available, or more locking might be required. The following discussion uses `RAND()` but applies to other nondeterministic functions as well.

Suppose that a table `t` has this definition:

```
CREATE TABLE t (id INT NOT NULL PRIMARY KEY, col_a VARCHAR(100));
```

Consider these two queries:

```
SELECT * FROM t WHERE id = POW(1,2);
SELECT * FROM t WHERE id = FLOOR(1 + RAND() * 49);
```

Both queries appear to use a primary key lookup because of the equality comparison against the primary key, but that is true only for the first of them:

- The first query always produces a maximum of one row because `POW()` with constant arguments is a constant value and is used for index lookup.
- The second query contains an expression that uses the nondeterministic function `RAND()`, which is not constant in the query but in fact has a new value for every row of table `t`. Consequently, the query reads every row of the table, evaluates the predicate for each row, and outputs all rows for which the primary key matches the random value. This might be zero, one, or multiple rows, depending on the `id` column values and the values in the `RAND()` sequence.

The effects of nondeterminism are not limited to `SELECT` statements. This `UPDATE` statement uses a nondeterministic function to select rows to be modified:

```
UPDATE t SET col_a = some_expr WHERE id = FLOOR(1 + RAND() * 49);
```

Presumably the intent is to update at most a single row for which the primary key matches the expression. However, it might update zero, one, or multiple rows, depending on the `id` column values and the values in the `RAND()` sequence.

The behavior just described has implications for performance and replication:

- Because a nondeterministic function does not produce a constant value, the optimizer cannot use strategies that might otherwise be applicable, such as index lookups. The result may be a table scan.
- `InnoDB` might escalate to a range-key lock rather than taking a single row lock for one matching row.
- Updates that do not execute deterministically are unsafe for replication.

The difficulties stem from the fact that the `RAND()` function is evaluated once for every row of the table. To avoid multiple function evaluations, use one of these techniques:

- Move the expression containing the nondeterministic function to a separate statement, saving the value in a variable. In the original statement, replace the expression with a reference to the variable, which the optimizer can treat as a constant value:

```
SET @keyval = FLOOR(1 + RAND() * 49);
UPDATE t SET col_a = some_expr WHERE id = @keyval;
```

- Assign the random value to a variable in a derived table. This technique causes the variable to be assigned a value, once, prior to its use in the comparison in the `WHERE` clause:

```
UPDATE /*+ NO_MERGE(dt) */ t, (SELECT FLOOR(1 + RAND() * 49) AS r) AS dt
SET col_a = some_expr WHERE id = dt.r;
```

As mentioned previously, a nondeterministic expression in the `WHERE` clause might prevent optimizations and result in a table scan. However, it may be possible to partially optimize the `WHERE` clause if other expressions are deterministic. For example:

```
SELECT * FROM t WHERE partial_key=5 AND some_column=RAND();
```

If the optimizer can use `partial_key` to reduce the set of rows selected, `RAND()` is executed fewer times, which diminishes the effect of nondeterminism on optimization.

### 8.2.1.19 Window Function Optimization

Window functions affect the strategies the optimizer considers:

- Derived table merging for a subquery is disabled if the subquery has window functions. The subquery is always materialized.
- Semi-joins are not applicable to window function optimization because semi-joins apply to subqueries in `WHERE` and `JOIN ... ON`, which cannot contain window functions.
- The optimizer processes multiple windows that have the same ordering requirements in sequence, so sorting can be skipped for windows following the first one.
- The optimizer makes no attempt to merge windows that could be evaluated in a single step; for example, when multiple `OVER` clauses contain identical window definitions. The workaround is to define the window in a `WINDOW` clause and refer to the window name in the `OVER` clauses.

An aggregate function not used as a window function is aggregated in the outermost possible query. For example, in this query, MySQL sees that `COUNT(t1.b)` is something that cannot exist in the outer query because of its placement in the `WHERE` clause:

```
SELECT * FROM t1 WHERE t1.a = (SELECT COUNT(t1.b) FROM t2);
```

Consequently, MySQL aggregates inside the subquery, treating `t1.b` as a constant and returning the count of rows of `t2`.

Replacing `WHERE` with `HAVING` results in an error:

```
mysql> SELECT * FROM t1 HAVING t1.a = (SELECT COUNT(t1.b) FROM t2);
ERROR 1140 (42000): In aggregated query without GROUP BY, expression #1
of SELECT list contains nonaggregated column 'test.t1.a'; this is
incompatible with sql_mode=only_full_group_by
```

The error occurs because `COUNT(t1.b)` can exist in the `HAVING`, and so makes the outer query aggregated.

Window functions (including aggregate functions used as window functions) do not have the preceding complexity. They always aggregate in the subquery where they are written, never in the outer query.

Window function evaluation may be affected by the value of the `windowing_use_high_precision` system variable, which determines whether to compute window operations without loss of precision. By default, `windowing_use_high_precision` is enabled.

For some moving frame aggregates, the inverse aggregate function can be applied to remove values from the aggregate. This can improve performance but possibly with a loss of precision. For example, adding a very small floating-point value to a very large value causes the very small value to be “hidden” by the large value. When inverting the large value later, the effect of the small value is lost.

Loss of precision due to inverse aggregation is a factor only for operations on floating-point (approximate-value) data types. For other types, inverse aggregation is safe; this includes `DECIMAL`, which permits a fractional part but is an exact-value type.

For faster execution, MySQL always uses inverse aggregation when it is safe:

- For floating-point values, inverse aggregation is not always safe and might result in loss of precision. The default is to avoid inverse aggregation, which is slower but preserves precision. If it is permissible to sacrifice safety for speed, `windowing_use_high_precision` can be disabled to permit inverse aggregation.
- For nonfloating-point data types, inverse aggregation is always safe and is used regardless of the `windowing_use_high_precision` value.
- `windowing_use_high_precision` has no effect on `MIN()` and `MAX()`, which do not use inverse aggregation in any case.

For evaluation of the variance functions `STDDEV_POP()`, `STDDEV_SAMP()`, `VAR_POP()`, `VAR_SAMP()`, and their synonyms, evaluation can occur in optimized mode or default mode. Optimized mode may produce slightly different results in the last significant digits. If such differences are permissible, `windowing_use_high_precision` can be disabled to permit optimized mode.

For `EXPLAIN`, windowing execution plan information is too extensive to display in traditional output format. To see windowing information, use `EXPLAIN FORMAT=JSON` and look for the `windowing` element.

### 8.2.1.20 Row Constructor Expression Optimization

Row constructors permit simultaneous comparisons of multiple values. For example, these two statements are semantically equivalent:

```
SELECT * FROM t1 WHERE (column1,column2) = (1,1);
SELECT * FROM t1 WHERE column1 = 1 AND column2 = 1;
```

In addition, the optimizer handles both expressions the same way.

The optimizer is less likely to use available indexes if the row constructor columns do not cover the prefix of an index. Consider the following table, which has a primary key on (`c1, c2, c3`):

```
CREATE TABLE t1 (
    c1 INT, c2 INT, c3 INT, c4 CHAR(100),
    PRIMARY KEY(c1,c2,c3)
);
```

In this query, the `WHERE` clause uses all columns in the index. However, the row constructor itself does not cover an index prefix, with the result that the optimizer uses only `c1` (`key_len=4`, the size of `c1`):

```
mysql> EXPLAIN SELECT * FROM t1
      WHERE c1=1 AND (c2,c3) > (1,1)\G
***** 1. row *****
      id: 1
  select_type: SIMPLE
        table: t1
    partitions: NULL
       type: ref
possible_keys: PRIMARY
         key: PRIMARY
      key_len: 4
        ref: const
       rows: 3
  filtered: 100.00
     Extra: Using where
```

In such cases, rewriting the row constructor expression using an equivalent nonconstructor expression may result in more complete index use. For the given query, the row constructor and equivalent nonconstructor expressions are:

```
(c2,c3) > (1,1)
c2 > 1 OR ((c2 = 1) AND (c3 > 1))
```

Rewriting the query to use the nonconstructor expression results in the optimizer using all three columns in the index (`key_len=12`):

```
mysql> EXPLAIN SELECT * FROM t1
      WHERE c1 = 1 AND (c2 > 1 OR ((c2 = 1) AND (c3 > 1)))\G
***** 1. row *****
      id: 1
  select_type: SIMPLE
        table: t1
    partitions: NULL
       type: range
possible_keys: PRIMARY
         key: PRIMARY
      key_len: 12
        ref: NULL
       rows: 3
  filtered: 100.00
     Extra: Using where
```

Thus, for better results, avoid mixing row constructors with `AND/OR` expressions. Use one or the other.

Under certain conditions, the optimizer can apply the range access method to `IN()` expressions that have row constructor arguments. See [Range Optimization of Row Constructor Expressions](#).

### 8.2.1.21 Avoiding Full Table Scans

The output from `EXPLAIN` shows `ALL` in the `type` column when MySQL uses a **full table scan** to resolve a query. This usually happens under the following conditions:

- The table is so small that it is faster to perform a table scan than to bother with a key lookup. This is common for tables with fewer than 10 rows and a short row length.
- There are no usable restrictions in the `ON` or `WHERE` clause for indexed columns.
- You are comparing indexed columns with constant values and MySQL has calculated (based on the index tree) that the constants cover too large a part of the table and that a table scan would be faster. See [Section 8.2.1.1, “WHERE Clause Optimization”](#).
- You are using a key with low cardinality (many rows match the key value) through another column. In this case, MySQL assumes that by using the key it probably will do many key lookups and that a table scan would be faster.

For small tables, a table scan often is appropriate and the performance impact is negligible. For large tables, try the following techniques to avoid having the optimizer incorrectly choose a table scan:

- Use `ANALYZE TABLE tbl_name` to update the key distributions for the scanned table. See [Section 13.7.3.1, “ANALYZE TABLE Syntax”](#).
- Use `FORCE INDEX` for the scanned table to tell MySQL that table scans are very expensive compared to using the given index:

```
SELECT * FROM t1, t2 FORCE INDEX (index_for_column)
  WHERE t1.col_name=t2.col_name;
```

See [Section 8.9.4, “Index Hints”](#).

- Start `mysqld` with the `--max-seeks-for-key=1000` option or use `SET max_seeks_for_key=1000` to tell the optimizer to assume that no key scan causes more than 1,000 key seeks. See [Section 5.1.8, “Server System Variables”](#).

## 8.2.2 Optimizing Subqueries, Derived Tables, View References, and Common Table Expressions

The MySQL query optimizer has different strategies available to evaluate subqueries:

- For `IN` (or `=ANY`) subqueries, the optimizer has these choices:
  - Semi-join
  - Materialization
  - `EXISTS` strategy
- For `NOT IN` (or `<>ALL`) subqueries, the optimizer has these choices:
  - Materialization
  - `EXISTS` strategy

For derived tables, the optimizer has these choices (which also apply to view references and common table expressions):

- Merge the derived table into the outer query block
- Materialize the derived table to an internal temporary table

The following discussion provides more information about the preceding optimization strategies.



#### Note

A limitation on `UPDATE` and `DELETE` statements that use a subquery to modify a single table is that the optimizer does not use semi-join or materialization subquery optimizations. As a workaround, try rewriting them as multiple-table `UPDATE` and `DELETE` statements that use a join rather than a subquery.

### 8.2.2.1 Optimizing Subqueries, Derived Tables, View References, and Common Table Expressions with Semi-Join Transformations

A semi-join is a preparation-time transformation that enables multiple execution strategies such as table pullout, duplicate weedout, first match, loose scan, and materialization. The optimizer uses semi-join strategies to improve subquery execution, as described in this section.

For an inner join between two tables, the join returns a row from one table as many times as there are matches in the other table. But for some questions, the only information that matters is whether there is a match, not the number of matches. Suppose that there are tables named `class` and `roster` that list classes in a course curriculum and class rosters (students enrolled in each class), respectively. To list the classes that actually have students enrolled, you could use this join:

```
SELECT class.class_num, class.class_name
FROM class INNER JOIN roster
WHERE class.class_num = roster.class_num;
```

However, the result lists each class once for each enrolled student. For the question being asked, this is unnecessary duplication of information.

Assuming that `class_num` is a primary key in the `class` table, duplicate suppression is possible by using `SELECT DISTINCT`, but it is inefficient to generate all matching rows first only to eliminate duplicates later.

The same duplicate-free result can be obtained by using a subquery:

```
SELECT class_num, class_name
FROM class
WHERE class_num IN (SELECT class_num FROM roster);
```

Here, the optimizer can recognize that the `IN` clause requires the subquery to return only one instance of each class number from the `roster` table. In this case, the query can use a *semi-join*; that is, an operation that returns only one instance of each row in `class` that is matched by rows in `roster`.

Outer join and inner join syntax is permitted in the outer query specification, and table references may be base tables, derived tables, view references, or common table expressions.

In MySQL, a subquery must satisfy these criteria to be handled as a semi-join:

- It must be an `IN` (or `=ANY`) subquery that appears at the top level of the `WHERE` or `ON` clause, possibly as a term in an `AND` expression. For example:

```
SELECT ...
FROM otl1, ...
```

```
WHERE (oe1, ...) IN (SELECT ie1, ... FROM it1, ... WHERE ...);
```

Here, `ot_i` and `it_i` represent tables in the outer and inner parts of the query, and `oe_i` and `ie_i` represent expressions that refer to columns in the outer and inner tables.

- It must be a single `SELECT` without `UNION` constructs.
- It must not contain a `GROUP BY` or `HAVING` clause.
- It must not be implicitly grouped (it must contain no aggregate functions).
- It must not have `ORDER BY` with `LIMIT`.
- The statement must not use the `STRAIGHT_JOIN` join type in the outer query.
- The `STRAIGHT_JOIN` modifier must not be present.
- The number of outer and inner tables together must be less than the maximum number of tables permitted in a join.

The subquery may be correlated or uncorrelated. `DISTINCT` is permitted, as is `LIMIT` unless `ORDER BY` is also used.

If a subquery meets the preceding criteria, MySQL converts it to a semi-join and makes a cost-based choice from these strategies:

- Convert the subquery to a join, or use table pullout and run the query as an inner join between subquery tables and outer tables. Table pullout pulls a table out from the subquery to the outer query.
- Duplicate Weedout: Run the semi-join as if it was a join and remove duplicate records using a temporary table.
- FirstMatch: When scanning the inner tables for row combinations and there are multiple instances of a given value group, choose one rather than returning them all. This "shortcuts" scanning and eliminates production of unnecessary rows.
- LooseScan: Scan a subquery table using an index that enables a single value to be chosen from each subquery's value group.
- Materialize the subquery into an indexed temporary table that is used to perform a join, where the index is used to remove duplicates. The index might also be used later for lookups when joining the temporary table with the outer tables; if not, the table is scanned. For more information about materialization, see [Section 8.2.2.2, “Optimizing Subqueries with Materialization”](#).

Each of these strategies can be enabled or disabled using the following `optimizer_switch` system variable flags:

- The `semijoin` flag controls whether semi-joins are used.
- If `semijoin` is enabled, the `firstmatch`, `loosescan`, `duplicateweedout`, and `materialization` flags enable finer control over the permitted semi-join strategies.
- If the `duplicateweedout` semi-join strategy is disabled, it is not used unless all other applicable strategies are also disabled.
- If `duplicateweedout` 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`.

These flags are enabled by default. See [Section 8.9.2, “Switchable Optimizations”](#).

The optimizer minimizes differences in handling of views and derived tables. This affects queries that use the `STRAIGHT_JOIN` modifier and a view with an `IN` subquery that can be converted to a semi-join. The following query illustrates this because the change in processing causes a change in transformation, and thus a different execution strategy:

```
CREATE VIEW v AS
SELECT *
FROM t1
WHERE a IN (SELECT b
             FROM t2);

SELECT STRAIGHT_JOIN *
FROM t3 JOIN v ON t3.x = v.a;
```

The optimizer first looks at the view and converts the `IN` subquery to a semi-join, then checks whether it is possible to merge the view into the outer query. Because the `STRAIGHT_JOIN` modifier in the outer query prevents semi-join, the optimizer refuses the merge, causing derived table evaluation using a materialized table.

