8.8.2 EXPLAIN Output Format

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

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

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

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

EXPLAIN Output Columns

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

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

Table 8.1 EXPLAIN Output Columns

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

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

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

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

- `select_type` (JSON name: none)

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

`select_type` **Value**	JSON Name	Meaning
`SIMPLE`	None	Simple `SELECT` (not using `UNION` or subqueries)
`PRIMARY`	None	Outermost `SELECT`
`UNION`	None	Second or later `SELECT` statement in a `UNION`
`DEPENDENT UNION`	`dependent` (`true`)	Second or later `SELECT` statement in a `UNION`, dependent on outer query
`UNION RESULT`	`union_result`	Result of a `UNION`.
`SUBQUERY`	None	First `SELECT` in subquery
`DEPENDENT SUBQUERY`	`dependent` (`true`)	First `SELECT` in subquery, dependent on outer query
`DERIVED`	None	Derived table
`DEPENDENT DERIVED`	`dependent` (`true`)	Derived table dependent on another table
`MATERIALIZED`	`materialized_from_subquery`	Materialized subquery
`UNCACHEABLE SUBQUERY`	`cacheable` (`false`)	A subquery for which the result cannot be cached and must be re-evaluated for each row of the outer query
`UNCACHEABLE UNION`	`cacheable` (`false`)	The second or later select in a `UNION` that belongs to an uncacheable subquery (see `UNCACHEABLE SUBQUERY`)

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

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

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

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

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

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

  - `<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 24.3.5, “Obtaining Information About Partitions”.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

- `Extra` (JSON name: none)

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

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

EXPLAIN Join Types

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

- `system`

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

- `const`

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

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

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

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

- `eq_ref`

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

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

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

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

- `ref`

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

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

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

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

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

- `fulltext`

  The join is performed using a `FULLTEXT` index.

- `ref_or_null`

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

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

  See Section 8.2.1.15, “IS NULL Optimization”.

- `index_merge`

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

- `unique_subquery`

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

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

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

- `index_subquery`

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

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

- `range`

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

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

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

  SELECT * FROM tbl_name
    WHERE key_column BETWEEN 10 and 20;

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

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

- `index`

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

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

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

- `ALL`

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

EXPLAIN Extra Information

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

  The optimizer determined 1) that at most one row should be returned, and 2) that to produce this row, a deterministic set of rows must be read. When the rows to be read can be read during the optimization phase (for example, by reading index rows), there is no need to read any tables during query execution.

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

  Consider the following implicitly grouped query:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

EXPLAIN Output Interpretation

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

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

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

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

For this example, make the following assumptions:

- The columns being compared have been declared as follows.

Table	Column	Data Type
tt	ActualPC	CHAR(10)
tt	AssignedPC	CHAR(10)
tt	ClientID	CHAR(10)
et	EMPLOYID	CHAR(15)
do	CUSTNMBR	CHAR(15)

- The tables have the following indexes.

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

- The `tt.ActualPC` values are not evenly distributed.

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

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

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

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

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

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

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

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

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

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

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

After that modification, `EXPLAIN` produces the output shown here:

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

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

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

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

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

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

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