`EXPLAIN` output indicates the use of semi-join strategies as follows:

- For extended `EXPLAIN` output, the text displayed by a following `SHOW WARNINGS` shows the rewritten query, which displays the semi-join structure. (See [Section 8.8.3, “Extended EXPLAIN Output Format”](#).) From this you can get an idea about which tables were pulled out of the semi-join. If a subquery was converted to a semi-join, you will see that the subquery predicate is gone and its tables and `WHERE` clause were merged into the outer query join list and `WHERE` clause.
- Temporary table use for Duplicate Weedout is indicated by `Start temporary` and `End temporary` in the `Extra` column. Tables that were not pulled out and are in the range of `EXPLAIN` output rows covered by `Start temporary` and `End temporary` have their `rowid` in the temporary table.
- `FirstMatch(tbl_name)` in the `Extra` column indicates join shortcutting.
- `LooseScan(m..n)` in the `Extra` column indicates use of the LooseScan strategy. `m` and `n` are key part numbers.
- Temporary table use for materialization is indicated by rows with a `select_type` value of `MATERIALIZED` and rows with a `table` value of `<subqueryN>`.

### 8.2.2.2 Optimizing Subqueries with Materialization

The optimizer uses materialization to enable more efficient subquery processing. Materialization speeds up query execution by generating a subquery result as a temporary table, normally in memory. The first time MySQL needs the subquery result, it materializes that result into a temporary table. Any subsequent time the result is needed, MySQL refers again to the temporary table. The optimizer may index the table with a hash index to make lookups fast and inexpensive. The index contains unique values to eliminate duplicates and make the table smaller.

Subquery materialization uses an in-memory temporary table when possible, falling back to on-disk storage if the table becomes too large. See [Section 8.4.4, “Internal Temporary Table Use in MySQL”](#).

If materialization is not used, the optimizer sometimes rewrites a noncorrelated subquery as a correlated subquery. For example, the following `IN` subquery is noncorrelated (`where_condition` involves only columns from `t2` and not `t1`):

```
SELECT * FROM t1
WHERE t1.a IN (SELECT t2.b FROM t2 WHERE where_condition);
```

The optimizer might rewrite this as an `EXISTS` correlated subquery:

```
SELECT * FROM t1
WHERE EXISTS (SELECT t2.b FROM t2 WHERE where_condition AND t1.a=t2.b);
```

Subquery materialization using a temporary table avoids such rewrites and makes it possible to execute the subquery only once rather than once per row of the outer query.

For subquery materialization to be used in MySQL, the `optimizer_switch` system variable `materialization` flag must be enabled. (See [Section 8.9.2, “Switchable Optimizations”](#).) With the `materialization` flag enabled, materialization applies to subquery predicates that appear anywhere (in the select list, `WHERE`, `ON`, `GROUP BY`, `HAVING`, or `ORDER BY`), for predicates that fall into any of these use cases:

- The predicate has this form, when no outer expression `oe_i` or inner expression `ie_i` is nullable. `N` is 1 or larger.

```
(oe_1, oe_2, ..., oe_N) [NOT] IN (SELECT ie_1, i_2, ..., ie_N ...)
```

- The predicate has this form, when there is a single outer expression `oe` and inner expression `ie`. The expressions can be nullable.

```
oe [NOT] IN (SELECT ie ...)
```

- The predicate is `IN` or `NOT IN` and a result of `UNKNOWN` (`NULL`) has the same meaning as a result of `FALSE`.

The following examples illustrate how the requirement for equivalence of `UNKNOWN` and `FALSE` predicate evaluation affects whether subquery materialization can be used. Assume that `where_condition` involves columns only from `t2` and not `t1` so that the subquery is noncorrelated.

This query is subject to materialization:

```
SELECT * FROM t1
WHERE t1.a IN (SELECT t2.b FROM t2 WHERE where_condition);
```

Here, it does not matter whether the `IN` predicate returns `UNKNOWN` or `FALSE`. Either way, the row from `t1` is not included in the query result.

An example where subquery materialization is not used is the following query, where `t2.b` is a nullable column:

```
SELECT * FROM t1
WHERE (t1.a,t1.b) NOT IN (SELECT t2.a,t2.b FROM t2
                           WHERE where_condition);
```

The following restrictions apply to the use of subquery materialization:

- The types of the inner and outer expressions must match. For example, the optimizer might be able to use materialization if both expressions are integer or both are decimal, but cannot if one expression is integer and the other is decimal.

- The inner expression cannot be a `BLOB`.

Use of `EXPLAIN` with a query provides some indication of whether the optimizer uses subquery materialization:

- Compared to query execution that does not use materialization, `select_type` may change from `DEPENDENT SUBQUERY` to `SUBQUERY`. This indicates that, for a subquery that would be executed once per outer row, materialization enables the subquery to be executed just once.
- For extended `EXPLAIN` output, the text displayed by a following `SHOW WARNINGS` includes `materialize` and `materialized-subquery`.

### 8.2.2.3 Optimizing Subqueries with the EXISTS Strategy

Certain optimizations are applicable to comparisons that use the `IN` (or `=ANY`) operator to test subquery results. This section discusses these optimizations, particularly with regard to the challenges that `NULL` values present. The last part of the discussion suggests how you can help the optimizer.

Consider the following subquery comparison:

```
outer_expr IN (SELECT inner_expr FROM ... WHERE subquery_where)
```

MySQL evaluates queries “from outside to inside.” That is, it first obtains the value of the outer expression `outer_expr`, and then runs the subquery and captures the rows that it produces.

A very useful optimization is to “inform” the subquery that the only rows of interest are those where the inner expression `inner_expr` is equal to `outer_expr`. This is done by pushing down an appropriate equality into the subquery’s `WHERE` clause to make it more restrictive. The converted comparison looks like this:

```
EXISTS (SELECT 1 FROM ... WHERE subquery_where AND outer_expr=inner_expr)
```

After the conversion, MySQL can use the pushed-down equality to limit the number of rows it must examine to evaluate the subquery.

More generally, a comparison of `N` values to a subquery that returns `N`-value rows is subject to the same conversion. If `oe_i` and `ie_i` represent corresponding outer and inner expression values, this subquery comparison:

```
(oe_1, ..., oe_N) IN
  (SELECT ie_1, ..., ie_N FROM ... WHERE subquery_where)
```

Becomes:

```
EXISTS (SELECT 1 FROM ... WHERE subquery_where
        AND oe_1 = ie_1
        AND ...
        AND oe_N = ie_N)
```

For simplicity, the following discussion assumes a single pair of outer and inner expression values.

The conversion just described has its limitations. It is valid only if we ignore possible `NULL` values. That is, the “pushdown” strategy works as long as both of these conditions are true:

- `outer_expr` and `inner_expr` cannot be `NULL`.

- You need not distinguish `NULL` from `FALSE` subquery results. If the subquery is a part of an `OR` or `AND` expression in the `WHERE` clause, MySQL assumes that you do not care. Another instance where the optimizer notices that `NULL` and `FALSE` subquery results need not be distinguished is this construct:

```
... WHERE outer_expr IN (subquery)
```

In this case, the `WHERE` clause rejects the row whether `IN (subquery)` returns `NULL` or `FALSE`.

When either or both of those conditions do not hold, optimization is more complex.

Suppose that `outer_expr` is known to be a non-`NULL` value but the subquery does not produce a row such that `outer_expr = inner_expr`. Then `outer_expr IN (SELECT ...)` evaluates as follows:

- `NULL`, if the `SELECT` produces any row where `inner_expr` is `NULL`
- `FALSE`, if the `SELECT` produces only non-`NULL` values or produces nothing

In this situation, the approach of looking for rows with `outer_expr = inner_expr` is no longer valid. It is necessary to look for such rows, but if none are found, also look for rows where `inner_expr` is `NULL`. Roughly speaking, the subquery can be converted to something like this:

```
EXISTS (SELECT 1 FROM ... WHERE subquery_where AND
        (outer_expr=inner_expr OR inner_expr IS NULL))
```

The need to evaluate the extra `IS NULL` condition is why MySQL has the `ref_or_null` access method:

```
mysql> EXPLAIN
      SELECT outer_expr IN (SELECT t2.maybe_null_key
                            FROM t2, t3 WHERE ...)
        FROM t1;
*****
1. row *****
    id: 1
  select_type: PRIMARY
    table: t1
...
*****
2. row *****
    id: 2
  select_type: DEPENDENT SUBQUERY
    table: t2
    type: ref_or_null
possible_keys: maybe_null_key
      key: maybe_null_key
    key_len: 5
      ref: func
    rows: 2
  Extra: Using where; Using index
...
```

The `unique_subquery` and `index_subquery` subquery-specific access methods also have “or `NULL`” variants.

The additional `OR ... IS NULL` condition makes query execution slightly more complicated (and some optimizations within the subquery become inapplicable), but generally this is tolerable.

The situation is much worse when `outer_expr` can be `NULL`. According to the SQL interpretation of `NULL` as “unknown value,” `NULL IN (SELECT inner_expr ...)` should evaluate to:

- `NULL`, if the `SELECT` produces any rows
- `FALSE`, if the `SELECT` produces no rows

For proper evaluation, it is necessary to be able to check whether the `SELECT` has produced any rows at all, so `outer_expr = inner_expr` cannot be pushed down into the subquery. This is a problem because many real world subqueries become very slow unless the equality can be pushed down.

Essentially, there must be different ways to execute the subquery depending on the value of `outer_expr`.

The optimizer chooses SQL compliance over speed, so it accounts for the possibility that `outer_expr` might be `NULL`:

- If `outer_expr` is `NULL`, to evaluate the following expression, it is necessary to execute the `SELECT` to determine whether it produces any rows:

```
NULL IN (SELECT inner_expr FROM ... WHERE subquery_where)
```

It is necessary to execute the original `SELECT` here, without any pushed-down equalities of the kind mentioned previously.

- On the other hand, when `outer_expr` is not `NULL`, it is absolutely essential that this comparison:

```
outer_expr IN (SELECT inner_expr FROM ... WHERE subquery_where)
```

Be converted to this expression that uses a pushed-down condition:

```
EXISTS (SELECT 1 FROM ... WHERE subquery_where AND outer_expr=inner_expr)
```

Without this conversion, subqueries will be slow.

To solve the dilemma of whether or not to push down conditions into the subquery, the conditions are wrapped within “trigger” functions. Thus, an expression of the following form:

```
outer_expr IN (SELECT inner_expr FROM ... WHERE subquery_where)
```

Is converted into:

```
EXISTS (SELECT 1 FROM ... WHERE subquery_where
        AND trigcond(outer_expr=inner_expr))
```

More generally, if the subquery comparison is based on several pairs of outer and inner expressions, the conversion takes this comparison:

```
(oe_1, ..., oe_N) IN (SELECT ie_1, ..., ie_N FROM ... WHERE subquery_where)
```

And converts it to this expression:

```
EXISTS (SELECT 1 FROM ... WHERE subquery_where
        AND trigcond(oe_1=ie_1)
        AND ...
        AND trigcond(oe_N=ie_N))
      )
```

Each `trigcond(X)` is a special function that evaluates to the following values:

- `X` when the “linked” outer expression `oe_i` is not `NULL`
- `TRUE` when the “linked” outer expression `oe_i` is `NULL`

**Note**

Trigger functions are *not* triggers of the kind that you create with `CREATE TRIGGER`.

Equalities that are wrapped within `trigcond()` functions are not first class predicates for the query optimizer. Most optimizations cannot deal with predicates that may be turned on and off at query execution time, so they assume any `trigcond(X)` to be an unknown function and ignore it. Triggered equalities can be used by those optimizations:

- Reference optimizations: `trigcond(X=Y [OR Y IS NULL])` can be used to construct `ref`, `eq_ref`, or `ref_or_null` table accesses.
- Index lookup-based subquery execution engines: `trigcond(X=Y)` can be used to construct `unique_subquery` or `index_subquery` accesses.
- Table-condition generator: If the subquery is a join of several tables, the triggered condition is checked as soon as possible.

When the optimizer uses a triggered condition to create some kind of index lookup-based access (as for the first two items of the preceding list), it must have a fallback strategy for the case when the condition is turned off. This fallback strategy is always the same: Do a full table scan. In `EXPLAIN` output, the fallback shows up as `Full scan on NULL key` in the `Extra` column:

```
mysql> EXPLAIN SELECT t1.col1,
    t1.col1 IN (SELECT t2.key1 FROM t2 WHERE t2.col2=t1.col2) FROM t1\G
*****
   1. row *****
      id: 1
      select_type: PRIMARY
      table: t1
      ...
*****
   2. row *****
      id: 2
      select_type: DEPENDENT SUBQUERY
      table: t2
      type: index_subquery
possible_keys: key1
      key: key1
      key_len: 5
      ref: func
      rows: 2
      Extra: Using where; Full scan on NULL key
```

If you run `EXPLAIN` followed by `SHOW WARNINGS`, you can see the triggered condition:

```
*****
   1. row *****
Level: Note
Code: 1003
Message: select `test`.`t1`.`col1` AS `col1`,
<in_optimizer>(`test`.`t1`.`col1`,
<exists>(<index_lookup>(<cache>(`test`.`t1`.`col1`) in t2
on key1 checking NULL
where (`test`.`t2`.`col2` = `test`.`t1`.`col2`) having
trigcond(<is_not_null_test>(`test`.`t2`.`key1`))) AS
`t1.col1 IN (select t2.key1 from t2 where t2.col2=t1.col2)`-
from `test`.`t1`-
```

The use of triggered conditions has some performance implications. A `NULL IN (SELECT ...)` expression now may cause a full table scan (which is slow) when it previously did not. This is the price paid for correct results (the goal of the trigger-condition strategy is to improve compliance, not speed).

For multiple-table subqueries, execution of `NULL IN (SELECT ...)` is particularly slow because the join optimizer does not optimize for the case where the outer expression is `NULL`. It assumes that subquery evaluations with `NULL` on the left side are very rare, even if there are statistics that indicate otherwise. On the other hand, if the outer expression might be `NULL` but never actually is, there is no performance penalty.

To help the query optimizer better execute your queries, use these suggestions:

- Declare a column as `NOT NULL` if it really is. This also helps other aspects of the optimizer by simplifying condition testing for the column.
- If you need not distinguish a `NULL` from `FALSE` subquery result, you can easily avoid the slow execution path. Replace a comparison that looks like this:

```
outer_expr IN (SELECT inner_expr FROM ...)
```

with this expression:

```
(outer_expr IS NOT NULL) AND (outer_expr IN (SELECT inner_expr FROM ...))
```

Then `NULL IN (SELECT ...)` is never evaluated because MySQL stops evaluating `AND` parts as soon as the expression result is clear.

Another possible rewrite:

```
EXISTS (SELECT inner_expr FROM ...
        WHERE inner_expr=outer_expr)
```

This would apply when you need not distinguish `NULL` from `FALSE` subquery results, in which case you may actually want `EXISTS`.

The `subquery_materialization_cost_based` flag of the `optimizer_switch` system variable enables control over the choice between subquery materialization and `IN`-to-`EXISTS` subquery transformation. See [Section 8.9.2, “Switchable Optimizations”](#).

#### 8.2.2.4 Optimizing Derived Tables, View References, and Common Table Expressions with Merging or Materialization

The optimizer can handle derived table references using two strategies (which also apply to view references and common table expressions):

- Merge the derived table into the outer query block
- Materialize the derived table to an internal temporary table

Example 1:

```
SELECT * FROM (SELECT * FROM t1) AS derived_t1;
```

With merging of the derived table `derived_t1`, that query is executed similar to:

```
SELECT * FROM t1;
```

Example 2:

```
SELECT *
  FROM t1 JOIN (SELECT t2.f1 FROM t2) AS derived_t2 ON t1.f2=derived_t2.f1
 WHERE t1.f1 > 0;
```

With merging of the derived table `derived_t2`, that query is executed similar to:

```
SELECT t1.* , t2.f1
  FROM t1 JOIN t2 ON t1.f2=t2.f1
 WHERE t1.f1 > 0;
```

With materialization, `derived_t1` and `derived_t2` are each treated as a separate table within their respective queries.

The optimizer handles derived tables, view references, and common table expressions the same way: It avoids unnecessary materialization whenever possible, which enables pushing down conditions from the outer query to derived tables and produces more efficient execution plans. (For an example, see [Section 8.2.2.2, “Optimizing Subqueries with Materialization”](#).)

If merging would result in an outer query block that references more than 61 base tables, the optimizer chooses materialization instead.

The optimizer propagates an `ORDER BY` clause in a derived table or view reference to the outer query block if these conditions are all true:

- The outer query is not grouped or aggregated.
- The outer query does not specify `DISTINCT`, `HAVING`, or `ORDER BY`.
- The outer query has this derived table or view reference as the only source in the `FROM` clause.

Otherwise, the optimizer ignores the `ORDER BY` clause.

The following means are available to influence whether the optimizer attempts to merge derived tables, view references, and common table expressions into the outer query block:

- The `MERGE` and `NO_MERGE` optimizer hints can be used. They apply assuming that no other rule prevents merging. See [Section 8.9.3, “Optimizer Hints”](#).
- Similarly, you can use the `derived_merge` flag of the `optimizer_switch` system variable. See [Section 8.9.2, “Switchable Optimizations”](#). By default, the flag is enabled to permit merging. Disabling the flag prevents merging and avoids `ER_UPDATE_TABLE_USED` errors.

The `derived_merge` flag also applies to views that contain no `ALGORITHM` clause. Thus, if an `ER_UPDATE_TABLE_USED` error occurs for a view reference that uses an expression equivalent to the subquery, adding `ALGORITHM=TEMPTABLE` to the view definition prevents merging and takes precedence over the `derived_merge` value.

- It is possible to disable merging by using in the subquery any constructs that prevent merging, although these are not as explicit in their effect on materialization. Constructs that prevent merging are the same for derived tables, common table expressions, and view references:

- Aggregate functions or window functions (`SUM()`, `MIN()`, `MAX()`, `COUNT()`, and so forth)
- `DISTINCT`
- `GROUP BY`

- `HAVING`
- `LIMIT`
- `UNION` or `UNION ALL`
- Subqueries in the select list
- Assignments to user variables
- References only to literal values (in this case, there is no underlying table)

If the optimizer chooses the materialization strategy rather than merging for a derived table, it handles the query as follows:

- The optimizer postpones derived table materialization until its contents are needed during query execution. This improves performance because delaying materialization may result in not having to do it at all. Consider a query that joins the result of a derived table to another table: If the optimizer processes that other table first and finds that it returns no rows, the join need not be carried out further and the optimizer can completely skip materializing the derived table.
- During query execution, the optimizer may add an index to a derived table to speed up row retrieval from it.

Consider the following `EXPLAIN` statement, for a `SELECT` query that contains a derived table:

```
EXPLAIN SELECT * FROM (SELECT * FROM t1) AS derived_t1;
```

The optimizer avoids materializing the derived table by delaying it until the result is needed during `SELECT` execution. In this case, the query is not executed (because it occurs in an `EXPLAIN` statement), so the result is never needed.

Even for queries that are executed, delay of derived table materialization may enable the optimizer to avoid materialization entirely. When this happens, query execution is quicker by the time needed to perform materialization. Consider the following query, which joins the result of a derived table to another table:

```
SELECT *
  FROM t1 JOIN (SELECT t2.f1 FROM t2) AS derived_t2
            ON t1.f2=derived_t2.f1
 WHERE t1.f1 > 0;
```

If the optimization processes `t1` first and the `WHERE` clause produces an empty result, the join must necessarily be empty and the derived table need not be materialized.

For cases when a derived table requires materialization, the optimizer may add an index to the materialized table to speed up access to it. If such an index enables `ref` access to the table, it can greatly reduce amount of data read during query execution. Consider the following query:

```
SELECT *
  FROM t1 JOIN (SELECT DISTINCT f1 FROM t2) AS derived_t2
            ON t1.f1=derived_t2.f1;
```

The optimizer constructs an index over column `f1` from `derived_t2` if doing so would enable use of `ref` access for the lowest cost execution plan. After adding the index, the optimizer can treat the materialized derived table the same as a regular table with an index, and it benefits similarly from the generated index.

The overhead of index creation is negligible compared to the cost of query execution without the index. If `ref` access would result in higher cost than some other access method, the optimizer creates no index and loses nothing.

For optimizer trace output, a merged derived table or view reference is not shown as a node. Only its underlying tables appear in the top query's plan.

What is true for materialization of derived tables is also true for common table expressions (CTEs). In addition, the following considerations pertain specifically to CTEs.

If a CTE is materialized by a query, it is materialized once for the query, even if the query references it several times.

A recursive CTE is always materialized.

If a CTE is materialized, the optimizer automatically adds relevant indexes if it estimates that indexing will speed up access by the top-level statement to the CTE. This is similar to automatic indexing of derived tables, except that if the CTE is referenced multiple times, the optimizer may create multiple indexes, to speed up access by each reference in the most appropriate way.

The `MERGE` and `NO_MERGE` optimizer hints can be applied to CTEs. Each CTE reference in the top-level statement can have its own hint, permitting CTE references to be selectively merged or materialized. The following statement uses hints to indicate that `cte1` should be merged and `cte2` should be materialized:

```
WITH
  cte1 AS (SELECT a, b FROM table1),
  cte2 AS (SELECT c, d FROM table2)
SELECT /*+ MERGE(cte1) NO_MERGE(cte2) */ cte1.b, cte2.d
FROM cte1 JOIN cte2
WHERE cte1.a = cte2.c;
```

The `ALGORITHM` clause for `CREATE VIEW` does not affect materialization for any `WITH` clause preceding the `SELECT` statement in the view definition. Consider this statement:

```
CREATE ALGORITHM={TEMPTABLE|MERGE} VIEW v1 AS WITH ... SELECT ...
```

The `ALGORITHM` value affects materialization only of the `SELECT`, not the `WITH` clause.

Prior to MySQL 8.0.16, if `internal_tmp_disk_storage_engine=MYISAM`, an error occurred for any attempt to materialize a CTE using an on-disk temporary table, since for CTEs, the storage engine used for on-disk internal temporary tables could not be `MyISAM`. Beginning with MySQL 8.0.16, this is no longer an issue, since `TempTable` now always uses `InnoDB` for on-disk internal temporary tables.

As mentioned previously, a CTE, if materialized, is materialized once, even if referenced multiple times. To indicate one-time materialization, optimizer trace output contains an occurrence of `creating_tmp_table` plus one or more occurrences of `reusing_tmp_table`.

CTEs are similar to derived tables, for which the `materialized_from_subquery` node follows the reference. This is true for a CTE that is referenced multiple times, so there is no duplication of `materialized_from_subquery` nodes (which would give the impression that the subquery is executed multiple times, and produce unnecessarily verbose output). Only one reference to the CTE has a complete `materialized_from_subquery` node with the description of its subquery plan. Other references have a reduced `materialized_from_subquery` node. The same idea applies to `EXPLAIN` output in `TRADITIONAL` format: Subqueries for other references are not shown.

## 8.2.3 Optimizing INFORMATION\_SCHEMA Queries

Applications that monitor databases may make frequent use of [INFORMATION\\_SCHEMA](#) tables. To write queries for these tables most efficiently, use the following general guidelines:

- Try to query only [INFORMATION\\_SCHEMA](#) tables that are views on data dictionary tables.
- Try to query only for static metadata. Selecting columns or using retrieval conditions for dynamic metadata along with static metadata adds overhead to process the dynamic metadata.

**Note**

Comparison behavior for database and table names in [INFORMATION\\_SCHEMA](#) queries might differ from what you expect. For details, see [Section 10.8.7, “Using Collation in INFORMATION\\_SCHEMA Searches”](#).

These [INFORMATION\\_SCHEMA](#) tables are implemented as views on data dictionary tables, so queries on them retrieve information from the data dictionary:

```
CHARACTER_SETS
CHECK_CONSTRAINTS
COLLATIONS
COLLATION_CHARACTER_SET_APPLICABILITY
COLUMNS
EVENTS
FILES
INNODB_COLUMNS
INNODB_DATAFILES
INNODB_FIELDS
INNODB_FOREIGN
INNODB_FOREIGN_COLS
INNODB_INDEXES
INNODB_TABLES
INNODB_TABLESPACES
INNODB_TABLESPACES_BRIEF
INNODB_TABLESTATS
KEY_COLUMN_USAGE
PARAMETERS
PARTITIONS
REFERENTIAL_CONSTRAINTS
RESOURCE_GROUPS
ROUTINES
SCHEMATA
STATISTICS
TABLES
TABLE_CONSTRAINTS
TRIGGERS
VIEWS
VIEW_ROUTINE_USAGE
VIEW_TABLE_USAGE
```

Some types of values, even for a non-view [INFORMATION\\_SCHEMA](#) table, are retrieved by lookups from the data dictionary. This includes values such as database and table names, table types, and storage engines.

Some [INFORMATION\\_SCHEMA](#) tables contain columns that provide table statistics:

```
STATISTICS.CARDINALITY
TABLES.AUTO_INCREMENT
TABLES.AVG_ROW_LENGTH
TABLES.CHECKSUM
TABLES.CHECK_TIME
TABLES.CREATE_TIME
```

```
TABLES . DATA_FREE
TABLES . DATA_LENGTH
TABLES . INDEX_LENGTH
TABLES . MAX_DATA_LENGTH
TABLES . TABLE_ROWS
TABLES . UPDATE_TIME
```

Those columns represent dynamic table metadata; that is, information that changes as table contents change.

By default, MySQL retrieves cached values for those columns from the `mysql.index_stats` and `mysql.table_stats` dictionary tables when the columns are queried, which is more efficient than retrieving statistics directly from the storage engine. If cached statistics are not available or have expired, MySQL retrieves the latest statistics from the storage engine and caches them in the `mysql.index_stats` and `mysql.table_stats` dictionary tables. Subsequent queries retrieve the cached statistics until the cached statistics expire.

The `information_schema_stats_expiry` session variable defines the period of time before cached statistics expire. The default is 86400 seconds (24 hours), but the time period can be extended to as much as one year.

To update cached values at any time for a given table, use `ANALYZE TABLE`.

Querying statistics columns does not store or update statistics in the `mysql.index_stats` and `mysql.table_stats` dictionary tables under these circumstances:

- When cached statistics have not expired.
- When `information_schema_stats_expiry` is set to 0.
- When the server is started in `read_only`, `super_read_only`, `transaction_read_only`, or `innodb_read_only` mode.
- When the query also fetches Performance Schema data.

`information_schema_stats_expiry` is a session variable, and each client session can define its own expiration value. Statistics that are retrieved from the storage engine and cached by one session are available to other sessions.



#### Note

If the `innodb_read_only` system variable is enabled, `ANALYZE TABLE` may fail because it cannot update statistics tables in the data dictionary, which use InnoDB. For `ANALYZE TABLE` operations that update the key distribution, failure may occur even if the operation updates the table itself (for example, if it is a MyISAM table). To obtain the updated distribution statistics, set `information_schema_stats_expiry=0`.

For `INFORMATION_SCHEMA` tables implemented as views on data dictionary tables, indexes on the underlying data dictionary tables permit the optimizer to construct efficient query execution plans. To see the choices made by the optimizer, use `EXPLAIN`. To also see the query used by the server to execute an `INFORMATION_SCHEMA` query, use `SHOW WARNINGS` immediately following `EXPLAIN`.

Consider this statement, which identifies collations for the `utf8mb4` character set:

```
mysql> SELECT COLLATION_NAME
   FROM INFORMATION_SCHEMA.COLLATION_CHARACTER_SET_APPLICABILITY
```

```
WHERE CHARACTER_SET_NAME = 'utf8mb4';
+-----+
| COLLATION_NAME |
+-----+
| utf8mb4_general_ci |
| utf8mb4_bin |
| utf8mb4_unicode_ci |
| utf8mb4_icelandic_ci |
| utf8mb4_latvian_ci |
| utf8mb4_romanian_ci |
| utf8mb4_slovenian_ci |
...
...
```

How does the server process that statement? To find out, use `EXPLAIN`:

```
mysql> EXPLAIN SELECT COLLATION_NAME
      FROM INFORMATION_SCHEMA.COLLATION_CHARACTER_SET_APPLICABILITY
     WHERE CHARACTER_SET_NAME = 'utf8mb4'\G
***** 1. row *****
    id: 1
  select_type: SIMPLE
        table: cs
    partitions: NULL
       type: const
possible_keys: PRIMARY,name
         key: name
      key_len: 194
        ref: const
       rows: 1
  filtered: 100.00
    Extra: Using index
***** 2. row *****
    id: 1
  select_type: SIMPLE
        table: col
    partitions: NULL
       type: ref
possible_keys: character_set_id
         key: character_set_id
      key_len: 8
        ref: const
       rows: 68
  filtered: 100.00
    Extra: NULL
2 rows in set, 1 warning (0.01 sec)
```

To see the query used to satisfy that statement, use `SHOW WARNINGS`:

```
mysql> SHOW WARNINGS\G
***** 1. row *****
  Level: Note
   Code: 1003
Message: /* select#1 */ select `mysql`.`col`.`name` AS `COLLATION_NAME`
      from `mysql`.`character_sets` `cs`
      join `mysql`.`collations` `col`
     where ((`mysql`.`col`.`character_set_id` = '45')
      and ('utf8mb4' = 'utf8mb4'))
```

As indicated by `SHOW WARNINGS`, the server handles the query on `COLLATION_CHARACTER_SET_APPLICABILITY` as a query on the `character_sets` and `collations` data dictionary tables in the `mysql` system database.

## 8.2.4 Optimizing Performance Schema Queries

Applications that monitor databases may make frequent use of Performance Schema tables. To write queries for these tables most efficiently, take advantage of their indexes. For example, include a `WHERE` clause that restricts retrieved rows based on comparison to specific values in an indexed column.

Most Performance Schema tables have indexes. Tables that do not are those that normally contain few rows or are unlikely to be queried frequently. Performance Schema indexes give the optimizer access to execution plans other than full table scans. These indexes also improve performance for related objects, such as `sys` schema views that use those tables.

To see whether a given Performance Schema table has indexes and what they are, use `SHOW INDEX` or `SHOW CREATE TABLE`:

```
mysql> SHOW INDEX FROM performance_schema.accounts\G
*****
 1. row *****

   Table: accounts
   Non_unique: 0
      Key_name: ACCOUNT
 Seq_in_index: 1
 Column_name: USER
    Collation: NULL
 Cardinality: NULL
     Sub_part: NULL
        Packed: NULL
       Null: YES
 Index_type: HASH
    Comment:
Index_comment:
   Visible: YES
*****
 2. row *****

   Table: accounts
   Non_unique: 0
      Key_name: ACCOUNT
 Seq_in_index: 2
 Column_name: HOST
    Collation: NULL
 Cardinality: NULL
     Sub_part: NULL
        Packed: NULL
       Null: YES
 Index_type: HASH
    Comment:
Index_comment:
   Visible: YES

mysql> SHOW CREATE TABLE performance_schema.rwlock_instances\G
*****
 1. row *****

   Table: rwlock_instances
Create Table: CREATE TABLE `rwlock_instances` (
  `NAME` varchar(128) NOT NULL,
  `OBJECT_INSTANCE_BEGIN` bigint(20) unsigned NOT NULL,
  `WRITE_LOCKED_BY_THREAD_ID` bigint(20) unsigned DEFAULT NULL,
  `READ_LOCKED_BY_COUNT` int(10) unsigned NOT NULL,
  PRIMARY KEY (`OBJECT_INSTANCE_BEGIN`),
  KEY `NAME` (`NAME`),
  KEY `WRITE_LOCKED_BY_THREAD_ID` (`WRITE_LOCKED_BY_THREAD_ID`)
) ENGINE=PERFORMANCE_SCHEMA DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci
```

To see the execution plan for a Performance Schema query and whether it uses any indexes, use `EXPLAIN`:

```
mysql> EXPLAIN SELECT * FROM performance_schema.accounts
 WHERE (USER,HOST) = ('root','localhost')\G
*****
 1. row *****
```

```

    id: 1
  select_type: SIMPLE
      table: accounts
    partitions: NULL
       type: const
possible_keys: ACCOUNT
         key: ACCOUNT
      key_len: 278
        ref: const,const
       rows: 1
  filtered: 100.00
     Extra: NULL

```

The `EXPLAIN` output indicates that the optimizer uses the `accounts` table `ACCOUNT` index that comprises the `USER` and `HOST` columns.

Performance Schema indexes are virtual: They are a construct of the Performance Schema storage engine and use no memory or disk storage. The Performance Schema reports index information to the optimizer so that it can construct efficient execution plans. The Performance Schema in turn uses optimizer information about what to look for (for example, a particular key value), so that it can perform efficient lookups without building actual index structures. This implementation provides two important benefits:

- It entirely avoids the maintenance cost normally incurred for tables that undergo frequent updates.
- It reduces at an early stage of query execution the amount of data retrieved. For conditions on the indexed columns, the Performance Schema efficiently returns only table rows that satisfy the query conditions. Without an index, the Performance Schema would return all rows in the table, requiring that the optimizer later evaluate the conditions against each row to produce the final result.

Performance Schema indexes are predefined and cannot be dropped, added, or altered.

Performance Schema indexes are similar to hash indexes. For example:

- They are used only for equality comparisons that use the `=` or `<= >` operators.
- They are unordered. If a query result must have specific row ordering characteristics, include an `ORDER BY` clause.

For additional information about hash indexes, see [Section 8.3.9, “Comparison of B-Tree and Hash Indexes”](#).

## 8.2.5 Optimizing Data Change Statements

This section explains how to speed up data change statements: `INSERT`, `UPDATE`, and `DELETE`. Traditional OLTP applications and modern web applications typically do many small data change operations, where concurrency is vital. Data analysis and reporting applications typically run data change operations that affect many rows at once, where the main considerations is the I/O to write large amounts of data and keep indexes up-to-date. For inserting and updating large volumes of data (known in the industry as ETL, for “extract-transform-load”), sometimes you use other SQL statements or external commands, that mimic the effects of `INSERT`, `UPDATE`, and `DELETE` statements.

### 8.2.5.1 Optimizing INSERT Statements

To optimize insert speed, combine many small operations into a single large operation. Ideally, you make a single connection, send the data for many new rows at once, and delay all index updates and consistency checking until the very end.

The time required for inserting a row is determined by the following factors, where the numbers indicate approximate proportions:

- Connecting: (3)
- Sending query to server: (2)
- Parsing query: (2)
- Inserting row: ( $1 \times$  size of row)
- Inserting indexes: ( $1 \times$  number of indexes)
- Closing: (1)

This does not take into consideration the initial overhead to open tables, which is done once for each concurrently running query.

The size of the table slows down the insertion of indexes by log  $N$ , assuming B-tree indexes.

You can use the following methods to speed up inserts:

- If you are inserting many rows from the same client at the same time, use `INSERT` statements with multiple `VALUES` lists to insert several rows at a time. This is considerably faster (many times faster in some cases) than using separate single-row `INSERT` statements. If you are adding data to a nonempty table, you can tune the `bulk_insert_buffer_size` variable to make data insertion even faster. See [Section 5.1.8, “Server System Variables”](#).
- When loading a table from a text file, use `LOAD DATA`. This is usually 20 times faster than using `INSERT` statements. See [Section 13.2.7, “LOAD DATA Syntax”](#).
- Take advantage of the fact that columns have default values. Insert values explicitly only when the value to be inserted differs from the default. This reduces the parsing that MySQL must do and improves the insert speed.
- See [Section 8.5.5, “Bulk Data Loading for InnoDB Tables”](#) for tips specific to `InnoDB` tables.
- See [Section 8.6.2, “Bulk Data Loading for MyISAM Tables”](#) for tips specific to `MyISAM` tables.

### 8.2.5.2 Optimizing UPDATE Statements

An update statement is optimized like a `SELECT` query with the additional overhead of a write. The speed of the write depends on the amount of data being updated and the number of indexes that are updated. Indexes that are not changed do not get updated.

Another way to get fast updates is to delay updates and then do many updates in a row later. Performing multiple updates together is much quicker than doing one at a time if you lock the table.

For a `MyISAM` table that uses dynamic row format, updating a row to a longer total length may split the row. If you do this often, it is very important to use `OPTIMIZE TABLE` occasionally. See [Section 13.7.3.4, “OPTIMIZE TABLE Syntax”](#).

### 8.2.5.3 Optimizing DELETE Statements

The time required to delete individual rows in a `MyISAM` table is exactly proportional to the number of indexes. To delete rows more quickly, you can increase the size of the key cache by increasing the `key_buffer_size` system variable. See [Section 5.1.1, “Configuring the Server”](#).

To delete all rows from a `MyISAM` table, `TRUNCATE TABLE tbl_name` is faster than `DELETE FROM tbl_name`. Truncate operations are not transaction-safe; an error occurs when attempting one in the course of an active transaction or active table lock. See [Section 13.1.37, “TRUNCATE TABLE Syntax”](#).

## 8.2.6 Optimizing Database Privileges

The more complex your privilege setup, the more overhead applies to all SQL statements. Simplifying the privileges established by `GRANT` statements enables MySQL to reduce permission-checking overhead when clients execute statements. For example, if you do not grant any table-level or column-level privileges, the server need not ever check the contents of the `tables_priv` and `columns_priv` tables. Similarly, if you place no resource limits on any accounts, the server does not have to perform resource counting. If you have a very high statement-processing load, consider using a simplified grant structure to reduce permission-checking overhead.

## 8.2.7 Other Optimization Tips

This section lists a number of miscellaneous tips for improving query processing speed:

- If your application makes several database requests to perform related updates, combining the statements into a stored routine can help performance. Similarly, if your application computes a single result based on several column values or large volumes of data, combining the computation into a UDF (user-defined function) can help performance. The resulting fast database operations are then available to be reused by other queries, applications, and even code written in different programming languages. See [Section 24.2, “Using Stored Routines”](#) and [Section 29.4, “Adding New Functions to MySQL”](#) for more information.
- To fix any compression issues that occur with `ARCHIVE` tables, use `OPTIMIZE TABLE`. See [Section 16.5, “The ARCHIVE Storage Engine”](#).
- If possible, classify reports as “live” or as “statistical”, where data needed for statistical reports is created only from summary tables that are generated periodically from the live data.
- If you have data that does not conform well to a rows-and-columns table structure, you can pack and store data into a `BLOB` column. In this case, you must provide code in your application to pack and unpack information, but this might save I/O operations to read and write the sets of related values.
- With Web servers, store images and other binary assets as files, with the path name stored in the database rather than the file itself. Most Web servers are better at caching files than database contents, so using files is generally faster. (Although you must handle backups and storage issues yourself in this case.)
- If you need really high speed, look at the low-level MySQL interfaces. For example, by accessing the MySQL `InnoDB` or `MyISAM` storage engine directly, you could get a substantial speed increase compared to using the SQL interface.
- Replication can provide a performance benefit for some operations. You can distribute client retrievals among replication servers to split up the load. To avoid slowing down the master while making backups, you can make backups using a slave server. See [Chapter 17, Replication](#).

## 8.3 Optimization and Indexes

The best way to improve the performance of `SELECT` operations is to create indexes on one or more of the columns that are tested in the query. The index entries act like pointers to the table rows, allowing the query to quickly determine which rows match a condition in the `WHERE` clause, and retrieve the other column values for those rows. All MySQL data types can be indexed.

Although it can be tempting to create an indexes for every possible column used in a query, unnecessary indexes waste space and waste time for MySQL to determine which indexes to use. Indexes also add to the cost of inserts, updates, and deletes because each index must be updated. You must find the right balance to achieve fast queries using the optimal set of indexes.

### 8.3.1 How MySQL Uses Indexes

Indexes are used to find rows with specific column values quickly. Without an index, MySQL must begin with the first row and then read through the entire table to find the relevant rows. The larger the table, the more this costs. If the table has an index for the columns in question, MySQL can quickly determine the position to seek to in the middle of the data file without having to look at all the data. This is much faster than reading every row sequentially.

Most MySQL indexes (`PRIMARY KEY`, `UNIQUE`, `INDEX`, and `FULLTEXT`) are stored in **B-trees**. Exceptions: Indexes on spatial data types use R-trees; `MEMORY` tables also support **hash indexes**; `InnoDB` uses inverted lists for `FULLTEXT` indexes.

In general, indexes are used as described in the following discussion. Characteristics specific to hash indexes (as used in `MEMORY` tables) are described in [Section 8.3.9, “Comparison of B-Tree and Hash Indexes”](#).

MySQL uses indexes for these operations:

- To find the rows matching a `WHERE` clause quickly.
- To eliminate rows from consideration. If there is a choice between multiple indexes, MySQL normally uses the index that finds the smallest number of rows (the most **selective** index).
- If the table has a multiple-column index, any leftmost prefix of the index can be used by the optimizer to look up rows. For example, if you have a three-column index on `(col1, col2, col3)`, you have indexed search capabilities on `(col1)`, `(col1, col2)`, and `(col1, col2, col3)`. For more information, see [Section 8.3.6, “Multiple-Column Indexes”](#).
- To retrieve rows from other tables when performing joins. 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. For example, `VARCHAR(10)` and `CHAR(10)` are the same size, but `VARCHAR(10)` and `CHAR(15)` are not.

For comparisons between nonbinary string columns, both columns should use the same character set. For example, comparing a `utf8` column with a `latin1` column precludes use of an index.

Comparison of dissimilar columns (comparing a string column to a temporal or numeric column, for example) may prevent use of indexes if values cannot be compared directly without conversion. For a given value such as `1` in the numeric column, it might compare equal to any number of values in the string column such as `'1'`, `' 1'`, `'00001'`, or `'01.e1'`. This rules out use of any indexes for the string column.

- To find the `MIN()` or `MAX()` value for a specific indexed column `key_col`. This is optimized by a preprocessor that checks whether you are using `WHERE key_part_N = constant` on all key parts that occur before `key_col` in the index. In this case, MySQL does a single key lookup for each `MIN()` or `MAX()` expression and replaces it with a constant. If all expressions are replaced with constants, the query returns at once. For example:

```
SELECT MIN(key_part2),MAX(key_part2)
  FROM tbl_name WHERE key_part1=10;
```

- To sort or group a table if the sorting or grouping is done on a leftmost prefix of a usable index (for example, `ORDER BY key_part1, key_part2`). If all key parts are followed by `DESC`, the key is read in reverse order. (Or, if the index is a descending index, the key is read in forward order.) See [Section 8.2.1.14, “ORDER BY Optimization”](#), [Section 8.2.1.15, “GROUP BY Optimization”](#), and [Section 8.3.13, “Descending Indexes”](#).

- In some cases, a query can be optimized to retrieve values without consulting the data rows. (An index that provides all the necessary results for a query is called a [covering index](#).) If a query uses from a table only columns that are included in some index, the selected values can be retrieved from the index tree for greater speed:

```
SELECT key_part3 FROM tbl_name  
WHERE key_part1=1
```

Indexes are less important for queries on small tables, or big tables where report queries process most or all of the rows. When a query needs to access most of the rows, reading sequentially is faster than working through an index. Sequential reads minimize disk seeks, even if not all the rows are needed for the query. See [Section 8.2.1.21, “Avoiding Full Table Scans”](#) for details.

### 8.3.2 Primary Key Optimization

The primary key for a table represents the column or set of columns that you use in your most vital queries. It has an associated index, for fast query performance. Query performance benefits from the [NOT NULL](#) optimization, because it cannot include any [NULL](#) values. With the [InnoDB](#) storage engine, the table data is physically organized to do ultra-fast lookups and sorts based on the primary key column or columns.

If your table is big and important, but does not have an obvious column or set of columns to use as a primary key, you might create a separate column with auto-increment values to use as the primary key. These unique IDs can serve as pointers to corresponding rows in other tables when you join tables using foreign keys.

### 8.3.3 SPATIAL Index Optimization

MySQL permits creation of [SPATIAL](#) indexes on [NOT NULL](#) geometry-valued columns (see [Section 11.5.10, “Creating Spatial Indexes”](#)). The optimizer checks the [SRID](#) attribute for indexed columns to determine which spatial reference system (SRS) to use for comparisons, and uses calculations appropriate to the SRS. (Prior to MySQL 8.0, the optimizer performs comparisons of [SPATIAL](#) index values using Cartesian calculations; the results of such operations are undefined if the column contains values with non-Cartesian SRIDs.)

For comparisons to work properly, each column in a [SPATIAL](#) index must be SRID-restricted. That is, the column definition must include an explicit [SRID](#) attribute, and all column values must have the same SRID.

The optimizer considers [SPATIAL](#) indexes only for SRID-restricted columns:

- Indexes on columns restricted to a Cartesian SRID enable Cartesian bounding box computations.
- Indexes on columns restricted to a geographic SRID enable geographic bounding box computations.

The optimizer ignores [SPATIAL](#) indexes on columns that have no [SRID](#) attribute (and thus are not SRID-restricted). MySQL still maintains such indexes, as follows:

- They are updated for table modifications ([INSERT](#), [UPDATE](#), [DELETE](#), and so forth). Updates occur as though the index was Cartesian, even though the column might contain a mix of Cartesian and geographical values.
- They exist only for backward compatibility; for example, the ability to perform a dump in MySQL 5.7 and restore in MySQL 8.0. Because [SPATIAL](#) indexes on columns that are not SRID-restricted are of no use to the optimizer, each such column should be modified:
  - Verify that all values within the column have the same SRID. To determine the SRIDs contained in a geometry column [col\\_name](#), use the following query:

```
SELECT DISTINCT ST_SRID(col_name) FROM tbl_name;
```

If the query returns more than one row, the column contains a mix of SRIDs. In that case, modify its contents so all values have the same SRID.

- Redefine the column to have an explicit [SRID](#) attribute.
- Recreate the [SPATIAL](#) index.

### 8.3.4 Foreign Key Optimization

If a table has many columns, and you query many different combinations of columns, it might be efficient to split the less-frequently used data into separate tables with a few columns each, and relate them back to the main table by duplicating the numeric ID column from the main table. That way, each small table can have a primary key for fast lookups of its data, and you can query just the set of columns that you need using a join operation. Depending on how the data is distributed, the queries might perform less I/O and take up less cache memory because the relevant columns are packed together on disk. (To maximize performance, queries try to read as few data blocks as possible from disk; tables with only a few columns can fit more rows in each data block.)

### 8.3.5 Column Indexes

The most common type of index involves a single column, storing copies of the values from that column in a data structure, allowing fast lookups for the rows with the corresponding column values. The B-tree data structure lets the index quickly find a specific value, a set of values, or a range of values, corresponding to operators such as `=`, `>`, `≤`, `BETWEEN`, `IN`, and so on, in a `WHERE` clause.

The maximum number of indexes per table and the maximum index length is defined per storage engine. See [Chapter 15, The InnoDB Storage Engine](#), and [Chapter 16, Alternative Storage Engines](#). All storage engines support at least 16 indexes per table and a total index length of at least 256 bytes. Most storage engines have higher limits.

For additional information about column indexes, see [Section 13.1.15, “CREATE INDEX Syntax”](#).

- [Index Prefixes](#)
- [FULLTEXT Indexes](#)
- [Spatial Indexes](#)
- [Indexes in the MEMORY Storage Engine](#)

#### Index Prefixes

With `col_name(N)` syntax in an index specification for a string column, you can create an index that uses only the first *N* characters of the column. Indexing only a prefix of column values in this way can make the index file much smaller. When you index a `BLOB` or `TEXT` column, you *must* specify a prefix length for the index. For example:

```
CREATE TABLE test (blob_col BLOB, INDEX(blob_col(10)));
```

Prefixes can be up to 767 bytes long for [InnoDB](#) tables that use the [REDUNDANT](#) or [COMPACT](#) row format. The prefix length limit is 3072 bytes for [InnoDB](#) tables that use the [DYNAMIC](#) or [COMPRESSED](#) row format. For MyISAM tables, the prefix length limit is 1000 bytes.

**Note**

Prefix limits are measured in bytes, whereas the prefix length in `CREATE TABLE`, `ALTER TABLE`, and `CREATE INDEX` statements is interpreted as number of characters for nonbinary string types (`CHAR`, `VARCHAR`, `TEXT`) and number of bytes for binary string types (`BINARY`, `VARBINARY`, `BLOB`). Take this into account when specifying a prefix length for a nonbinary string column that uses a multibyte character set.

If a search term exceeds the index prefix length, the index is used to exclude non-matching rows, and the remaining rows are examined for possible matches.

For additional information about index prefixes, see [Section 13.1.15, “CREATE INDEX Syntax”](#).

## FULLTEXT Indexes

`FULLTEXT` indexes are used for full-text searches. Only the `InnoDB` and `MyISAM` storage engines support `FULLTEXT` indexes and only for `CHAR`, `VARCHAR`, and `TEXT` columns. Indexing always takes place over the entire column and column prefix indexing is not supported. For details, see [Section 12.9, “Full-Text Search Functions”](#).

Optimizations are applied to certain kinds of `FULLTEXT` queries against single `InnoDB` tables. Queries with these characteristics are particularly efficient:

- `FULLTEXT` queries that only return the document ID, or the document ID and the search rank.
- `FULLTEXT` queries that sort the matching rows in descending order of score and apply a `LIMIT` clause to take the top N matching rows. For this optimization to apply, there must be no `WHERE` clauses and only a single `ORDER BY` clause in descending order.
- `FULLTEXT` queries that retrieve only the `COUNT(*)` value of rows matching a search term, with no additional `WHERE` clauses. Code the `WHERE` clause as `WHERE MATCH(text) AGAINST ('other_text')`, without any `> 0` comparison operator.

For queries that contain full-text expressions, MySQL evaluates those expressions during the optimization phase of query execution. The optimizer does not just look at full-text expressions and make estimates, it actually evaluates them in the process of developing an execution plan.

An implication of this behavior is that `EXPLAIN` for full-text queries is typically slower than for non-full-text queries for which no expression evaluation occurs during the optimization phase.

`EXPLAIN` for full-text queries may show `Select tables optimized away` in the `Extra` column due to matching occurring during optimization; in this case, no table access need occur during later execution.

## Spatial Indexes

You can create indexes on spatial data types. `MyISAM` and `InnoDB` support R-tree indexes on spatial types. Other storage engines use B-trees for indexing spatial types (except for `ARCHIVE`, which does not support spatial type indexing).

## Indexes in the MEMORY Storage Engine

The `MEMORY` storage engine uses `HASH` indexes by default, but also supports `BTREE` indexes.

### 8.3.6 Multiple-Column Indexes

MySQL can create composite indexes (that is, indexes on multiple columns). An index may consist of up to 16 columns. For certain data types, you can index a prefix of the column (see [Section 8.3.5, “Column Indexes”](#)).

MySQL can use multiple-column indexes for queries that test all the columns in the index, or queries that test just the first column, the first two columns, the first three columns, and so on. If you specify the columns in the right order in the index definition, a single composite index can speed up several kinds of queries on the same table.

A multiple-column index can be considered a sorted array, the rows of which contain values that are created by concatenating the values of the indexed columns.



### Note

As an alternative to a composite index, you can introduce a column that is “hashed” based on information from other columns. If this column is short, reasonably unique, and indexed, it might be faster than a “wide” index on many columns. In MySQL, it is very easy to use this extra column:

```
SELECT * FROM tbl_name
  WHERE hash_col=MD5(CONCAT(val1,val2))
    AND col1=val1 AND col2=val2;
```

Suppose that a table has the following specification:

```
CREATE TABLE test (
  id      INT NOT NULL,
  last_name CHAR(30) NOT NULL,
  first_name CHAR(30) NOT NULL,
  PRIMARY KEY (id),
  INDEX name (last_name,first_name)
);
```

The `name` index is an index over the `last_name` and `first_name` columns. The index can be used for lookups in queries that specify values in a known range for combinations of `last_name` and `first_name` values. It can also be used for queries that specify just a `last_name` value because that column is a leftmost prefix of the index (as described later in this section). Therefore, the `name` index is used for lookups in the following queries:

```
SELECT * FROM test WHERE last_name='Widenius';

SELECT * FROM test
  WHERE last_name='Widenius' AND first_name='Michael';

SELECT * FROM test
  WHERE last_name='Widenius'
    AND (first_name='Michael' OR first_name='Monty');

SELECT * FROM test
  WHERE last_name='Widenius'
    AND first_name >='M' AND first_name < 'N';
```

However, the `name` index is *not* used for lookups in the following queries:

```
SELECT * FROM test WHERE first_name='Michael';

SELECT * FROM test
  WHERE last_name='Widenius' OR first_name='Michael';
```

Suppose that you issue the following `SELECT` statement:

```
SELECT * FROM tbl_name
  WHERE col1=val1 AND col2=val2;
```

If a multiple-column index exists on `col1` and `col2`, the appropriate rows can be fetched directly. If separate single-column indexes exist on `col1` and `col2`, the optimizer attempts to use the Index Merge optimization (see [Section 8.2.1.3, “Index Merge Optimization”](#)), or attempts to find the most restrictive index by deciding which index excludes more rows and using that index to fetch the rows.

If the table has a multiple-column index, any leftmost prefix of the index can be used by the optimizer to look up rows. For example, if you have a three-column index on `(col1, col2, col3)`, you have indexed search capabilities on `(col1)`, `(col1, col2)`, and `(col1, col2, col3)`.

MySQL cannot use the index to perform lookups if the columns do not form a leftmost prefix of the index. Suppose that you have the `SELECT` statements shown here:

```
SELECT * FROM tbl_name WHERE col1=val1;
SELECT * FROM tbl_name WHERE col1=val1 AND col2=val2;
SELECT * FROM tbl_name WHERE col2=val2;
SELECT * FROM tbl_name WHERE col2=val2 AND col3=val3;
```

If an index exists on `(col1, col2, col3)`, only the first two queries use the index. The third and fourth queries do involve indexed columns, but do not use an index to perform lookups because `(col2)` and `(col2, col3)` are not leftmost prefixes of `(col1, col2, col3)`.

### 8.3.7 Verifying Index Usage

Always check whether all your queries really use the indexes that you have created in the tables. Use the `EXPLAIN` statement, as described in [Section 8.8.1, “Optimizing Queries with EXPLAIN”](#).

### 8.3.8 InnoDB and MyISAM Index Statistics Collection

Storage engines collect statistics about tables for use by the optimizer. Table statistics are based on value groups, where a value group is a set of rows with the same key prefix value. For optimizer purposes, an important statistic is the average value group size.

MySQL uses the average value group size in the following ways:

- To estimate how many rows must be read for each `ref` access
- To estimate how many rows a partial join will produce; that is, the number of rows that an operation of this form will produce:

```
(...) JOIN tbl_name ON tbl_name.key = expr
```

As the average value group size for an index increases, the index is less useful for those two purposes because the average number of rows per lookup increases: For the index to be good for optimization purposes, it is best that each index value target a small number of rows in the table. When a given index value yields a large number of rows, the index is less useful and MySQL is less likely to use it.

The average value group size is related to table cardinality, which is the number of value groups. The `SHOW INDEX` statement displays a cardinality value based on  $N/S$ , where  $N$  is the number of rows in the table and  $S$  is the average value group size. That ratio yields an approximate number of value groups in the table.

For a join based on the `<=>` comparison operator, `NULL` is not treated differently from any other value: `NULL <=> NULL`, just as `N <=> N` for any other `N`.

However, for a join based on the `=` operator, `NULL` is different from non-`NULL` values: `expr1 = expr2` is not true when `expr1` or `expr2` (or both) are `NULL`. This affects `ref` accesses for comparisons of the form `tbl_name.key = expr`: MySQL will not access the table if the current value of `expr` is `NULL`, because the comparison cannot be true.

For `=` comparisons, it does not matter how many `NULL` values are in the table. For optimization purposes, the relevant value is the average size of the non-`NULL` value groups. However, MySQL does not currently enable that average size to be collected or used.

For InnoDB and MyISAM tables, you have some control over collection of table statistics by means of the `innodb_stats_method` and `myisam_stats_method` system variables, respectively. These variables have three possible values, which differ as follows:

- When the variable is set to `nulls_equal`, all `NULL` values are treated as identical (that is, they all form a single value group).

If the `NULL` value group size is much higher than the average non-`NULL` value group size, this method skews the average value group size upward. This makes index appear to the optimizer to be less useful than it really is for joins that look for non-`NULL` values. Consequently, the `nulls_equal` method may cause the optimizer not to use the index for `ref` accesses when it should.

- When the variable is set to `nulls_unequal`, `NULL` values are not considered the same. Instead, each `NULL` value forms a separate value group of size 1.

If you have many `NULL` values, this method skews the average value group size downward. If the average non-`NULL` value group size is large, counting `NULL` values each as a group of size 1 causes the optimizer to overestimate the value of the index for joins that look for non-`NULL` values. Consequently, the `nulls_unequal` method may cause the optimizer to use this index for `ref` lookups when other methods may be better.

- When the variable is set to `nulls_ignored`, `NULL` values are ignored.

If you tend to use many joins that use `<=>` rather than `=`, `NULL` values are not special in comparisons and one `NULL` is equal to another. In this case, `nulls_equal` is the appropriate statistics method.

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 will be 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](#)
- [Hash Index Characteristics](#)

#### 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 optimizer cannot use the primary key in this case because that comprises columns `(i1, i2)` and the query does not refer to `i2`. Instead, the optimizer can use the secondary index `k_d` on `(d)`, and 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 will use 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 will 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 will 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 index 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.

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 FULLTEXT 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.

### 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 5.1.13, “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 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  $N \times$  the maximum byte length of the character set. Many languages can be written primarily using single-byte `utf8` 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  $N \times$  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 Syntax”](#)). 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.6.2.17, “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 13.2.4, “HANDLER Syntax”](#).

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 13.2.11.8, “Derived Tables”](#)).
- Evaluation of common table expressions (see [Section 13.2.13, “WITH Syntax \(Common Table Expressions\)”](#)).
- Tables created for subquery or semi-join 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 13.2.6.1, “INSERT ... SELECT Syntax”](#).
- 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` will 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.

When the server creates an internal temporary table (either in memory or on disk), it increments the `Created_tmp_tables` status variable. If the server creates the table on disk (either initially or by converting an in-memory table) it increments the `Created_tmp_disk_tables` status variable.

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

The `TempTable` storage engine provides efficient storage for `VARCHAR` and `VARBINARY` columns. Storage of other binary large object types is supported as of MySQL 8.0.13. The `temptable_max_ram` configuration option defines the maximum amount of RAM that can be occupied by the `TempTable` storage engine before it starts allocating space from disk in the form memory-mapped temporary files (the default) or `InnoDB` on-disk internal temporary tables. The default `temptable_max_ram` setting is 1GiB. The `temptable_use_mmap` variable (introduced in MySQL 8.0.16) controls whether the `TempTable` storage engine uses memory-mapped files or `InnoDB` on-disk internal temporary tables when the `temptable_max_ram` limit is exceeded. The default setting is `temptable_use_mmap=ON`.

Use of memory-mapped temporary files by the [TempTable](#) storage engine as an overflow mechanism for in-memory temporary tables 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.

If the [TempTable](#) storage engine is configured to use [InnoDB](#) on-disk internal temporary tables as the overflow mechanism ([temptable\\_use\\_mmap=OFF](#)), an in-memory table that exceeds the [temptable\\_max\\_ram](#) limit is converted to an [InnoDB](#) on-disk internal temporary table, and any rows belonging to that table are moved from memory to the [InnoDB](#) on-disk internal temporary table. The [internal\\_tmp\\_disk\\_storage\\_engine](#) (removed in MySQL 8.0.16) variable setting has no affect on the [TempTable](#) storage engine overflow mechanism.

Consider using [InnoDB](#) on-disk internal temporary tables as the [TempTable](#) overflow mechanism if the [TempTable](#) storage engine often exceeds the memory limit defined by the [temptable\\_max\\_ram](#) variable and uses excessive space in the temporary directory for memory-mapped files. This may occur due to use of large internal temporary tables or extensive use of internal temporary tables. [InnoDB](#) on-disk internal temporary tables are created in session temporary tablespaces, which reside in the data directory by default. For more information, see [Section 15.6.3.5, “Temporary Tablespaces”](#).

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 ([temptable\\_use\\_mmap=ON](#)). If the [physical\\_disk](#) instrument reports a value other than 0 and memory-mapped files are used as the [TempTable](#) overflow mechanism, the [temptable\\_max\\_ram](#) threshold 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 26.12.16.10, “Memory Summary Tables”](#).

When using the [MEMORY](#) storage engine for in-memory temporary tables, MySQL automatically converts an in-memory temporary table to an on-disk table if it becomes too large. The maximum size for in-memory temporary tables 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

Starting with MySQL 8.0.16, the server always uses the [InnoDB](#) storage engine for managing internal temporary tables on disk.

In MySQL 8.0.15 and earlier, the [internal\\_tmp\\_disk\\_storage\\_engine](#) variable was used to define the storage engine used for on-disk internal temporary tables. This variable was removed in MySQL 8.0.16, and the storage engine used for this purpose is no longer user-configurable.

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.

In MySQL 8.0.15 and earlier: 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, or 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 exceeds the `temptable_max_ram` limit and starts allocating space from disk as memory-mapped files or `InnoDB` on-disk internal temporary tables.

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.

Previous 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 handled 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.

## 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 `utf8` 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 15.17.2, “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 a crash 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 Syntax`”](#).



#### 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:

- Make your redo log files big, even as big as the `buffer pool`. When InnoDB has written the 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. Although historically big redo log files caused lengthy recovery times, recovery is now much faster and you can confidently use large redo log files.

The size and number of redo log files are configured using the `innodb_log_file_size` and `innodb_log_files_in_group` configuration options. For information about modifying an existing redo log file configuration, see [Changing the Number or Size of Redo Log Files](#).

- 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.

- 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 15.6.1.4, “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:
  - 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);
```

- Load the data into the table.
- 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 will add the `FTS_DOC_ID` as a hidden column with the next `CREATE FULLTEXT INDEX` call. This approach, however, requires a table rebuild which will impact performance.

## 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 size for the write buffer

By default, when InnoDB creates a new data file, such as a new log file or tablespace file, it flushes the contents of the write buffer to disk only after the file is fully written, which can cause a large amount of disk write activity to occur at once. To force smaller, periodic flushes, use `innodb_fsync_threshold` (introduced in MySQL 8.0.13) to define a threshold size for the write buffer, in bytes. The contents of the write buffer are flushed to disk when the threshold size is reached. The default value of 0 forces the default behavior.

Specifying a write buffer threshold size 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 write buffer threshold size helps avoid such surges in disk write activity.

- 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 15.6.3.6, “Creating a Tablespace Outside of the Data Directory”](#)
- [Creating a General Tablespace](#)
- [Section 15.6.1.2, “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](#) and [change buffering](#)) 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](#)

This option 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\\_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](#)

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 system tablespace files ("ibdata files") on Fusion-io devices that support atomic writes. In this case, doublewrite buffering (`innodb_doublewrite`) 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, doublewrite buffering is also disabled for data files residing on non-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 will not 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 15.15, “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 the InnoDB Master Thread I/O Rate”](#). 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 InnoDB Buffer Pool Flushing”](#). The algorithm works for some types of workloads but not others, so might turn off this setting 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, “Configuring InnoDB Purge Scheduling”](#). 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` option, 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 Syntax”](#).
- 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 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 will use the faster filesort method and avoid 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
<code>key_buffer_size</code>	<code>key_buffer_size</code>
<code>myisam_sort_buffer_size</code>	<code>myisam_sort_buffer_size</code>
<code>read_buffer_size</code>	<code>read_buffer_size</code>
<code>write_buffer_size</code>	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 8.2.1.21, “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 23.3.5, “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 13.2.10, “SELECT Syntax”](#).) However, `STRAIGHT_JOIN` may prevent indexes from being used because it disables semi-join transformations. See [Section 8.2.2.1, “Optimizing Subqueries, Derived Tables, View References, and Common Table Expressions with Semi-Join 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 Syntax”](#).



#### 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 Syntax”](#), and [Section 13.7.6.5, “SHOW COLUMNS Syntax”](#).

## 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. MySQL resolves all joins using a nested-loop join method. This means that MySQL reads a row from the first table, and then finds a matching row in the second table, 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
<code>id</code>	<code>select_id</code>	The <code>SELECT</code> identifier
<code>select_type</code>	<code>None</code>	The <code>SELECT</code> type
<code>table</code>	<code>table_name</code>	The table for the output row
<code>partitions</code>	<code>partitions</code>	The matching partitions
<code>type</code>	<code>access_type</code>	The join type
<code>possible_keys</code>	<code>possible_keys</code>	The possible indexes to choose
<code>key</code>	<code>key</code>	The index actually chosen
<code>key_len</code>	<code>key_length</code>	The length of the chosen key
<code>ref</code>	<code>ref</code>	The columns compared to the index
<code>rows</code>	<code>rows</code>	Estimate of rows to be examined
<code>filtered</code>	<code>filtered</code>	Percentage of rows filtered by table condition
<code>Extra</code>	<code>None</code>	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.

<code>select_type</code> Value	JSON Name	Meaning
<code>SIMPLE</code>	None	Simple <code>SELECT</code> (not using <code>UNION</code> or subqueries)
<code>PRIMARY</code>	None	Outermost <code>SELECT</code>
<code>UNION</code>	None	Second or later <code>SELECT</code> statement in a <code>UNION</code>
<code>DEPENDENT UNION</code>	<code>dependent (true)</code>	Second or later <code>SELECT</code> statement in a <code>UNION</code> , dependent on outer query
<code>UNION RESULT</code>	<code>union_result</code>	Result of a <code>UNION</code> .
<code>SUBQUERY</code>	None	First <code>SELECT</code> in subquery
<code>DEPENDENT SUBQUERY</code>	<code>dependent (true)</code>	First <code>SELECT</code> in subquery, dependent on outer query
<code>DERIVED</code>	None	Derived table
<code>DEPENDENT DERIVED</code>	<code>dependent (true)</code>	Derived table dependent on another table
<code>MATERIALIZED</code>	<code>materialized_from_subquery</code>	Materialized subquery
<code>UNCACHEABLE SUBQUERY</code>	<code>cacheable (false)</code>	A subquery for which the result cannot be cached and must be re-evaluated for each row of the outer query
<code>UNCACHEABLE UNION</code>	<code>cacheable (false)</code>	The second or later select in a <code>UNION</code> that belongs to an uncacheable subquery (see <code>UNCACHEABLE SUBQUERY</code> )

`DEPENDENT` typically signifies the use of a correlated subquery. See [Section 13.2.11.7, “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:

- `<unionM,N>`: The row refers to the union of the rows with `id` values of `M` and `N`.
- `<derivedN>`: 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.
- `<subqueryN>`: 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 23.3.5, “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 Syntax”](#).

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` will 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 Syntax”](#), 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 `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 will be 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 will be 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  $1000 \times 50\% = 500$ .

- `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 8.2.1.13, “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`.

- `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 semi-join FirstMatch join shortcircuiting 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 semi-join 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`.

- `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 will be considered.

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

This indicates that the row applies to the recursive `SELECT` part of a recursive common table expression. See [Section 13.2.13, “WITH Syntax \(Common Table Expressions\)”](#).

- **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 semi-join 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 8.2.1.14, “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`.

- `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 8.2.1.5, “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 8.2.1.15, “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)` (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 and (`Batched Key Access`) indicates use of the Batched Key Access algorithm. That is, the keys from the table on the preceding line of the `EXPLAIN` output will be buffered, and the matching rows will be 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 either one of `Block Nested Loop` or `Batched Key Access`.

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

Tables are read using the Multi-Range Read optimization strategy. See [Section 8.2.1.10, “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 8.2.1.4, “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)
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, 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  $74 \times 2135 \times 74 \times 3872 = 45,268,558,720$  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,
                  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,
                  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 where
		ClientID,					
		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 semi-join transformations. See [Section 8.2.2.1, “Optimizing Subqueries, Derived Tables, View References, and Common Table Expressions with Semi-Join 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 13.2.11.8, “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 you will see `<temporary table>` 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 semi-join operation. `inner_tables` shows the tables that were not pulled out. See [Section 8.2.2.1, “Optimizing Subqueries, Derived Tables, View References, and Common Table Expressions with Semi-Join 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.

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:  $\log(\text{row\_count}) / \log(\text{index\_block\_length} / 3 * 2 / (\text{index\_length} + \text{data\_pointer\_length})) + 1$ .

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  $500,000 * 7 * 3/2 = 5.2\text{MB}$  (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 also maintains statistics about column values, although the optimizer does not yet use this information.

### 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
```

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
<code>default</code>	Reset every optimization to its default value
<code>opt_name=default</code>	Set the named optimization to its default value
<code>opt_name=off</code>	Disable the named optimization
<code>opt_name=on</code>	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 8.2.1.11, “Block Nested-Loop and Batched Key Access Joins”](#).

- Block Nested-Loop Flags

- `block_nested_loop` (default `on`)

Controls use of BNL join algorithm.

For more information, see [Section 8.2.1.11, “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 8.2.1.12, “Condition Filtering”](#).

- 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 8.2.1.4, “Engine Condition Pushdown Optimization”](#).

- Index Condition Pushdown Flags
  - `index_condition_pushdown` (default `on`)  
Controls index condition pushdown.

For more information, see [Section 8.2.1.5, “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”](#).

- 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 8.2.1.10, “Multi-Range Read Optimization”](#).

- Skip Scan Flags
  - `skip_scan` (default `on`)  
Controls use of Skip Scan access method.
- Semi-Join Flags
  - `semijoin` (default `on`)  
Controls all semi-join strategies.
  - `duplicateweedout` (default `on`)  
Controls the semi-join Duplicate Weedout strategy.
  - `firstmatch` (default `on`)  
Controls the semi-join FirstMatch strategy.
  - `loosescan` (default `on`)  
Controls the semi-join LooseScan strategy (not to be confused with Loose Index Scan for `GROUP BY`).

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

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

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

For more information, see [Section 8.2.2.1, “Optimizing Subqueries, Derived Tables, View References, and Common Table Expressions with Semi-Join Transformations”](#).

- Subquery Materialization Flags
  - `materialization` (default `on`)  
Controls materialization (including semi-join 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`, semi-joins 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”](#).

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
```

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 will 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
```

### 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's 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
BNL, NO_BNL	Affects Block Nested-Loop join processing	Query block, table
INDEX_MERGE, NO_INDEX_MERGE	Affects Index Merge optimization	Table, index
JOIN_FIXED_ORDER	Use table order specified in <code>FROM</code> clause for join order	Query block
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
QB_NAME	Assigns name to query block	Query block
RESOURCE_GROUP	Set resource group during statement execution	Global
SEMIJOIN, NO_SEMIJOIN	Affects semi-join strategies	Query block
SKIP_SCAN, NO_SKIP_SCAN	Affects Skip Scan optimization	Table, index
SET_VAR	Set variable during statement execution	Global
SUBQUERY	Affects materialization, <code>IN</code> -to- <code>EXISTS</code> 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 will use it.

## Optimizer Hint Syntax

MySQL supports comments in SQL statements as described in [Section 9.6, “Comment Syntax”](#). 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 9.2.2, “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 semi-join tables that are merged to the outer query block. If subqueries `subq1` and `subq2` are converted to semi-joins, 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 8.2.1.11, “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.

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 BKA for the specified tables.
  - `BNL`, `NO_BNL`: Enable or disable BNL for the specified tables.
  - `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 BNL or BKA 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:
  - `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.

- `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 8.2.1.10, “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 8.2.1.5, “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.

- **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 *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 semi-join transformations and which semi-join strategies to permit, and, when semi-joins 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 semi-join 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 semi-join strategies.
  - `strategy`: A semi-join strategy to be enabled or disabled. These strategy names are permitted: `DUPSWEEOUT, FIRSTMATCH, LOOSESCAN, MATERIALIZATION`.

For `SEMIJOIN` hints, if no strategies are named, semi-join 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, `DUPSWEEOUT` is used.

For `NO_SEMIJOIN` hints, if no strategies are named, semi-join is not used. If strategies are named that rule out all applicable strategies for the statement, `DUPSWEEOUT` is used.

If one subquery is nested within another and both are merged into a semi-join of an outer query, any specification of semi-join strategies for the innermost query are ignored. `SEMIJOIN` and `NO_SEMIJOIN` hints can still be used to enable or disable semi-join transformations for such nested subqueries.

If `DUPSWEEOUT` 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, DUPSWEEOUT) */ * 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 semi-join 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 = 'utf8') */ 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_flag` 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.

Slave servers 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 8.12.5, “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 Syntax”](#).

## 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 only to `SELECT` statements. (They are accepted by the parser for `UPDATE` statements but are ignored and have no effect.)

Index hints are specified following a table name. (For the general syntax for specifying tables in a `SELECT` statement, see [Section 13.2.10.2, “JOIN Syntax”](#).) 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.

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 master and slaves 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 (index1) FOR ORDER BY
  IGNORE INDEX (index1) FOR GROUP BY
  WHERE ... IN BOOLEAN MODE ... ;

SELECT * FROM t
  USE INDEX (index1)
  WHERE ... IN BOOLEAN MODE ... ;
```

## 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](#)
- [The Cost Model Database](#)
- [Making Changes to the Cost Model Database](#)

### 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 reread 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 rereads 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 replication slaves.

### 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 reread 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_temp_table_create_cost, disk_temp_table_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_temp_table_create_cost, memory_temp_table_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_temp_table_create_cost, memory_temp_table_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_temp_table_create_cost`, `disk_temp_table_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 will have 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 will be 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; see [Section 13.7.3.1, “ANALYZE TABLE Syntax”](#).

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 .hhmmss` 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 InnoDB 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 15.8.3.7, “Saving and Restoring the Buffer Pool State”](#)
- [Section 15.8.3.6, “Fine-tuning InnoDB Buffer Pool Flushing”](#)
- [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 of 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 13.7.7.2, “CACHE INDEX Syntax”](#)). 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` option 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  $N * \text{key\_cache\_age\_threshold} / 100$  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 will 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 reconvert ing 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 will return 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 26.12.12, “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 12.14, “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 will 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 13.2.10, “SELECT Syntax”](#).
- 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 inserts to the table occur. This permits `READ` locks after a certain number of `WRITE` locks.
- If you have problems with `INSERT` combined with `SELECT`, consider switching to [MyISAM](#) tables, which support concurrent `SELECT` and `INSERT` statements. (See [Section 8.11.3, “Concurrent Inserts”](#).)

- If you have problems with mixed `SELECT` and `DELETE` statements, the `LIMIT` option to `DELETE` may help. See [Section 13.2.2, “DELETE Syntax”](#).
- Using `SQL_BUFFER_RESULT` with `SELECT` statements can help to make the duration of table locks shorter. See [Section 13.2.10, “SELECT Syntax”](#).
- 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, and scheduled events), and tablespaces.

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 26.12.12.3, “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 tblob, tblc TO tbla;
```

In this case, the statement acquires metadata locks, in order, on `tbla`, `tblob`, and `tblc` (because `tblob` 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) will block and be 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 as checking tables. In this case, if the server tries to update a table that `myisamchk` is using, the server will wait 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`. (See [System Factors](#).) 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=ALL` option 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  $N \text{ MOD } \text{number\_of\_disks}$  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  $2 \times N$  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

Caution is advised when considering using NFS with MySQL. Potential issues, which vary by operating system and NFS version, include:

- 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 on the `CREATE TABLE` statement instead of symbolic links, as explained in [Section 15.6.3.6, “Creating a Tablespace Outside of the Data Directory”](#). 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, the way to symlink a database is first to create a directory on some disk where you have free space and then to create a soft link to it from the MySQL data directory.

```
shell> mkdir /dr1/databases/test
shell> ln -s /dr1/databases/test /path/to/datadir
```

MySQL does not support linking one directory to multiple databases. Replacing a database directory with a symbolic link works as long as you do not make a symbolic link between databases. Suppose that you have a database `db1` under the MySQL data directory, and then make a symlink `db2` that points to `db1`:

```
shell> cd /path/to/datadir
shell> ln -s db1 db2
```

The result is that, for any table `tbl_a` in `db1`, there also appears to be a table `tbl_a` in `db2`. If one client updates `db1.tbl_a` and another client updates `db2.tbl_a`, problems are likely to occur.

### 8.12.2.2 Using Symbolic Links for MyISAM Tables on Unix



#### Note

Symbolic link support as described here, along with the the `--symbolic-links` option that controls it, is deprecated and will 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 15.6.3.6, “Creating a Tablespace Outside of the Data Directory”](#) 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 Syntax”](#). 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 will be ineffective. 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. Change location into the data directory:

```
C:\> cd \path\to\datadir
```

2. 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
```

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 will 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 26.17, “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 26.10, “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.6.3.5, “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 will be an increase in memory use. This cached memory can be 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 master limits the maximum message size that the master sends to its slaves for processing. This setting defaults to 64M.
  - The `slave_pending_jobs_size_max` system variable on a multi-threaded slave 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 master or slave 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 masters and coordinator threads on slaves.
  - 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.

## 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 26.12.16.10, “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, sys.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

For more information about `sys` schema, see [Chapter 27, MySQL sys Schema](#).

### 8.12.3.2 Enabling Large Page Support

Some hardware/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 kernel for some recent systems such as Red Hat Enterprise Linux appear to 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”:

```
shell> cat /proc/meminfo | grep -i huge
HugePages_Total:      0
HugePages_Free:       0
HugePages_Rsvd:       0
HugePages_Surp:       0
Hugepagesize:        4096 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 commands. Normally, you put these in an `rc` file or equivalent startup file that is executed during the system boot sequence, so that the commands execute each time the system starts. The commands should execute early in the boot sequence, before the MySQL server starts. Be sure to change the allocation numbers and the group number as appropriate for your system.

```
# Set the number of pages to be used.
```

```
# Each page is normally 2MB, so a value of 20 = 40MB.  
# This command actually allocates memory, so this much  
# memory must be available.  
echo 20 > /proc/sys/vm/nr_hugepages  
  
# Set the group number that is permitted to access this  
# memory (102 in this case). The mysql user must be a  
# member of this group.  
echo 102 > /proc/sys/vm/hugetlb_shm_group  
  
# Increase the amount of shmem permitted per segment  
# (12G in this case).  
echo 1560281088 > /proc/sys/kernel/shmmmax  
  
# Increase total amount of shared memory. The value  
# is the number of pages. At 4KB/page, 4194304 = 16GB.  
echo 4194304 > /proc/sys/kernel/shmall
```

For MySQL usage, you normally want the value of `shmmmax` to be close to the value of `shmlll`.

To verify the large page configuration, check `/proc/meminfo` again as described previously. Now you should see some nonzero values:

```
shell> cat /proc/meminfo | grep -i huge  
HugePages_Total:      20  
HugePages_Free:       20  
HugePages_Rsvd:        0  
HugePages_Surp:        0  
Hugepagesize:     4096 kB
```

The final step to make use of the `hugetlb_shm_group` is to give the `mysql` user an “unlimited” value for the memlock limit. This can be done either by editing `/etc/security/limits.conf` or by adding the following command to your `mysqld_safe` script:

```
ulimit -l unlimited
```

Adding the `ulimit` command to `mysqld_safe` causes the `root` user to set the memlock limit to `unlimited` before switching to the `mysql` user. (This assumes that `mysqld_safe` is started by `root`.)

Large page support in MySQL is disabled by default. To enable it, start the server with the `--large-pages` option. For example, you can use the following lines in the server `my.cnf` file:

```
[mysqld]  
large-pages
```

With this option, `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`

To verify that large pages are being used, check `/proc/meminfo` again:

```
shell> cat /proc/meminfo | grep -i huge  
HugePages_Total:      20  
HugePages_Free:       20  
HugePages_Rsvd:        2  
HugePages_Surp:        0  
Hugepagesize:     4096 kB
```

## 8.12.4 Optimizing Network Use

### 8.12.4.1 How MySQL Handles Client Connections

This section describes aspects of how the MySQL server manages client connections.

- [Network Interfaces and Connection Manager Threads](#)
- [Client Connection Thread Management](#)
- [Connection Volume Management](#)
- [Administrative Connection Management](#)

#### Network Interfaces and Connection Manager Threads

The server is capable of listening for client connections on multiple network interfaces. Connection manager threads handle client connection requests on the network interfaces that the server listens to:

- On all platforms, one manager thread handles TCP/IP connection requests.
- On Unix, the same manager thread also handles Unix socket file connection requests.
- On Windows, a manager thread handles shared-memory connection requests, and another handles named-pipe connection requests.
- On all platforms, an additional network interface may be enabled to accept administrative TCP/IP connection requests. This interface can use the manager thread that handles “ordinary” TCP/IP requests, or a separate thread.

The server does not create threads to handle interfaces that it does not listen to. For example, a Windows server that does not have support for named-pipe connections enabled does not create a thread to handle them.

#### Client Connection Thread Management

Connection manager threads associate each client connection with a thread dedicated to it that handles authentication and request processing for that connection. Manager threads create a new thread when necessary but try to avoid doing so by consulting the thread cache first to see whether it contains a thread that can be used for the connection. When a connection ends, its thread is returned to the thread cache if the cache is not full.

In this connection thread model, there are as many threads as there are clients currently connected, which has some disadvantages when server workload must scale to handle large numbers of connections. For example, thread creation and disposal becomes expensive. Also, each thread requires server and kernel resources, such as stack space. To accommodate a large number of simultaneous connections, the stack size per thread must be kept small, leading to a situation where it is either too small or the server consumes large amounts of memory. Exhaustion of other resources can occur as well, and scheduling overhead can become significant.

MySQL Enterprise Edition includes a thread pool plugin that provides an alternative thread-handling model designed to reduce overhead and improve performance. It implements a thread pool that increases server performance by efficiently managing statement execution threads for large numbers of client connections. See [Section 5.6.3, “MySQL Enterprise Thread Pool”](#).

To control and monitor how the server manages threads that handle client connections, several system and status variables are relevant. (See [Section 5.1.8, “Server System Variables”](#), and [Section 5.1.10, “Server Status Variables”](#).)

The `thread_cache_size` system variable determines the thread cache size. By default, the server autosizes the value at startup, but it can be set explicitly to override this default. A value of 0 disables caching, which causes a thread to be set up for each new connection and disposed of when the connection terminates. To enable  $N$  inactive connection threads to be cached, set `thread_cache_size` to  $N$  at server startup or at runtime. A connection thread becomes inactive when the client connection with which it was associated terminates.

To monitor the number of threads in the cache and how many threads have been created because a thread could not be taken from the cache, check the `Threads_cached` and `Threads_created` status variables.

When the thread stack is too small, this limits the complexity of the SQL statements which the server can handle, the recursion depth of stored procedures, and other memory-consuming actions. To set a stack size of  $N$  bytes for each thread, start the server with `thread_stack`, set to  $N$  at server startup.

## Connection Volume Management

To control the maximum number of clients the server permits to connect simultaneously, set the `max_connections` system variable at server startup or at runtime. It may be necessary to increase `max_connections` if more clients attempt to connect simultaneously than the server is configured to handle (see [Section B.6.2.6, “Too many connections”](#)).

`mysqld` actually permits `max_connections` + 1 client connections. The extra connection is reserved for use by accounts that have the `CONNECTION_ADMIN` or `SUPER` privilege. By granting the privilege to administrators and not to normal users (who should not need it), an administrator can connect to the server and use `SHOW PROCESSLIST` to diagnose problems even if the maximum number of unprivileged clients are connected. See [Section 13.7.6.29, “SHOW PROCESSLIST Syntax”](#). (The server also permits administrative connections on a dedicated interface. See [Administrative Connection Management](#).)

If the server refuses a connection because the `max_connections` limit is reached, it increments the `Connection_errors_max_connections` status variable.

The maximum number of connections MySQL supports (that is, the maximum value to which `max_connections` can be set) depends on several factors:

- The quality of the thread library on a given platform.
- The amount of RAM available.
- The amount of RAM is used for each connection.
- The workload from each connection.
- The desired response time.
- The number of file descriptors available.

Linux or Solaris should be able to support at least 500 to 1000 simultaneous connections routinely and as many as 10,000 connections if you have many gigabytes of RAM available and the workload from each is low or the response time target undemanding.

Increasing the `max_connections` value increases the number of file descriptors that `mysqld` requires. If the required number of descriptors are not available, the server reduces the value of `max_connections`. For comments on file descriptor limits, see [Section 8.4.3.1, “How MySQL Opens and Closes Tables”](#).

Increasing the `open_files_limit` system variable may be necessary, which may also require raising the operating system limit on how many file descriptors can be used by MySQL. Consult your operating

system documentation to determine whether it is possible to increase the limit and how to do so. See also [Section B.6.2.17, “File Not Found and Similar Errors”](#).

## Administrative Connection Management

As of MySQL 8.0.14, the server permits a TCP/IP port to be configured specifically for administrative connections. This provides an alternative to the single administrative connection that is permitted on the network interfaces used for ordinary connections even when `max_connections` connections are already established (see [Connection Volume Management](#)).

The administrative network interface has these characteristics:

- The interface is available only if the `admin_address` system variable is set at startup to indicate the IP address for the administrative interface. If no `admin_address` value is specified, the server maintains no administrative interface.
- The `admin_port` system variable specifies the interface TCP/IP port number (default 33062).
- There is no limit on the number of administrative connections.
- Connections are permitted only by users who have the `SERVICE_CONNECTION_ADMIN` privilege.

The `create_admin_listener_thread` system variable enables DBAs to choose at startup whether the administrative interface is implemented using the listener thread used for ordinary connections, or has its own separate thread. The default is to implement the administrative interface using the listener thread used for ordinary connections.

### 8.12.4.2 DNS Lookup Optimization and the Host Cache

The MySQL server maintains a host cache in memory that contains information about clients: IP address, host name, and error information. The Performance Schema `host_cache` table exposes the contents of the host cache so that it can be examined using `SELECT` statements. This may help you diagnose the causes of connection problems. See [Section 26.12.17.1, “The host\\_cache Table”](#).



#### Note

The server uses the host cache only for nonlocal TCP connections. It does not use the cache for TCP connections established using a loopback interface address (for example, `127.0.0.1` or `::1`), or for connections established using a Unix socket file, named pipe, or shared memory.

- [Host Cache Operation](#)
- [Host Cache Configuration](#)

#### Host Cache Operation

The server uses the host cache for several purposes:

- By caching the results of IP-to-host name lookups, the server avoids doing a Domain Name System (DNS) lookup for each client connection. Instead, for a given host, it needs to perform a lookup only for the first connection from that host.
- The cache contains information about errors that occur during the connection process. Some errors are considered “blocking.” If too many of these occur successively from a given host without a successful connection, the server blocks further connections from that host. The `max_connect_errors`

system variable determines the permitted number of successive errors before blocking occurs (see [Section B.6.2.5, “Host ‘host\\_name’ is blocked”](#)).

For each new client connection, the server uses the client IP address to check whether the client host name is in the host cache. If so, the server refuses or continues to process the connection request depending on whether or not the host is blocked. If the host is not in the cache, the server attempts to resolve the host name. First, it resolves the IP address to a host name and resolves that host name back to an IP address. Then it compares the result to the original IP address to ensure that they are the same. The server stores information about the result of this operation in the host cache. If the cache is full, the least recently used entry is discarded.

The server handles entries in the host cache like this:

1. When the first TCP client connection reaches the server from a given IP address, a new cache entry is created to record the client IP, host name, and client lookup validation flag. Initially, the host name is set to `NULL` and the flag is false. This entry is also used for subsequent client TCP connections from the same originating IP.
2. If the validation flag for the client IP entry is false, the server attempts an IP-to-host name-to-IP DNS resolution. If that is successful, the host name is updated with the resolved host name and the validation flag is set to true. If resolution is unsuccessful, the action taken depends on whether the error is permanent or transient. For permanent failures, the host name remains `NULL` and the validation flag is set to true. For transient failures, the host name and validation flag remain unchanged. (In this case, another DNS resolution attempt occurs the next time a client connects from this IP.)
3. If an error occurs while processing an incoming client connection from a given IP address, the server updates the corresponding error counters in the entry for that IP. For a description of the errors recorded, see [Section 26.12.17.1, “The host\\_cache Table”](#).

The server performs host name resolution using the `gethostbyaddr()` and `gethostbyname()` system calls.

To unblock blocked hosts, flush the host cache by executing a `FLUSH HOSTS` statement, a `TRUNCATE TABLE` statement that truncates the Performance Schema `host_cache` table, or a `mysqladmin flush-hosts` command. `FLUSH HOSTS` and `mysqladmin flush-hosts` require the `RELOAD` privilege. `TRUNCATE TABLE` requires the `DROP` privilege for the `host_cache` table.

It is possible for a blocked host to become unblocked even without flushing the host cache if activity from other hosts has occurred since the last connection attempt from the blocked host. This can occur because the server discards the least recently used cache entry to make room for a new entry if the cache is full when a connection arrives from a client IP not in the cache. If the discarded entry is for a blocked host, that host becomes unblocked.

Some connection errors are not associated with TCP connections, occur very early in the connection process (even before an IP address is known), or are not specific to any particular IP address (such as out-of-memory conditions). For information about these errors, check the `Connection_errors_xxx` status variables (see [Section 5.1.10, “Server Status Variables”](#)).

## Host Cache Configuration

The host cache is enabled by default. The `host_cache_size` system variable controls its size, as well as the size of the Performance Schema `host_cache` table that exposes the cache contents. The cache size can be set at server startup and changed at runtime. For example, to set the size to 100 at startup, put these lines in the server `my.cnf` file:



```
[mysqld]
host_cache_size=200
```

To change the size to 300 at runtime, do this:

```
SET GLOBAL host_cache_size=300;
```

Setting `host_cache_size` to 0, either at server startup or at runtime, disables the host cache. With the cache disabled, the server performs a DNS lookup every time a client connects.

Changing the cache size at runtime causes an implicit `FLUSH HOSTS` operation that clears the host cache, truncates the `host_cache` table, and unblocks any blocked hosts.

Using the `--skip-host-cache` option is similar to setting the `host_cache_size` system variable to 0, but `host_cache_size` is more flexible because it can also be used to resize, enable, and disable the host cache at runtime, not just at server startup. Starting the server with `--skip-host-cache` does not prevent changes to the value of `host_cache_size`, but such changes have no effect and the cache is not re-enabled even if `host_cache_size` is set larger than 0 at runtime.

To disable DNS host name lookups, start the server with the `--skip-name-resolve` option. In this case, the server uses only IP addresses and not host names to match connecting hosts to rows in the MySQL grant tables. Only accounts specified in those tables using IP addresses can be used. (A client may not be able to connect if no account exists that specifies the client IP address.)

If you have a very slow DNS and many hosts, you might be able to improve performance either by disabling DNS lookups with `--skip-name-resolve` or by increasing the value of `host_cache_size` to make the host cache larger.

To disallow TCP/IP connections entirely, start the server with the `--skip-networking` option.

## 8.12.5 Resource Groups

MySQL supports creation and management of resource groups, and permits assigning threads running within the server to particular groups so that threads execute according to the resources available to the group. Group attributes enable control over its resources, to enable or restrict resource consumption by threads in the group. DBAs can modify these attributes as appropriate for different workloads.

Currently, CPU time is a manageable resource, represented by the concept of “virtual CPU” as a term that includes CPU cores, hyperthreads, hardware threads, and so forth. The server determines at startup how many virtual CPUs are available, and database administrators with appropriate privileges can associate these CPUs with resource groups and assign threads to groups.

For example, to manage execution of batch jobs that need not execute with high priority, a DBA can create a `Batch` resource group, and adjust its priority up or down depending on how busy the server is. (Perhaps batch jobs assigned to the group should run at lower priority during the day and at higher priority during the night.) The DBA can also adjust the set of CPUs available to the group. Groups can be enabled or disabled to control whether threads are assignable to them.

The following sections describe aspects of resource group use in MySQL:

- [Resource Group Components](#)
- [Resource Group Attributes](#)
- [Resource Group Management](#)

- Resource Group Replication
- Resource Group Restrictions

**Important**

On some platforms or MySQL server configurations, resource groups are unavailable or have limitations. In particular, Linux systems might require a manual step for some installation methods. For details, see [Resource Group Restrictions](#).

## Resource Group Components

These capabilities provide the SQL interface for resource group management in MySQL:

- SQL statements enable creating, altering, and dropping resource groups, and enable assigning threads to resource groups. An optimizer hint enables assigning individual statements to resource groups.
- Resource group privileges provide control over which users can perform resource group operations.
- The `INFORMATION_SCHEMA.RESOURCE_GROUPS` table exposes information about resource group definitions and the Performance Schema `threads` table shows the resource group assignment for each thread.
- Status variables provide execution counts for each management SQL statement.

## Resource Group Attributes

Resource groups have attributes that define the group. All attributes can be set at group creation time. Some attributes are fixed at creation time; others can be modified any time thereafter.

These attributes are defined at resource group creation time and cannot be modified:

- Each group has a name. Resource group names are identifiers like table and column names, and need not be quoted in SQL statements unless they contain special characters or are reserved words. Group names are not case sensitive and may be up to 64 characters long.
- Each group has a type, which is either `SYSTEM` or `USER`. The resource group type affects the range of priority values assignable to the group, as described later. This attribute together with the differences in permitted priorities enables system threads to be identified so as to protect them from contention for CPU resources against user threads.

System and user threads correspond to background and foreground threads as listed in the Performance Schema `threads` table.

These attributes are defined at resource group creation time and can be modified any time thereafter:

- The CPU affinity is the set of virtual CPUs the resource group can use. An affinity can be any nonempty subset of the available CPUs. If a group has no affinity, it can use all available CPUs.
- The thread priority is the execution priority for threads assigned to the resource group. Priority values range from -20 (highest priority) to 19 (lowest priority). The default priority is 0, for both system and user groups.

System groups are permitted a higher priority than user groups, ensuring that user threads never have a higher priority than system threads:

- For system resource groups, the permitted priority range is -20 to 0.

- For user resource groups, the permitted priority range is 0 to 19.
- Each group can be enabled or disabled, affording administrators control over thread assignment. Threads can be assigned only to enabled groups.

## Resource Group Management

By default, there is one system group and one user group, named `SYS_default` and `USR_default`, respectively. These default groups cannot be dropped and their attributes cannot be modified. Each default group has no CPU affinity and priority 0.

Newly created system and user threads are assigned to the `SYS_default` and `USR_default` groups, respectively.

For user-defined resource groups, all attributes are assigned at group creation time. After a group has been created, its attributes can be modified, with the exception of the name and type attributes.

To create and manage user-defined resource groups, use these SQL statements:

- `CREATE RESOURCE GROUP` creates a new group. See [Section 13.7.2.2, “CREATE RESOURCE GROUP Syntax”](#).
- `ALTER RESOURCE GROUP` modifies an existing group. See [Section 13.7.2.1, “ALTER RESOURCE GROUP Syntax”](#).
- `DROP RESOURCE GROUP` drops an existing group. See [Section 13.7.2.3, “DROP RESOURCE GROUP Syntax”](#).

Those statements require the `RESOURCE_GROUP_ADMIN` privilege.

To manage resource group assignments, use these capabilities:

- `SET RESOURCE GROUP` assigns threads to a group. See [Section 13.7.2.4, “SET RESOURCE GROUP Syntax”](#).
- The `RESOURCE_GROUP` optimizer hint assigns individual statements to a group. See [Section 8.9.3, “Optimizer Hints”](#).

Those operations require the `RESOURCE_GROUP_ADMIN` or `RESOURCE_GROUP_USER` privilege.

Resource group definitions are stored in the `resource_groups` data dictionary table so that groups persist across server restarts. Because `resource_groups` is part of the data dictionary, it is not directly accessible by users. Resource group information is available using the `INFORMATION_SCHEMA.RESOURCE_GROUPS` table, which is implemented as a view on the data dictionary table. See [Section 25.22, “The INFORMATION\\_SCHEMA RESOURCE\\_GROUPS Table”](#).

Initially, the `RESOURCE_GROUPS` table has these rows describing the default groups:

```
mysql> SELECT * FROM INFORMATION_SCHEMA.RESOURCE_GROUPS\G
***** 1. row *****
  RESOURCE_GROUP_NAME: USR_default
  RESOURCE_GROUP_TYPE: USER
  RESOURCE_GROUP_ENABLED: 1
    VCPU_IDS: 0-3
  THREAD_PRIORITY: 0
***** 2. row *****
```

```

RESOURCE_GROUP_NAME: SYS_default
RESOURCE_GROUP_TYPE: SYSTEM
RESOURCE_GROUP_ENABLED: 1
VCPU_IDS: 0-3
THREAD_PRIORITY: 0

```

The `THREAD_PRIORITY` values are 0, indicating the default priority. The `VCPU_IDS` values show a range comprising all available CPUs. For the default groups, the displayed value varies depending on the system on which the MySQL server runs.

Earlier discussion mentioned a scenario involving a resource group named `Batch` to manage execution of batch jobs that need not execute with high priority. To create such a group, use a statement similar to this:

```

CREATE RESOURCE GROUP Batch
  TYPE = USER
  VCPU = 2-3          -- assumes a system with at least 4 CPUs
  THREAD_PRIORITY = 10;

```

To verify that the resource group was created as expected, check the `RESOURCE_GROUPS` table:

```

mysql> SELECT * FROM INFORMATION_SCHEMA.RESOURCE_GROUPS
      WHERE RESOURCE_GROUP_NAME = 'Batch'\G
*****
*** 1. row ****
RESOURCE_GROUP_NAME: Batch
RESOURCE_GROUP_TYPE: USER
RESOURCE_GROUP_ENABLED: 1
VCPU_IDS: 2-3
THREAD_PRIORITY: 10

```

If the `THREAD_PRIORITY` value is 0 rather than 10, check whether your platform or system configuration limits the resource group capability; see [Resource Group Restrictions](#).

To assign a thread to the `Batch` group, do this:

```
SET RESOURCE GROUP Batch FOR thread_id;
```

Thereafter, statements in the named thread execute with `Batch` group resources.

If a session's own current thread should be in the `Batch` group, execute this statement within the session:

```
SET RESOURCE GROUP Batch;
```

Thereafter, statements in the session execute with `Batch` group resources.

To execute a single statement using the `Batch` group, use the `RESOURCE_GROUP` optimizer hint:

```
INSERT /*+ RESOURCE_GROUP(Batch) */ INTO t2 VALUES(2);
```

Threads assigned to the `Batch` group execute with its resources, which can be modified as desired:

- For times when the system is highly loaded, decrease the number of CPUs assigned to the group, lower its priority, or (as shown) both:

```

ALTER RESOURCE GROUP Batch
  VCPU = 3
  THREAD_PRIORITY = 19;

```

- For times when the system is lightly loaded, increase the number of CPUs assigned to the group, raise its priority, or (as shown) both:

```
ALTER RESOURCE GROUP Batch
  VCPU = 0-3
  THREAD_PRIORITY = 0;
```

## Resource Group Replication

Resource group management is local to the server on which it occurs. Resource group SQL statements and modifications to the `resource_groups` data dictionary table are not written to the binary log and are not replicated.

## Resource Group Restrictions

On some platforms or MySQL server configurations, resource groups are unavailable or have limitations:

- Resource groups are unavailable if the thread pool plugin is installed.
- Resource groups are unavailable on macOS, which provides no API for binding CPUs to a thread.
- On FreeBSD and Solaris, resource group thread priorities are ignored. (Effectively, all threads run at priority 0.) Attempts to change priorities result in a warning:

```
mysql> ALTER RESOURCE GROUP abc THREAD_PRIORITY = 10;
Query OK, 0 rows affected, 1 warning (0.18 sec)

mysql> SHOW WARNINGS;
+-----+-----+
| Level | Code | Message
+-----+-----+
| Warning | 4560 | Attribute thread_priority is ignored (using default value).. |
+-----+-----+
```

- On Linux, resource groups thread priorities are ignored unless the `CAP_SYS_NICE` capability is set. Granting `CAP_SYS_NICE` capability to a process enables a range of privileges; consult <http://man7.org/linux/man-pages/man7/capabilities.7.html> for the full list. Please be careful when enabling this capability.

On Linux platforms using systemd and kernel support for Ambient Capabilities (Linux 4.3 or newer), the recommended way to enable `CAP_SYS_NICE` capability is to modify the MySQL service file and leave the `mysqld` binary unmodified. To adjust the service file for MySQL, use this procedure:

- Run the appropriate command for your platform:

- Oracle Linux, Red Hat, and Fedora systems:

```
shell> sudo systemctl edit mysqld
```

- SUSE, Ubuntu, and Debian systems:

```
shell> sudo systemctl edit mysql
```

- Using an editor, add the following text to the service file:

```
[Service]
AmbientCapabilities=CAP_SYS_NICE
```

3. Restart the MySQL service.

If you cannot enable the `CAP_SYS_NICE` capability as just described, it can be set manually using the `setcap` command, specifying the path name to the `mysqld` executable (this requires `sudo` access). You can check the capabilities using `getcap`. For example:

```
shell> sudo setcap cap_sys_nice+ep /path/to/mysqld
shell> getcap /path/to/mysqld
/path/to/mysqld = cap_sys_nice+ep
```

As a safety measure, restrict execution of the `mysqld` binary to the `root` user and users with `mysql` group membership:

```
shell> sudo chown root:mysql /path/to/mysqld
shell> sudo chmod 0750 /path/to/mysqld
```



### Important

If manual use of `setcap` is required, it must be performed after each reinstall.

- On Windows, threads run at one of five thread priority levels. The resource group thread priority range of -20 to 19 maps onto those levels as indicated in the following table.

**Table 8.3 Resource Group Thread Priority on Windows**

Priority Range	Windows Priority Level
-20 to -10	<code>THREAD_PRIORITY_HIGHEST</code>
-9 to -1	<code>THREAD_PRIORITY_ABOVE_NORMAL</code>
0	<code>THREAD_PRIORITY_NORMAL</code>
1 to 10	<code>THREAD_PRIORITY_BELOW_NORMAL</code>
11 to 19	<code>THREAD_PRIORITY_LOWEST</code>

## 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 — Load Emulation Client”](#).
- You can also try benchmarking packages such as SysBench and DBT2, available at <https://launchpad.net/sysbench>, and <http://osldbt.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 26, “MySQL Performance Schema”](#) for details.

## 8.14 Examining Thread Information

When you are attempting to ascertain what your MySQL server is doing, it can be helpful to examine the process list, which is the set of threads currently executing within the server. Process list information is available from these sources:

- The `SHOW [FULL] PROCESSLIST` statement: [Section 13.7.6.29, “SHOW PROCESSLIST Syntax”](#)
- The `SHOW PROFILE` statement: [Section 13.7.6.31, “SHOW PROFILES Syntax”](#)
- The `INFORMATION_SCHEMA PROCESSLIST` table: [Section 25.19, “The INFORMATION\\_SCHEMA PROCESSLIST Table”](#)
- The `mysqladmin processlist` command: [Section 4.5.2, “mysqladmin — Client for Administering a MySQL Server”](#)
- The Performance Schema `threads` table, stage tables, and lock tables: [Section 26.12.17, “Performance Schema Miscellaneous Tables”](#), [Section 26.12.5, “Performance Schema Stage Event Tables”](#), [Section 26.12.12, “Performance Schema Lock Tables”](#).

Access to `threads` does not require a mutex and has minimal impact on server performance. `INFORMATION_SCHEMA.PROCESSLIST` and `SHOW PROCESSLIST` have negative performance consequences because they require a mutex. `threads` also shows information about background threads, which `INFORMATION_SCHEMA.PROCESSLIST` and `SHOW PROCESSLIST` do not. This means that `threads` can be used to monitor activity the other thread information sources cannot.

You can always view information about your own threads. To view information about threads being executed for other accounts, you must have the `PROCESS` privilege.

Each process list entry contains several pieces of information:

- `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 is 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.

- `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 thread running on a slave that is processing events from the master, the thread time is set to the time found in the events and thus reflects current time on the master and not the slave.
- `Info` contains the text of the statement being executed by the thread, or `NULL` if it is not executing one. By default, this value contains only the first 100 characters of the statement. To see the complete statements, use `SHOW FULL PROCESSLIST`.
- The `sys` schema `processlist` view, which presents information from the Performance Schema `threads` table in a more accessible format: [Section 27.4.3.22, “The processlist and x\\$processlist Views”](#)
- The `sys` schema `session` view, which presents information about user sessions (like the `sys` schema `processlist` view, but with background processes filtered out): [Section 27.4.3.33, “The session and x\\$session Views”](#)

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.

### 8.14.1 Thread Command Values

A thread can have any of the following `Command` values:

- `Binlog Dump`

This is a thread on a master server for sending binary log contents to a slave server.

- `Change user`

The thread is executing a change-user operation.

- `Close stmt`

The thread is closing a prepared statement.

- `Connect`

A replication slave is connected to its master.

- `Connect Out`

A replication slave is connecting to its master.

- `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`  
The thread is executing a statement.
- `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 slave 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.

- `Table Dump`

The thread is sending table contents to a slave server.

- `Time`

Unused.

## 8.14.2 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.

- `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.

- `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 26.12.5, “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.

- `altering table`

The server is in the process of executing an in-place `ALTER TABLE`.

- `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 will be `Copying to tmp table on disk`.

- `committing alter table to storage engine`

The server has finished an in-place `ALTER TABLE` and is committing the result.

- `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.

- `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.

For the `end` state, the following operations could be happening:

- Writing an event to the binary log
- Freeing memory buffers, including for blobs
- `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.

- `NULL`

This state is used for the `SHOW PROCESSLIST` state.

- `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.

- `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.

- `preparing for alter table`

The server is preparing to execute an in-place `ALTER TABLE`.

- `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](#)

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.

- [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.

- [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 will have 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 global read lock`

`FLUSH TABLES WITH READ LOCK` is waiting for a global read lock or the global `read_only` system variable is being set.

- `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 26.12.12, “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.

### 8.14.3 Replication Master Thread States

The following list shows the most common states you may see in the `State` column for the master's `Binlog Dump` thread. If you see no `Binlog Dump` threads on a master server, this means that replication is not running; that is, that no slaves are currently connected.

- `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 slave.

- `Master has sent all binlog to slave; waiting for more updates`

The thread has read all remaining updates from the binary logs and sent them to the slave. The thread is now idle, waiting for new events to appear in the binary log resulting from new updates occurring on the master.

- `Sending binlog event to slave`

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 slave.

- `Waiting to finalize termination`

A very brief state that occurs as the thread is stopping.

#### 8.14.4 Replication Slave I/O Thread States

The following list shows the most common states you see in the `State` column for a slave server I/O thread. This state also appears in the `Slave_IO_State` column displayed by `SHOW SLAVE STATUS`, so you can get a good view of what is happening by using that statement.

- `Checking master version`

A state that occurs very briefly, after the connection to the master is established.

- `Connecting to master`

The thread is attempting to connect to the master.

- `Queueing master 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 master.

- `Reconnecting after a failed master event read`

The thread is trying to reconnect to the master. When connection is established again, the state becomes `Waiting for master to send event`.

- `Registering slave on master`

A state that occurs very briefly after the connection to the master is established.

- `Requesting binlog dump`

A state that occurs very briefly, after the connection to the master is established. The thread sends to the master 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 slave thread is waiting for older worker threads to commit if `slave_preserve_commit_order` is enabled.

- `Waiting for master to send event`

The thread has connected to the master and is waiting for binary log events to arrive. This can last for a long time if the master is idle. If the wait lasts for `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`

The initial state before `Connecting to master`.

- `Waiting for slave 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`

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 thread is waiting until the SQL 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 MASTER TO` statement.

- `Waiting to reconnect after a failed master event read`

An error occurred while reading (due to disconnection). The thread is sleeping for the number of seconds set by the `CHANGE MASTER TO` statement (default 60) before attempting to reconnect.

## 8.14.5 Replication Slave SQL Thread States

The following list shows the most common states you may see in the `State` column for a slave server SQL thread:

- `Killing slave`

The thread is processing a `STOP SLAVE` statement.

- `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 slave will read 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 slave will read rows. This state can only be encountered if the original `LOAD DATA` statement was logged by a master 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`

The thread has processed all events in the relay log files, and is now waiting for the I/O thread to write new events to the relay log.

- Waiting for an event from Coordinator

Using the multithreaded slave (`slave_parallel_workers` is greater than 1), one of the slave worker threads is waiting for an event from the coordinator thread.

- Waiting for slave mutex on exit

A very brief state that occurs as the thread is stopping.

- Waiting for Slave Workers to free pending events

This waiting action occurs when the total size of events being processed by Workers exceeds the size of the `slave_pending_jobs_size_max` system variable. The Coordinator resumes scheduling when the size drops below this limit. This state occurs only when `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

The SQL thread has read an event but is waiting for the slave delay to lapse. This delay is set with the `MASTER_DELAY` option of `CHANGE MASTER TO`.

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.

## 8.14.6 Replication Slave Connection Thread States

These thread states occur on a replication slave but are associated with connection threads, not with the I/O or SQL threads.

- Changing master

The thread is processing a `CHANGE MASTER TO` statement.

- Killing slave

The thread is processing a `STOP SLAVE` 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`.

## 8.14.7 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).

### 8.14.8 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 will execute 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

9.1 Literal Values .....	1587
9.1.1 String Literals .....	1587
9.1.2 Numeric Literals .....	1590
9.1.3 Date and Time Literals .....	1590
9.1.4 Hexadecimal Literals .....	1593
9.1.5 Bit-Value Literals .....	1595
9.1.6 Boolean Literals .....	1597
9.1.7 NULL Values .....	1597
9.2 Schema Object Names .....	1597
9.2.1 Identifier Qualifiers .....	1600
9.2.2 Identifier Case Sensitivity .....	1601
9.2.3 Mapping of Identifiers to File Names .....	1603
9.2.4 Function Name Parsing and Resolution .....	1604
9.3 Keywords and Reserved Words .....	1608
9.4 User-Defined Variables .....	1637
9.5 Expressions .....	1640
9.6 Comment Syntax .....	1645

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
- 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: