# CloudLimit Row

Select \* From <TableName> LIMIT 10; --Top 10 rows

SELECT \* FROM <TableName> LIMIT 1, 4;

-- starting the second row and limits the results to 4.

# Check version

SELECT VERSION();

# Types of Indexes

MySQL allows for up to 32 indexes for each table, and each index can incorporate up to 16 columns.  Indexes are best used on columns

* That are frequently used in the WHERE part of a query
* That are frequently used in an ORDER BY part of a query
* That have many different values (columns with numerous repeating values ought not to be indexed)

MySQL has three types of indexes: INDEX, UNIQUE (which requires each row to have a unique value), and PRIMARY KEY (which is just a particular UNIQUE index) and are stored in [B-trees](https://dev.mysql.com/doc/refman/8.0/en/glossary.html#glos_b_tree).

MySQL requires that foreign key columns be indexed; if you create a table with a foreign key constraint but no index on a given column, an index is created.

To display foreign key and references

SELECT TABLE\_SCHEMA, TABLE\_NAME, COLUMN\_NAME, CONSTRAINT\_NAME,REFERENCED\_TABLE\_NAME,REFERENCED\_COLUMN\_NAME

cFROM INFORMATION\_SCHEMA.KEY\_COLUMN\_USAGE WHERE TABLE\_NAME='AAMS\_Property' and REFERENCED\_TABLE\_NAME IS NOT NULL

To display All references for a table

SELECT TABLE\_SCHEMA, TABLE\_NAME, COLUMN\_NAME, CONSTRAINT\_NAME,REFERENCED\_TABLE\_NAME,REFERENCED\_COLUMN\_NAME

FROM INFORMATION\_SCHEMA.KEY\_COLUMN\_USAGE WHERE REFERENCED\_TABLE\_NAME=**'AAMS\_Property'**

### Can a foreign key be NULL and/or duplicate?

Yes, it can be NULL or duplicate.First remember a Foreign key simply requires that the value in that field must exist first in a different table (the parent table).

## Displaying INDEX Information

### To list all indexes of a specific table

SHOW INDEX FROM <*table\_name>*

To list all indexes of a schema

SELECT DISTINCT TABLE\_NAME, INDEX\_NAME,COLUMN\_NAME

FROM INFORMATION\_SCHEMA.STATISTICS WHERE TABLE\_NAME=**'AAMS\_Property**'

### ANALYZE TABLE

ANALYZE TABLE examines key distribution and stores them in INFORMATION\_SCHEMA.STATISTICS.

ANALYZE TABLE mydb.mytable

For a particular query, the query optimizer uses the stored key distribution and other factors to decide the order in which tables should be joined when you performing the join, and which index should be used for a specific table.

However, the key distributions can be sometimes inaccurate e.g., after you have done a lot of data changes in the table including insert, delete, or update. IIf the key distribution isn't correct, the question optimizer could pick a nasty query execution arrange that will cause a severe performance issue.

If there are a high number of INSERTs, UPDATEs, and DELETEs, then you will need to ANALYZE TABLE.

### Covering Index

A *covering index* is a special case of an index in *InnoDB* where all required fields for a query are included in the index; in other words, the index itself contains the required data to execute the queries without having to execute additional reads.

#### ****When to use covering indexes****

To serve queries without additional *IO* reads on big tables. Sometimes, even avoiding extra physical random *IO* reads (the most expensive *IO*operation).The principal scenarios to use covering indexes are:

* filtering rows matching a certain condition (WHERE clauses)
* grouping data (GROUP BY clauses)
* sorting data in the same order of the covering index (ORDER BY clauses)
* projecting data (SELECT clauses)

#### Drawbacks

* *Covering indexes* duplicate data of the original table, so if the table is already large, the index will be also. In this case, space usage might become an issue.
* *Covering indexes* are effective as long as the queries don’t add additional columns (for projections, calculations, filtering, sorting, etc). This point might become an issue when the queries change quickly in the near future and MySQL decides to stop using the current *covering indexes*.

**Example**

CREATE TABLE big\_table(  
  id int primary key auto\_increment,  
  field01 int,  
  field02 int,  
  field03 int,  
  field04 decimal,  
  field05 int  
) engine=innodb;

Let’s suppose that we have the following query in our application as shown below

SELECT sum(field04) FROM big\_table WHERE field01=1 GROUP BY field03;

When the table is very big, a query like this can spend several seconds or even minutes to execute. If we want to optimize this query, we can create a *covering index* for it, so there's no need to hit the big table to fetch the rows; just fetching the data fields from the index it-self. This approach can speed up some of our queries by an order of magnitude.

In order to create a covering index for this query, the index must cover the WHERE/GROUP BY and SELECT clauses as shown

ALTER TABLE big\_table ADD INDEX (field01, field03, field04);

It's important to choose the correct order of the columns in the index to serve query correctly. The general rule is to choose the columns for filtering first (WHERE clause with equality conditions) e.g field01, then sorting/grouping (ORDER BY and GROUP BY clauses) e.g field03 and finally the data projection (SELECT clause) e.g field04.

we can verify if InnoDB is really using the covering index by checking the "Extra" field. we should actually see the "Using Index" value in the “Extra” field.

### INDEX on a VARCHAR field

<https://dba.stackexchange.com/questions/35821/possible-index-on-a-varchar-field-in-mysql>

#### Column Prefix Key Parts

For string columns, indexes can be created that use only the leading part of column values, using ***col\_name***(***length***) syntax to specify an index prefix length.

CREATE INDEX part\_of\_name ON customer (name(10));

MySQL enables you to define prefixed index which means you define first N characters from original string to be indexed, and the trick is to choose a number N that’s long enough to give good selectivity, but short enough to save space.

CREATE TABLE mytable

( id int not null auto\_increment,

fName varchar(255) not null,

Lname varchar(255) not null,

key (fName(N))

);

we can make index on column last\_name with indexing only first 10 characters. In table definition column last\_name is defined as VARCHAR(16), and this means we have saved 6 bytes (or more if there are UTF8 characters in the last name) per entry. In this table there are 1637 distinct values multiplied by 6 bytes is about 9KB, and imagine how this number would grow if our table contains million of rows.

* Prefixes can be specified for [CHAR](https://dev.mysql.com/doc/refman/8.0/en/char.html), [VARCHAR](https://dev.mysql.com/doc/refman/8.0/en/char.html), [BINARY](https://dev.mysql.com/doc/refman/8.0/en/binary-varbinary.html), and [VARBINARY](https://dev.mysql.com/doc/refman/8.0/en/binary-varbinary.html) key parts.

#### Full-Text Indexes

<https://dev.mysql.com/doc/refman/8.0/en/create-index.html>

It can include only [CHAR](https://dev.mysql.com/doc/refman/8.0/en/char.html), [VARCHAR](https://dev.mysql.com/doc/refman/8.0/en/char.html), and [TEXT](https://dev.mysql.com/doc/refman/8.0/en/blob.html) columns. Indexing always happens over the entire column; column prefix indexing is not supported and any prefix length is ignored if specified.

only one full-text index can be created on each table so if you have more than one column you need to be indexed the columns have to be moved to separate tables.

In specific the LIKE search which includes wildcards can not make use of any kind of index.So we create Full Text index wherein MySQL split the text into words and make an index over the words and not the whole text.

#### Functional Key Parts(MySQL 8.0.13 or later)

CREATE INDEX idx1 ON t1 ((col1 + col2));

CREATE INDEX idx2 ON t1 ((col1 + col2), (col1 - col2), col1);

CREATE INDEX idx2 ON t1 ((ABS(col1)))

* In index definitions, enclose expressions within parentheses to distinguish them from columns or column prefixes.

##### Index on DateTime

<https://forums.percona.com/t/index-on-datetime-column/241/2>

SELECT news\_id,news\_title,news\_hit FROM news WHERE DATE(news\_date) = ‘2007-03-14’

MySQL won’t/can’t use indexes where you apply a function to a column in a where condition. In your case it’s the DATE() function that’s stopping the query use an index.

Re-write the query as below which would use the index on news\_date.

SELECT news\_id,news\_title,news\_hit FROM news WHERE news\_date between ‘2007-03-14 00:00:00’ and ‘2007-03-15 00:00:00’

Else we can create functional index

**alter table news add index idx\_date\_createdon((Date(news\_date)));**

But there are a few restrictions on creating a functional index.

* Primary key cannot be included in functional key parts.
* Spatial and full text indexes cannot have functional key parts

To drop the columns containing functional index, we need to remove index first before dropping the column else it will throw an error.

alter table app\_user drop column createdOn;

ERROR 3755 (HY000): Cannot drop column 'createdOn' because it is used by a functional index. In order to drop the column, you must remove the functional index.

#### Unique Indexes

A UNIQUE index creates a constraint such that all values in the index must be distinct. An error occurs if you try to add a new row with a key value that matches an existing row. If you specify a prefix value for a column in a UNIQUE index, the column values must be unique within the prefix length. A UNIQUE index permits multiple NULL values for columns that can contain NULL.

# System table

|  |  |  |
| --- | --- | --- |
| TABLE\_SCHEMA | TABLE\_NAME | Description |
| information\_schema | CHARACTER\_SETS |  |
| information\_schema | COLLATIONS |  |
| information\_schema | COLLATION\_CHARACTER\_SET\_APPLICABILITY |  |
| information\_schema | COLUMNS | All column list |
| information\_schema | COLUMN\_PRIVILEGES |  |
| information\_schema | ENGINES |  |
| information\_schema | EVENTS |  |
| information\_schema | FILES |  |
| information\_schema | GLOBAL\_STATUS |  |
| information\_schema | GLOBAL\_VARIABLES |  |
| information\_schema | KEY\_COLUMN\_USAGE | Foriegn Key Info |
| information\_schema | OPTIMIZER\_TRACE |  |
| information\_schema | PARAMETERS |  |
| information\_schema | PARTITIONS |  |
| information\_schema | PLUGINS |  |
| information\_schema | PROCESSLIST |  |
| information\_schema | PROFILING |  |
| information\_schema | REFERENTIAL\_CONSTRAINTS |  |
| information\_schema | ROUTINES |  |
| information\_schema | SCHEMATA |  |
| information\_schema | SCHEMA\_PRIVILEGES |  |
| information\_schema | SESSION\_STATUS |  |
| information\_schema | SESSION\_VARIABLES |  |
| information\_schema | STATISTICS | Index Info |
| information\_schema | TABLES |  |
| information\_schema | TABLESPACES |  |
| information\_schema | TABLE\_CONSTRAINTS |  |
| information\_schema | TABLE\_PRIVILEGES |  |
| information\_schema | TRIGGERS |  |
| information\_schema | USER\_PRIVILEGES |  |
| information\_schema | VIEWS |  |

## Table Size

The INFORMATION\_SCHEMA.TABLES table contains around 20 columns, but for the purpose of determining the amount of disk space used by tables, we’ll focus on two columns in particular: DATA\_LENGTH and INDEX\_LENGTH.

* DATA\_LENGTH is the length (or size) of all data in the table (in bytes).
* INDEX\_LENGTH is the length (or size) of the index file for the table (also in bytes).

SELECT TABLE\_NAME AS `Table`,

ROUND((DATA\_LENGTH + INDEX\_LENGTH) / 1024 / 1024) AS `Size (MB)`

FROM information\_schema.TABLES WHERE TABLE\_SCHEMA = "bookstore"

## ALTER command to add and drop INDEX

There are four types of statements for adding indexes to a table −

* **ALTER TABLE tbl\_name ADD PRIMARY KEY (column\_list)** − This statement adds a **PRIMARY KEY**, which means that the indexed values must be unique and cannot be NULL.
* **ALTER TABLE <tbl\_name> ADD UNIQUE <index\_name> (column\_list)** − This statement creates an index for which the values must be unique (except for the NULL values, which may appear multiple times).
* **ALTER TABLE <tbl\_name> ADD INDEX <index\_name> (column\_list)**  − This adds an ordinary index in which any value may appear more than once.
* **ALTER TABLE tbl\_name ADD FULLTEXT index\_name (column\_list)** − This creates a special FULLTEXT index that is used for text-searching purposes.
* DROP INDEX <indexName> ON <tableName>
* **ALTER TABLE** <table\_name> **DROP INDEX** <index\_name>;

# List All table

SHOW FULL TABLES WHERE Table\_Type='BASE TABLE'

# To see View Definition

<https://dev.mysql.com/doc/refman/8.0/en/information-schema-views-table.html>

## Option1

SHOW CREATE VIEW view\_name

## Option2

SELECT VIEW\_DEFINITION FROM INFORMATION\_SCHEMA.VIEWS WHERE TABLE\_SCHEMA = 'Appian' AND TABLE\_NAME = 'AASC\_WorkRequestTasksView'

Use of [SHOW CREATE VIEW](https://dev.mysql.com/doc/refman/8.0/en/show-create-view.html) requires the [SHOW VIEW](https://dev.mysql.com/doc/refman/8.0/en/privileges-provided.html#priv_show-view) privilege, and the [SELECT](https://dev.mysql.com/doc/refman/8.0/en/privileges-provided.html#priv_select) privilege for the view in question.

### Cloud MySql

Click on view🡪Goto Structure Tab🡪in below, Click on ‘Edit view’🡪click on format🡪copy view definition

# Tuning MySQL

<https://dev.mysql.com/doc/refman/5.7/en/select-optimization.html>

To make a slow SELECT ... WHERE query faster, the first thing to check is whether you can add an [index](https://dev.mysql.com/doc/refman/5.7/en/glossary.html#glos_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](https://dev.mysql.com/doc/refman/5.7/en/glossary.html#glos_join) and [foreign keys](https://dev.mysql.com/doc/refman/5.7/en/glossary.html#glos_foreign_key). You can use the [EXPLAIN](https://dev.mysql.com/doc/refman/5.7/en/explain.html) statement to determine which indexes are used for a [SELECT](https://dev.mysql.com/doc/refman/5.7/en/select.html). The [DESCRIBE](https://dev.mysql.com/doc/refman/5.7/en/describe.html) and [EXPLAIN](https://dev.mysql.com/doc/refman/5.7/en/explain.html) statements are synonyms. In practice, the [DESCRIBE](https://dev.mysql.com/doc/refman/5.7/en/describe.html) keyword is more often used to obtain information about table structure, whereas [EXPLAIN](https://dev.mysql.com/doc/refman/5.7/en/explain.html) is used to obtain a query execution plan

* Minimize the number of [full table scans](https://dev.mysql.com/doc/refman/5.7/en/glossary.html#glos_full_table_scan) in your queries, particularly for big tables.
* Keep table statistics up to date by using the [ANALYZE TABLE](https://dev.mysql.com/doc/refman/5.7/en/analyze-table.html) statement periodically, so the optimizer has the information needed to construct an efficient execution plan.
* Deal with locking issues, where the speed of your query might be affected by other sessions accessing the tables at the same time.
* 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](https://dev.mysql.com/doc/refman/5.7/en/char.html) and [CHAR](https://dev.mysql.com/doc/refman/5.7/en/char.html) 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.
* 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.
* 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
* Create joins with INNER JOIN (not WHERE)

SELECT Customers.CustomerID, Customers.Name, Sales.LastSaleDate

FROM Customers, Sales WHERE Customers.CustomerID = Sales.CustomerID

In this example, if we had 1,000 customers with 1,000 total sales, the query would first generate 1,000,000 results, then filter for the 1,000 records where CustomerID is correctly joined.

SELECT Customers.CustomerID, Customers.Name, Sales.LastSaleDate

FROM Customers INNER JOIN Sales ON Customers.CustomerID = Sales.CustomerID

The database would only generate the 1,000 desired records where CustomerID is equal.

## When Mysql uses index

indexing can significantly improve the performance of a query. In general, mySQL queries benefit from data indexing in the following scenarios:

* finding rows that match a WHERE clause
* removing rows that do not match a WHERE clause condition
* row retrieval from other related tables when [performing joins](https://www.vertabelo.com/blog/learning-sql-joins-using-real-life-situations/)
* finding the minimum/maximum value for a specific column, as in our example above
* sorting or grouping the records of a table

it’s still a good idea to create basic indexes, but **do not just index all fields in every table**, as indexes use up memory and have a performance hit on your insert/update/delete queries. So creating a bunch of indexes you are actually not using will be bad for the performance of your database. It is also a good practice to visit your index list from time to time in the application development flow. So you can remove the indexes you are not using or add ones that’ll speed up your queries.

## Order By Optimization

<https://dev.mysql.com/doc/refman/5.7/en/limit-optimization.html>

The filesort operation used when an index cannot be used. MySQL may have to perform a “filesort” operation when a query uses the “order by” clause.  An ORDER BY with and without LIMIT may return rows in different orders.

### LIMIT Query Optimization

<https://www.percona.com/blog/2006/09/01/mysql-order-by-limit-performance-optimization/>

If we have a “LIMIT” clause in our query (and the limit is relatively small (i.e. LIMIT 10 or LIMIT 100) compared to the amount of rows in the table), MySQL can avoid using a filesort and can utilize index instead. In general this type of ORDER BY looks like SELECT ….. WHERE [conditions] ORDER BY [sort] LIMIT N,M

**Example 1**

SELECT \* FROM sites ORDER BY date\_created DESC LIMIT 10;

I would use index on (date\_created) to get a result set very fast.

*SELECT \* FROM sites WHERE category\_id=5 ORDER BY date\_created DESC LIMIT 10;*

In this case index by date\_created may also work but it might not be the most efficient – If it is a rare category large portion of table may be scanned to find 10 rows. So index on (category\_id, date\_created) will be a better idea.

SELECT \* FROM sites WHERE category\_id in (5,10,12) ORDER BY date\_created DESC LIMIT 10;

Even though it looks quite similar to the previous one it is a lot different as there are multiple category\_id values in the list now so index on (category\_id, date\_created) can’t be used directly. Index on date\_created separately would still work. This should be splitted into multiple unions.

**Another Workaround**

If you have to sort thousands of rows, you may get better results by using a small derived table, especially

SELECT t.\* FROM (SELECT date\_created FROM sites WHERE category\_id in (5,10,12)) tmp

JOIN sites t USING (date\_created) ORDER BY t. date\_created DESC

**Example 2**

If we have a “LIMIT” clause in our query (and the limit is relatively small (i.e. LIMIT 10 or LIMIT 100) compared to the amount of rows in the table), MySQL can avoid using a filesort and can utilize index instead.

mysql> explain select \* from ontime\_2012 where dayofweek in (6,7) order by DepDelayMinutes desc limit 10

We can create an index on DepDelayMinutes fields only and run the explain below (note the query with LIMIT 10):

**Example 3**

<https://www.percona.com/blog/2006/08/14/mysql-followup-on-union-for-query-optimization-query-profiling/>

mysql> explain select \* from people where age=18 order by last\_online desc limit 10;

+----+-------------+--------+------+---------------+------+---------+-------+-------+-------------+

| id | select\_type | table | type | possible\_keys | key | key\_len | ref | rows | Extra |

+----+-------------+--------+------+---------------+------+---------+-------+-------+-------------+

| 1 | SIMPLE | people | ref | age | age | 1 | const | 12543 | Using where |

+----+-------------+--------+------+---------------+------+---------+-------+-------+-------------+

mysql> explain select \* from people where age in(18,19,20) order by last\_online desc limit 10;

+----+-------------+--------+-------+---------------+------+---------+------+-------+-----------------------------+

| id | select\_type | table | type | possible\_keys | key | key\_len | ref | rows | Extra |

+----+-------------+--------+-------+---------------+------+---------+------+-------+-----------------------------+

| 1 | SIMPLE | people | range | age | age | 1 | NULL | 37915 | Using where; Using filesort |

+----+-------------+--------+-------+---------------+------+---------+------+-------+-----------------------------+

We can however use UNION to avoid filesort of full table:

mysql> explain (select \* from people where age=18 order by last\_online desc limit 10) **UNION ALL** (select \* from people where age=19 order by last\_online desc limit 10) **UNION ALL** (select \* from people where age=20 order by last\_online desc limit 10) ORDER BY last\_online desc limit 10;

+----+--------------+--------------+------+---------------+------+---------+-------+-------+----------------+

| id | select\_type  | table        | type | possible\_keys | key  | key\_len | ref   | rows  | Extra          |

+----+--------------+--------------+------+---------------+------+---------+-------+-------+----------------+

|  1 | PRIMARY      | people       | ref  | age           | age  | 1       | const | 12543 | Using where    |

|  2 | UNION        | people       | ref  | age           | age  | 1       | const | 12741 | Using where    |

|  3 | UNION        | people       | ref  | age           | age  | 1       | const | 12631 | Using where    |

|NULL | UNION RESULT | <union1,2,3> | ALL  | NULL          | NULL | NULL    | NULL  |  NULL | Using filesort |

+----+--------------+--------------+------+---------------+------+---------+-------+-------+----------------+

In this case there is also filesort but it applied only to very small table which is result of union, so it is rather fast.

--

SELECT \* FROM tbl WHERE c=5 ORDER BY a,b limit 10*;*

In this case first two columns from the index can be used to satisfy order by, index, however, will not be helpful to check c=5 (unless it is index covered query). Index on (c,a,b) would work better for the query above.

* 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.
* 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.

For a query with an ORDER BY or GROUP BY and a LIMIT clause, the optimizer tries to choose an ordered index by default when it appears doing so would speed up query execution. Prior to MySQL 5.7.33, there was no way to override this behavior, even in cases where using some other optimization might be faster. Beginning with MySQL 5.7.33, it is possible to turn off this optimization by setting the [optimizer\_switch](https://dev.mysql.com/doc/refman/5.7/en/server-system-variables.html#sysvar_optimizer_switch) system variable's [prefer\_ordering\_index](https://dev.mysql.com/doc/refman/5.7/en/switchable-optimizations.html#optflag_prefer-ordering-index) flag to off.

### Sort in one direction

If you have ORDER BY col1, col2 it can be optimized using index, if you have  
ORDER BY col1 DESC, col2 DESC same thing, however if you would have ORDER BY col1, col2 DESC MySQL will have to use filesort. Classic for solution for this would be to have index which is sorted appropriately (ascending by col1 and descending by col2) but MySQL can’t do it at this point. Workaround which can be currently used is separate column which holds reverse values, so you can do ORDER BY col1, col2\_reverse instead.

### Use of Indexes to Satisfy ORDER BY

<https://dev.mysql.com/doc/refman/5.7/en/order-by-optimization.html>

Assuming that there is an index on (key\_part1, key\_part2)

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 is not likely to use the index. If SELECT \* selects only the index columns, the index is used and sorting avoided.

SELECT pk, key\_part1, key\_part2 FROM t1 ORDER BY key\_part1, key\_part2;

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 \* FROM t1 WHERE key\_part1 = constant ORDER BY 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:

**Example**

<https://www.programmersought.com/article/40942207584/>

 MySQL can only use an index, and your SQL statement WHERE and ORDER BY conditions are not the same conditions, the index is not built, then. ORDER BY then use less than the index. Using filesort there was a problem.

To solve the problem is to create a mix, including index WHERE and ORDER BY conditions.

SELECT \* FROM user u where u.id=100 order by u.update\_time

The index is idx\_user\_id (id).Once again indexed idx\_user\_id\_update\_time (id, update\_time) should be created to avoid fileSort

### 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. 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](https://dev.mysql.com/doc/refman/8.0/en/server-system-variables.html#sysvar_sort_buffer_size) system variable, rather than allocating a fixed amount of [sort\_buffer\_size](https://dev.mysql.com/doc/refman/8.0/en/server-system-variables.html#sysvar_sort_buffer_size) bytes up front, as was done prior to MySQL 8.0.12.

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

SELECT ... FROM single\_table ... ORDER BY non\_index\_column [DESC] LIMIT [M,]N;

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](https://dev.mysql.com/doc/refman/8.0/en/server-system-variables.html#sysvar_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).

### ORDER BY Execution Plan

* If the Extra column of [EXPLAIN](https://dev.mysql.com/doc/refman/8.0/en/explain.html) output does not contain Using filesort, the index is used and a filesort is not performed.
* If the Extra column of [EXPLAIN](https://dev.mysql.com/doc/refman/8.0/en/explain.html) output contains Using filesort, the index is not used and a filesort is performed.

### [Is it possible to avoid filesort?](https://dba.stackexchange.com/questions/44614/is-it-possible-to-avoid-filesort)

In some cases, MySQL cannot use indexes to resolve the ORDER BY, although it still uses indexes to find the rows that match the WHERE clause. These cases include the following:

* You use ORDER BY on different keys
* The key used to fetch the rows is not the same as the one used in the ORDER BY

## Where Clause Optimization

### Applying algebraic equivalence rules

* 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)

* HAVING is merged with WHERE if you do not use GROUP BY or aggregate functions ([COUNT()](https://dev.mysql.com/doc/refman/5.7/en/aggregate-functions.html#function_count), [MIN()](https://dev.mysql.com/doc/refman/5.7/en/aggregate-functions.html#function_min), and so on).
* 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.

Example

SELECT col1,col2 FROM table1 WHERE col3='value1' AND col4='value2'

* I have 2 separate indexes one on col3 and the other on col4. In the query only one index is used with the highest cardinality.
* If you want to use both column index then it is better use a compound index that includes both col3 and col4. Only the **left-most** part of the compound index will be used. Index **(A,B,C)**, the following combination will be used: **(A)**; **(AB)** and **(A,B,C)**.

If only col3 is used in the WHERE clause then

index myindex (col3, col4) <<-- will work with your example.

index myindex (col4, col3) <<-- will not work.

## Obtaining Table Structure Information

mysql> DESCRIBE City;

+------------+----------+------+-----+---------+----------------+

| Field | Type | Null | Key | Default | Extra |

+------------+----------+------+-----+---------+----------------+

| Id | int(11) | NO | PRI | NULL | auto\_increment |

| Name | char(35) | NO | | | |

| Country | char(3) | NO | UNI | | |

| District | char(20) | YES | MUL | | |

| Population | int(11) | NO | | 0 | |

+------------+----------+------+-----+---------+----------------+

cardinality  : In **MySQL**, the term **cardinality** refers to the uniqueness of data values that can be put into columns. It is a kind of property which influences the ability to search, cluster and sort data.

## Obtaining Execution Plan Information

The [**EXPLAIN**](https://dev.mysql.com/doc/refman/5.7/en/explain.html) statement provides information about how MySQL executes statements:

[EXPLAIN](https://dev.mysql.com/doc/refman/5.7/en/explain.html) works with [SELECT](https://dev.mysql.com/doc/refman/5.7/en/select.html), [DELETE](https://dev.mysql.com/doc/refman/5.7/en/delete.html), [INSERT](https://dev.mysql.com/doc/refman/5.7/en/insert.html), [REPLACE](https://dev.mysql.com/doc/refman/5.7/en/replace.html), and [UPDATE](https://dev.mysql.com/doc/refman/5.7/en/update.html) statements.

mysql> explain select customer\_id, customer\_name from customers where customer\_id='140385';

+----+-------------+-----------+------------+------+---------------+------+---------+------+------+----------+-------------+

| id | select\_type | table | partitions | type | possible\_keys | key | key\_len | ref | rows | filtered

+----+-------------+-----------+------------+------+---------------+------+---------+------+------+----------+-------------+

| 1 | SIMPLE | customers | NULL | ALL | NULL | NULL | NULL | NULL | 500 | 10.00

+----+-------------+-----------+------------+------+---------------+------+---------+------+------+----------

First, it is clear that MySQL will conduct a full table scan because key column is '**NULL**'. Second, MySQL server has clearly indicated that it's going to conduct a full scan on the 500 rows in our database.

To optimize the above query, we can just add an index to the '**customer\_id**' field using the below syntax:

mysql> Create index customer\_id ON customers (customer\_Id);

Query OK, 0 rows affected (0.02 sec)

Records: 0 Duplicates: 0 Warnings: 0

If we run the explain statement one more time, we will get the below results:

+----+-------------+-----------+------------+------+---------------+-------------+---------+-------+------+----------+-------

|id| select\_type | table| partitions | type | possible\_keys | key| key\_len | ref| rows| filtered | Extra

+----+-------------+-----------+------------+------+---------------+-------------+---------+-------+------+----------+-------

|1 |SIMPLE| customers| NULL| ref | customer\_id| customer\_id | 13 | const | 1 |100.00 | NULL |

+----+-------------+-----------+------------+------+---------------+-------------+---------+-------+------+--

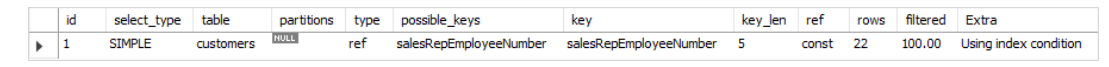
From the above explain output, it's clear that MySQL server will use our index (customer\_Id) to search the table. You can clearly see that the number of rows to scan will be 1. Although I run the above query in a table with 500 records, indexes can be very useful when you are querying a large dataset (e.g. a table with 1 million rows).

## IS NULL Optimization

MySQL can perform the same optimization on **col\_name**[**IS NULL**](https://dev.mysql.com/doc/refman/8.0/en/comparison-operators.html#operator_is-null) that it can use for **col\_name = constant\_value**.

**EXPLAIN** SELECT customerNumber,salesRepEmployeeNumber FROM customers

WHERE salesRepEmployeeNumber IS NULL;



MySQL can also optimize for the combination col = value OR col IS NULL. the EXPLAIN shows ref\_or\_null when the optimization is applied. [ref\_or\_null](https://dev.mysql.com/doc/refman/8.0/en/explain-output.html#jointype_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.

**EXPLAIN** SELECT customerNumber,salesRepEmployeeNumber FROM customers

WHERE salesRepEmployeeNumber = 1370 OR salesRepEmployeeNumber IS NULL;



The optimization can handle only one [IS NULL](https://dev.mysql.com/doc/refman/8.0/en/comparison-operators.html#operator_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);

If you have a key that consists of two or more columns, MySQL can perform optimization for any key part.

To search for column values that are NULL, you cannot use an expr = NULL test. The following statement returns no rows, because expr = NULL is never true for any expression:

mysql> SELECT \* FROM my\_table WHERE phone = NULL;

* You can add an index on a column that can have NULL values if you are using the MyISAM, InnoDB, or MEMORY storage engine.
* Aggregate (group) functions such as [COUNT()](https://dev.mysql.com/doc/refman/8.0/en/aggregate-functions.html#function_count), [MIN()](https://dev.mysql.com/doc/refman/8.0/en/aggregate-functions.html#function_min), and [SUM()](https://dev.mysql.com/doc/refman/8.0/en/aggregate-functions.html#function_sum) ignore NULL values.
* The exception to this is [COUNT(\*)](https://dev.mysql.com/doc/refman/8.0/en/aggregate-functions.html#function_count), which counts rows and not individual column values.

## Optimizing Subqueries, Derived Tables

For a subquery used with an IN, = ANY, or EXISTS predicate, the optimizer has these choices:

* Semijoin
* Materialization
* EXISTS strategy

### Materialization

<https://dev.mysql.com/doc/refman/8.0/en/subquery-materialization.html>

Materialization means that a subquery is evaluated only once. There are several types of subqueries that can be materialized.

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 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.

#### Expression subqueries that are not correlated

A subquery can be materialized if it is a noncorrelated expression subquery. A correlated subquery is one that references columns in the outer query, and so has to be evaluated for each row in the outer query.

SELECT \* FROM Staff WHERE id = (SELECT MAX(manager) FROM Org)

The original statement is transformed into the following two statements:

**constant** = SELECT MAX(manager) FROM Org

SELECT \* FROM Staff WHERE id = **constant**

**How to know whether materialization is used**

Use of [EXPLAIN](https://dev.mysql.com/doc/refman/8.0/en/explain.html) with a query provides some indication of whether the optimizer uses subquery materialization: Optimize may change from DEPENDENT SUBQUERY to SUBQUERY.

select\_type= SUBQUERY This indicates that, for a subquery that would be executed once per outer row, materialization enables the subquery to be executed just once.

### Semi-Join

It is a whole new set of algorithms for processing subqueries. It is based on transforming a subquery into a [semi-join operation](http://en.wikipedia.org/wiki/Semijoin), and then treating semi-join like another join operation throughout the optimizer. The optimizer uses semi join strategies to improve subquery execution

A subquery can be transformed to a semi-join if it matches these criteria:

* The subquery is part of an IN or =ANY predicate. It cannot be e.g. NOT IN.
* The subquery consists of a single query block (it must not contain UNION).
* The subquery does not contain GROUP BY or HAVING.
* The subquery is not implicitly grouped (it contains no aggregate functions).
* The subquery predicate is part of a WHERE clause.

#### Correlated subqueries

<https://flylib.com/books/en/1.142.1/tuning_subqueries.html>

We can identify the subquery through the DEPENDENT SUBQUERY tag in the Select type column of the EXPLAIN statement output.

The most obvious way to make a subquery run fast is to ensure that it is supported by an index. Ideally, we should create a concatenated index that includes every column referenced within the subquery.

We can see from the following EXPLAIN output that MySQL makes use of the index to resolve the subquery. The output also includes the **Using index** clause, indicating that only the index is used the most desirable execution plan for a subquery.

* Subqueries should be optimized by creating an index on all of the columns referenced in the subquery. SQL statements containing subqueries that are not supported by an index can show exponential degradation as table row counts increase.
* Instead of using a correlated query you could use a LEFT JOIN.

#### Differemce between semi-Join and Regular Join

* Semi join either returns each row from input A, or it does not. No row duplication can occur.
* Regular join duplicates rows if there are multiple matches on the join predicate.
* Semi join is defined to only return columns from input A.
* Regular join may return columns from either (or both) join inputs.

T-SQL currently lacks support for direct syntax like FROM A SEMI JOIN B ON A.x = B.y, so we need to use indirect forms like EXISTS, SOME/ANY (including the equivalent shorthand IN for equality comparisons), and set INTERSECT.

### Anti-Join –(NOT IN, NOT EXIST)

<https://mysqlserverteam.com/antijoin-in-mysql-8/>

This new query uses the [antijoin](https://en.wikipedia.org/wiki/Relational_algebra#Antijoin_(%E2%96%B7)) operator ; it is like the join operator except that instead of looking for matches it looks for non-matches ; precisely, it selects records from the left side for which the right side has no record matching the ON condition.

### Range Optimization

<https://use-the-index-luke.com/sql/where-clause/searching-for-ranges/greater-less-between-tuning-sql-access-filter-predicates>

The biggest performance risk of an INDEX RANGE SCAN is the [leaf node traversal](https://use-the-index-luke.com/sql/anatomy/the-leaf-nodes). It is therefore the golden rule of indexing to keep the scanned index range as small as possible. You can check that by asking yourself where an index scan starts and where it ends.

**Example**

SELECT first\_name, last\_name, date\_of\_birth FROM employees WHERE

date\_of\_birth >= TO\_DATE(?, 'YYYY-MM-DD') AND date\_of\_birth <= TO\_DATE(?, 'YYYY-MM-DD')

An index on DATE\_OF\_BIRTH is only scanned in the specified range. The scan starts at the first date and ends at the second. We cannot narrow the scanned index range any further.

SELECT first\_name, last\_name, date\_of\_birth FROM employees

WHERE date\_of\_birth >= TO\_DATE(?, 'YYYY-MM-DD') AND date\_of\_birth <= TO\_DATE(?, 'YYYY-MM-DD')

AND subsidiary\_id = ?

Of course an ideal index has to cover both columns, but the question is in which order?

**Rule of thumb**: index for equality first—then for ranges.

<http://dev.cs.ovgu.de/db/mysql/Query-Speed.html>

The range access method uses a single index to retrieve a subset of table records that are contained within one or several index value intervals. It can be used for a single-part or multiple-part index.

#### Range Access Method for Single-Part Indexes

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.
* For BTREE indexes, comparison of a key part with a constant value is a range condition when using the >, <, >=, <=, BETWEEN, !=, or <> operators, or LIKE '*pattern*' (where '*pattern*' doesn't start with a wildcard).
* For all types of indexes, 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 the same join
* The result of an uncorrelated subquery
* Any expression composed entirely from subexpressions of the preceding types

## Understanding Execution Plan

<https://dev.mysql.com/doc/refman/8.0/en/explain-output.html>

#### Select\_Type

SIMPLE – the query is a simple SELECT query without any subqueries or UNIONs

PRIMARY – the SELECT is in the outermost query in a JOIN

SUBQUERY – the first SELECT in a subquery

DEPENDENT SUBQUERY – a subquery which is dependent upon on outer query

UNION – the SELECT is the second or later statement of a UNION

DEPENDENT UNION – the second or later SELECT of a UNION is dependent on an outer query

#### possible\_keys

The possible\_keys column indicates the indexes from which MySQL can choose to find the rows in this table.

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](https://dev.mysql.com/doc/refman/8.0/en/explain.html) again

#### Keys

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.

* The **possible\_keys** and **key** values are both **NULL**, which indicates that MySQL does not have an index it can use for this query.

#### 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.

‘ref’ column tells us which columns or constants are compared to the index picked by the optimizer.

#### Extra

<https://stackoverflow.com/questions/25672552/whats-the-difference-between-using-index-and-using-where-using-index-in-the>

If you want to make your queries as fast as possible, look out for Extra column values of **Using filesort** and **Using temporary**

##### Using where

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](https://dev.mysql.com/doc/refman/8.0/en/explain-output.html#jointype_all) or [index](https://dev.mysql.com/doc/refman/8.0/en/explain-output.html#jointype_index).

using where does not mean you are using a **WHERE** clause. Using where just means there is some restricting clause on the table (**WHERE** or **ON**), and not all record will be returned.

##### 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.

Using index means that all information is returned from the index, without seeking the records in the table. This is only possible if all fields required by the query are covered by the index.

 Index can be used even when Using index is absent from the Extra column. This is the case if type is [index](https://dev.mysql.com/doc/refman/8.0/en/explain-output.html#jointype_index) and key is PRIMARY.

##### Using index condition

<https://mariadb.com/kb/en/index-condition-pushdown/>

Index Condition Pushdown is an optimization that is applied for access methods that access table data through indexes: range, ref, eq\_ref

The idea is to check part of the WHERE condition that refers to index fields (we call it Pushed Index Condition) as soon as we've accessed the index. If the Pushed Index Condition is not satisfied, we won't need to read the whole table record.Index Condition Pushdown optimization tries to cut down the number of full record reads by checking whether index records satisfy part of the WHERE condition that can be checked for them:

<https://dev.mysql.com/doc/refman/5.6/en/index-condition-pushdown-optimization.html>

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](https://dev.mysql.com/doc/refman/5.6/en/explain.html) 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.

##### Using filesort

<https://dev.mysql.com/doc/refman/8.0/en/order-by-optimization.html>

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.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.

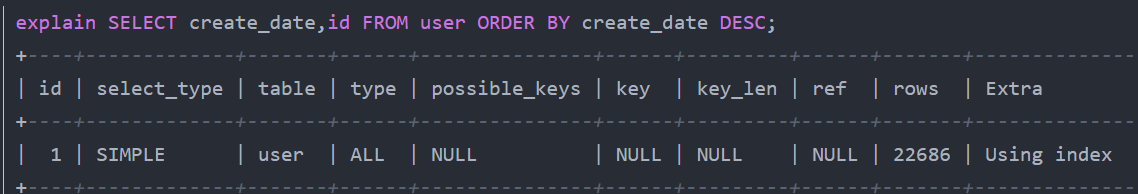
filesort can be used **Memory space.**The size is the value of the parameter sort\_buffer\_size, and the default is 2M. When there are too many records to sort sort\_buffer\_sizeWhen not enough, mysql will use **Temporary files to store each block**, And then each block is sorted and then merged multiple times to finally complete the sorting globally. Can be increased sort\_buffer\_sizeTo solve the problem of slow filesort, which is the second sort above.

**Example**

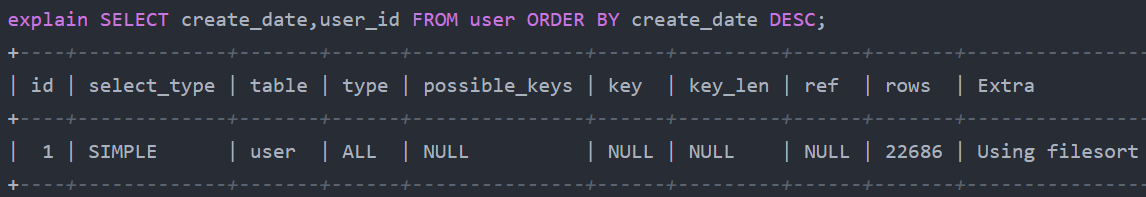
Only id has PRIMARY index, according to create\_date is sorted due to create\_date did not build an index.

1. explain SELECT create\_date,id FROM user ORDER BY create\_date DESC;
2. +*----+-------------+-------+------+---------------+------+---------+------+-------+-----*
3. | id | select\_type | table | type | possible\_keys | key  | key\_len| ref| rows |Extra
4. +*----+-------------+-------+------+---------------+------+---------+------+-------+------*
5. |  1 | SIMPLE      | user  | ALL  | NULL        | NULL| NULL| NULL|22686 | Using filesort
6. +*----+-------------+-------+------+---------------+------+---------+------+-------+-----*

When using a field without an index, Using filesort will appear. Then we build After create\_date index, take a look at the result.



If I want to query the user\_id as well, check the result because this is not part of index.



##### Using temporary

Temporary tables can be created under conditions such as these:

* If there is an ORDER BY clause and a different GROUP BY clause
* if the ORDER BY or GROUP BY contains columns from tables other than the first table in the join queue.
* Evaluation of derived tables (subqueries in the FROM clause).

For Using temporary; Using filesort, the index should be used appropriately. Using temporary is generally applicable to GROUP BY caluse.

##### Differennce between Using where/index and index condition

<https://developpaper.com/question/mysql-execution-plan-use-where-use-index-and-use-index-condition/#:~:text=The%20result%20is%20that%20idx_fk_customer_id,and%20retrieved%20from%20the%20index.&text=Extra%20is%20used%20as%20the%20index%20condition>.

Rental table, where **customer\_id** establishes an index named idx\_fk\_customer\_id.

Using Where;Using index

EXPLAIN SELECT customer\_id FROM rental WHERE customer\_id>=300;

MySQL execution plan (Use where, Use index and Use index condition)

'Using index' meaning not doing the scan of entire table. it means that first the index is used to retrieve the records (an actual access to the table is not needed) and then on top of this result set the filtering of the where clause is done.

Using where

EXPLAIN SELECT \* FROM rental WHERE customer\_id>=373;

MySQL execution plan (Use where, Use index and Use index condition)

because SELECT statements contain data that does not exist in the index, so you need to query the data through the index return table, so I can understand that Extra is using where, but here the type is actually ALL, that is to say, **full table scanning** is used in the execution plan.

EXPLAIN SELECT customer\_id FROM rental WHERE customer\_id>=373 AND customer\_id<400;

MySQL execution plan (Use where, Use index and Use index condition)

Using Index

Using index means that all information is returned from the index, without seeking the records in the table. This is only possible if all fields required by the query are covered by the index. Since you are selecting \*, this is impossible.

Using index condition

EXPLAIN SELECT \* FROM rental WHERE customer\_id>=373 AND customer\_id<400;

MySQL execution plan (Use where, Use index and Use index condition)

The execution plan of this statement is well understood. Firstly, the index is filtered using a condition in cusomter\_id>373 or customer\_id<400. After filtering the index, the index is scanned through the index return table and finds all data rows that meet the index criteria again and then filters these data rows with other criteria in the WHERE clause. Extra is used as the index condition. Index Condition Pushdown optimization is used for the [**range**](https://spec-zone.ru/mysql/5.6/optimization.html#jointype_range), [**ref**](https://spec-zone.ru/mysql/5.6/optimization.html#jointype_ref), [**eq\_ref**](https://spec-zone.ru/mysql/5.6/optimization.html#jointype_eq_ref), and [**ref\_or\_null**](https://spec-zone.ru/mysql/5.6/optimization.html#jointype_ref_or_null) access methods when there is a need to access full table rows.

**Which is better?**

'Using where; Using index' would be better then 'Using index condition' if query has index all covering.

**ORDER BY Execution Plan**

* If the Extra column of [EXPLAIN](https://dev.mysql.com/doc/refman/8.0/en/explain.html) output does not contain Using filesort, the index is used and a filesort is not performed.
* If the Extra column of [EXPLAIN](https://dev.mysql.com/doc/refman/8.0/en/explain.html) output contains Using filesort, the index is not used and a filesort is performed.

#### Access Type

The type column of [EXPLAIN](https://dev.mysql.com/doc/refman/8.0/en/explain.html) output describes how tables are joined.

From best to worst, here is the list

* system
* const
* eq\_ref
* ref
* range
* index
* all

1. [**const**](https://dev.mysql.com/doc/refman/8.0/en/explain-output.html#jointype_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. This is the **fastest type of join** because the table only has to be read once

[const](https://dev.mysql.com/doc/refman/8.0/en/explain-output.html#jointype_const) is used when you compare all parts of a PRIMARY KEY or UNIQUE index to constant values.

SELECT \* FROM *tbl\_name* WHERE *primary\_key*=1;

SELECT \* FROM *tbl\_name*  WHERE *primary\_key\_part1*=1 AND *primary\_key\_part2*=2;

1. [**eq\_ref**](https://dev.mysql.com/doc/refman/8.0/en/explain-output.html#jointype_eq_ref)

One row is read from this table for each combination of rows from the previous tables. Other than the [system](https://dev.mysql.com/doc/refman/8.0/en/explain-output.html#jointype_system) and [const](https://dev.mysql.com/doc/refman/8.0/en/explain-output.html#jointype_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](https://dev.mysql.com/doc/refman/8.0/en/explain-output.html#jointype_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. The where clause cannot match multiple rows because the primary key constraint ensures uniqueness of the EMPLOYEE\_ID values. The database does not need to follow the index leaf nodes—it is enough to traverse the index tree.

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;

* 1. [**ref**](https://dev.mysql.com/doc/refman/8.0/en/explain-output.html#jointype_ref)

it can be multiple matching rows and access will be done through an index. ref is used if the **key is not a PRIMARY KEY or UNIQUE index**.  If the key that is used matches only a **few rows**, this is a good join type. [ref](https://dev.mysql.com/doc/refman/8.0/en/explain-output.html#jointype_ref) can be used for indexed columns that are compared using  =  or  <=> operator.

* 1. [**fulltext**](https://dev.mysql.com/doc/refman/8.0/en/explain-output.html#jointype_fulltext)

The join is performed using a FULLTEXT index.

* 1. [**ref\_or\_null**](https://dev.mysql.com/doc/refman/8.0/en/explain-output.html#jointype_ref_or_null)

This join type is like [ref](https://dev.mysql.com/doc/refman/8.0/en/explain-output.html#jointype_ref), but with the addition that MySQL does an extra search for rows that contain NULL values.

SELECT \* FROM *ref\_table*

WHERE *key\_column*=*expr* OR *key\_column* IS NULL;

* 1. [**ALL**](https://dev.mysql.com/doc/refman/8.0/en/explain-output.html#jointype_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](https://dev.mysql.com/doc/refman/8.0/en/explain-output.html#jointype_const), and usually very bad in all other cases. Normally, you can avoid [ALL](https://dev.mysql.com/doc/refman/8.0/en/explain-output.html#jointype_all) by adding indexes that enable row retrieval from the table based on constant values or column values from earlier tables.  A full scan is only used when no other method is available.

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.

* 1. [**index**](https://dev.mysql.com/doc/refman/8.0/en/explain-output.html#jointype_index)

The index join type is the same as [ALL](https://dev.mysql.com/doc/refman/8.0/en/explain-output.html#jointype_all), the entire index tree is scanned to find matching rows. This usually is faster than ALL, because the index file usually is smaller than the data file. MySQL can use this join type when the query uses only columns that are part of a single index.

* 1. **Range**

An index is used to find matching rows in a specific range, typically when the key column is compared to a constant using operators like BETWEEN, IN, >, >=, IS NULL etc.

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);

### How to avoid Full Scan

<https://dev.mysql.com/doc/refman/8.0/en/table-scan-avoidance.html>

The output from [EXPLAIN](https://dev.mysql.com/doc/refman/8.0/en/explain.html) shows [ALL](https://dev.mysql.com/doc/refman/8.0/en/explain-output.html#jointype_all) in the type column when MySQL uses a [full table scan](https://dev.mysql.com/doc/refman/8.0/en/glossary.html#glos_full_table_scan) to resolve a query. This usually happens under the following conditions:

* The table is so small (even empty) 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.(no join or Filter condition)
* 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. I.e. filter condition returning most of record from table.
* 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 probably requires many key lookups and that a table scan would be faster. i.e. one row is matching with many rows

**Solution**

Use FORCE INDEX for the scanned table to tell MySQL that table scans are very expensive compared to using the given index:

<https://www.dbrnd.com/2016/10/mysql-select-with-index-hint-option-to-optimize-the-query-use-index-force-index-ignore-index/>

In most of the cases, Query Planner and Query Optimizer are accurate to choose perfect Index.But sometimes, still It is required to give Index Hint for reducing the overall Query Planning Time. If we know that this Index is good for this query, we should Provide the Hint to Query Optimizer. MySQL supports command like **USE INDEX**, **IGNORE INDEX**, **FORCE INDEX**, which we can use for Index Hint.

The USE INDEX hint tells MySQL to use only one of the named indexes to find rows in the table.The IGNORE INDEX tells MySQL to not use some particular index or indexes.  
The FORCE INDEX hint acts like USE INDEX , with the addition that a table scan is assumed to be very expensive.

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.

In case the query optimizer ignores the index, you can use the FORCE INDEX hint to instruct it to use the index instead. In this syntax, you put the FORCE INDEX clause after the FROM clause followed by a list of named indexes that the query optimizer must use.

**Syntax**

SELECT \* FROM table\_name FORCE INDEX (index\_list) WHERE condition;

**Example**

SELECT productName, buyPrice FROM products WHERE buyPrice BETWEEN 10 AND 80 ORDER BY buyPrice;

As you can guess, to return the products the query optimizer had to scan the whole table because no index is available for the buyPrice column:

Let’s create an index for the buyPrice column and execute the query again.

CREATE INDEX idx\_buyprice ON products(buyPrice);

Surprisingly, the query optimize did not use the index for the buyPrice column even though the index exists. The reason is that the query returns 94 rows out of 110 rows of the products table, therefore, the query optimizer decided to perform a full table scan.

To force the query optimizer to use the idx\_buyprice index, you use the following query:

SELECT productName, buyPrice FROM products **FORCE INDEX (idx\_buyPrice)**

WHERE buyPrice BETWEEN 10 AND 80 ORDER BY buyPrice;

This time, the index was used for finding the products

#### Types of Force index

|  |  |
| --- | --- |
| **New Hint(version** 8.0.20) | **Old Hint(Old Version)** |
| JOIN\_INDEX NO\_JOIN\_INDEX | FORCE INDEX FOR JOIN IGNORE INDEX FOR JOIN |
| GROUP\_INDEX NO\_GROUP\_INDEX | FORCE INDEX FOR GROUP BY IGNORE INDEX FOR GROUP BY |
| ORDER\_INDEX NO\_ORDER\_INDEX | FORCE INDEX FOR ORDER BY IGNORE INDEX FOR ORDER BY |
| INDEX NO\_INDEX | FORCE INDEX IGNORE INDEX |

SELECT ci.CountryCode, co.Name AS Country, ci.Name AS City, ci.District   FROM world.country co **IGNORE INDEX** (Primary)

 INNER JOIN world.city ci **FORCE INDEX** **FOR ORDER BY** (CountryCode)

 ON ci.CountryCode = co.Code  WHERE co.Continent = 'Asia' ORDER BY ci.CountryCode, ci.ID;

##### FORCE INDEX FOR ORDER BY

MySQL can use an index on the columns in the ORDER BY (under certain conditions). However, MySQL cannot use an index for mixed ASC,DESC order by (SELECT \* FROM foo ORDER BY bar ASC, pants DESC).

### Create Index wherever possible

EXPLAIN SELECT city.name, city.district FROM city, country WHERE city.countrycode = country.code AND country.code = 'IND';

+----+-------------+---------+-------+---------------+---------+---------+-------+------+-------------+  
| id | select\_type | table   | type  | possible\_keys | key     | key\_len | ref  | rows | Extra       |  
+----+-------------+---------+-------+---------------+---------+---------+-------+------+-------------+  
|  1 | SIMPLE      | country | const | PRIMARY       | PRIMARY | 3       | const |    1 | Using index |  
|  1 | SIMPLE      | city    | ALL   | NULL          | NULL    | NULL    | NULL | 4079 | Using where |  
+----+-------------+---------+-------+---------------+---------+---------+-------+------+-------------+

 It should be clear the current design will require MySQL to process only one record in the country table (which is indexed) but all 4079 records in the city table (which isn't). This then suggests scope for improvement using other optimization tricks - for example, adding an index to the city table

CREATE INDEX idx\_ccode ON city(countrycode);

After index creation, below is EPLAIN Output

+----+-------------+---------+-------+---------------+-----------+---------+-------+------+-------------+  
| id | select\_type | table   | type  | possible\_keys | key       | key\_len | ref   | rows | Extra       |  
+----+-------------+---------+-------+---------------+-----------+---------+-------+------+-------------+  
|  1 | SIMPLE      | country | const | PRIMARY       | PRIMARY   | 3       | const |    1 | Using index |  
|  1 | SIMPLE      | city    | ref   | idx\_ccode     | idx\_ccode | 3       | const |  333 | Using where |  
+----+-------------+---------+-------+---------------+-----------+---------+-------+------+-------------+

MySQL now only needs to scan 333 records in the city table to produce a result set -- a reduction of almost 90 percent! Naturally, this translates into faster query execution time and more efficient usage of database resources.

### Optimize Like Statements With Union Clause

Example, consider a case where you are running the below query with the '**first\_name**' and '**last\_name**' indexed:

select \* from students where first\_name like 'Ade%' **or** last\_name like 'Ade%' ;

The query above can run far much slower compared to the below query which uses a union operator merge the results of 2 separate fast queries that takes advantage of the indexes.

select from students where first\_name like 'Ade%'

union all

select from students where last\_name like

Example2

SELECT DISTINCT PRODUCT.ProductID,PRODUCT.Name

FROM Production.Product PRODUCT

INNER JOIN Sales.SalesOrderDetail DETAIL

ON PRODUCT.ProductID = DETAIL.ProductID

**OR** PRODUCT.rowguid = DETAIL.rowguid;

SQL Server cannot easily process an OR condition across multiple columns. The best way to deal with an OR is to eliminate it (if possible) or break it into smaller queries. Breaking a short and simple query into a longer, more drawn-out query may not seem elegant, but when dealing with OR problems, it is often the best choice:

Best way to write above query as below

SELECT PRODUCT.ProductID,PRODUCT.Name FROM Production.Product PRODUCT

INNER JOIN Sales.SalesOrderDetail DETAIL ON PRODUCT.ProductID = DETAIL.ProductID

UNION

SELECT PRODUCT.ProductID,PRODUCT.Name FROM Production.Product PRODUCT

INNER JOIN Sales.SalesOrderDetail DETAIL ON PRODUCT.rowguid = DETAIL.rowguid

### Avoid Like Expressions With Leading Wildcards

MySQL is not able to utilize indexes when there is a leading wildcard in a query. If we take our example above on the students table, a search like this will cause MySQL to perform full table scan even if you have indexed the '**first\_name**' field on the students table.

select \* from students where first\_name like '%Ade'

+----+-------------+----------+------------+------+---------------+------+---------+------+------+----------+-------------+

| id | select\_type | table | partitions | type | possible\_keys | key | key\_len | ref | rows | filtered

+----+-------------+----------+------------+------+---------------+------+---------+------+------+----------+-------------+

| 1 | SIMPLE | students | NULL | ALL | NULL | NULL | NULL | NULL | 500 | 11.11

+----+-------------+----------+------------+------+---------------+------+---------+------+------+----------+

### Take Advantage of MySQL Full-Text Searches

If you are faced with a situation where you need to search data using wildcards and you don't want your database to underperform, you should consider using MySQL full-text search (FTS) because it is far much faster than queries using wildcard characters.

To add a full-text search index to the students sample table, we can use the below MySQL command:

Alter table students ADD FULLTEXT (first\_name, last\_name);

If we query the optimizer about the execution plan of the above query, we will get the following results:

explain Select \* from students where match(first\_name, last\_name) AGAINST ('Ade');

+----+-------------+----------+------------+----------+---------------+------------+---------+-------+------+----------+------

| id | select\_type | table| partitions | type | possible\_keys | key | key\_len | ref| rows | filtered

+----+-------------+----------+------------+----------+---------------+------------+---------+-------+------+----------+------

| 1 | SIMPLE| students | NULL | fulltext | first\_name | first\_name | 0 | const | 1 | 100.00

+----+-------------+----------+------------+----------+---------------+------------+---------+-------+------

### Avoid Too Many Columns in table rather split it into two

Wide tables can be extremely expensive and require more CPU time to process. If possible, don't go above a hundred unless your business logic specifically calls for this.

Instead of creating one wide table, consider splitting it apart in to logical structures. For instance, if you are creating a customer table but you realize a customer can have multiple addresses, it is better to create a separate table for holding customers addresses that refer back to the customers table using the '**customer\_id**' field.

### Fetching more rows than needed

You should always be suspicious when you see SELECT \*. Do you really need all columns? Probably not. Retrieving all columns can prevent optimizations such as covering indexes, as well as adding I/O, memory, and CPU overhead for the server.

SELECT \* FROM sakila.actor

INNER JOIN sakila.film\_actor USING(actor\_id)

INNER JOIN sakila.film USING(film\_id)

WHERE sakila.film.title = 'Academy Dinosaur';

That returns all columns from all three tables. Instead, write the query as follows:

SELECT sakila.actor.\* FROM sakila.actor.;

### Chopping Up a Query

Purging old data is a great example. Periodic purge jobs may need to remove quite a bit of data, and doing this in one massive query could lock a lot of rows for a long time, fill up transaction logs, hog resources, and block small queries that shouldn’t be interrupted. Chopping up the DELETE statement and using medium-size queries can improve performance considerably, and reduce replication lag when a query is replicated. For example, instead of running this monolithic query:

### JOIN OPTIMIZER

It is often possible to join the tables in several different orders and get the same results. The join optimizer estimates the cost for various plans and tries to choose the least expensive one that gives the same result.

<https://severalnines.com/database-blog/using-explain-improve-sql-query-performance>

SELECT title, description

FROM film AS f

JOIN film\_actor AS fa ON f.film\_id = fa.film\_id

JOIN actor AS a ON fa.actor\_id = a.actor\_id

WHERE a.last\_name = 'MIRANDA' AND f.release\_year = 2006

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 1. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

           id: 1

  select\_type: SIMPLE

        table: a

         type: ref

possible\_keys: PRIMARY,idx\_actor\_last\_name

          key: idx\_actor\_last\_name

      key\_len: 137

          ref: const

         rows: 1

        Extra: Using where; Using index

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 2. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

           id: 1

  select\_type: SIMPLE

        table: fa

         type: ref

possible\_keys: PRIMARY,idx\_fk\_film\_id

          key: PRIMARY

      key\_len: 2

          ref: sakila.a.actor\_id

         rows: 13

        Extra: Using index

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 3. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

           id: 1

  select\_type: SIMPLE

        table: f

         type: eq\_ref

possible\_keys: PRIMARY

          key: PRIMARY

      key\_len: 2

          ref: sakila.fa.film\_id

         rows: 1

        Extra: Using where

By looking at the output we can see that the optimizer decided to start with the ‘actor’ table, using ‘idx\_actor\_last\_name’ index to search for the correct last name. As a next step, ‘film\_actor’ table was joined. PK was used to identify rows and as a reference sakila.a.actor\_id column was used. Finally, ‘film’ table was joined, again using primary key. As you can see, join type was ‘eq\_ref’ which means that for every row in the ‘film\_actor’ table, only a single row in the ‘film’ table was queried.

Please keep in mind that, in joins, **we need to multiply rows accessed on every step to get the total number of combinations** - here we have 1 x 13 x 1 = 13 rows. This is pretty important as, when joining multiple tables, you can easily end up scanning billions of rows if your joins are not indexed properly.

### Join Decomposition

Many high-performance web sites use join decomposition. You can decompose a join by running multiple single-table queries instead of a multitable join, and then performing the join in the application. For example, instead of this single query:

mysql> **SELECT \* FROM tag**

->  **JOIN tag\_post ON tag\_post.tag\_id=tag.id**

->  **JOIN post ON tag\_post.post\_id=post.id**

-> **WHERE tag.tag='mysql';**

You might run these queries:

mysql> **SELECT \* FROM tag WHERE tag='mysql';**

mysql> **SELECT \* FROM tag\_post WHERE tag\_id=1234;**

mysql> **SELECT \* FROM post WHERE post.id in (123,456,567,9098,8904);**

This looks wasteful at first glance, because you’ve increased the number of queries without getting anything in return. However, such restructuring can actually give significant performance advantages.

* You replace joins with IN() lists on large tables

### Join Order

<https://www.sqlshack.com/query-optimization-techniques-in-sql-server-tips-and-tricks/>

Based on how tables are joined, a query will fall into one of two basic forms:

* **Left-Deep Tree**: A join B, B join C, C join D, D join E, etc…This is a query in which most tables are sequentially joined one after another. (Record Count- **n!**  where n=no of table) **Best join**
* **Bushy Tree**: A join B, A join C, B join D, C join E, etc…This is a query in which tables branch out into multiple logical units within each branch of the tree.-- **(2n-2)! / (n-1)!**
* Move metadata or lookup tables into a separate query that places this data into a temporary table.
* Joins that are used to return a single constant can be moved to a parameter or variable.
* Break a large query into smaller queries whose data sets can later be joined together when ready.

SELECT \* FROM Production.Product

INNER JOIN Production.ProductSubCategory ON ProductSubCategory.ProductSubcategoryID = Product.ProductSubcategoryID

INNER JOIN Production.ProductCategory ON ProductCategory.ProductCategoryID = ProductSubCategory.ProductCategoryID

First Query--------

SELECT \* INTO #Product FROM Production.Product

INNER JOIN Production.ProductSubCategory ON ProductSubCategory.ProductSubcategoryID = Product.ProductSubcategoryID

Dependent Query----

SELECT \* FROM #Product Product

INNER JOIN Production.ProductModel ON ProductModel.ProductModelID = Product.ProductModelID

* For very heavily used queries, consider an indexed view to streamline constant access to important data.
* Remove unneeded tables, subqueries, and joins.

### Condition Filtering

<https://dev.mysql.com/doc/refman/5.7/en/condition-filtering.html>

SELECT \* FROM employee JOIN department ON employee.dept\_no = department.dept\_no

AND employee.first\_name = 'John'

**OR**

SELECT \* FROM employee JOIN department ON employee.dept\_no = department.dept\_no

WHERE employee.first\_name = 'John' --Condition filtering

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](https://dev.mysql.com/doc/refman/5.7/en/explain.html) 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. 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 × 20% = 1000 × .2 = 200.

### covered index--Complex Slow Queries(GROUP BY,ORDER BY)

<https://jaxenter.com/level-up-your-mysql-query-tuning-107279.html#:~:text=MySQL%20will%20need%20to%20create,and%20avoid%20creating%20temporary%20table>.

Those queries are usually the slowest ones. We will show how to optimize those queries and decrease the query response time as well as the application performance in general.

The table is 6 million rows and approximately 2GB in size.

* From this data we want to:
* Find maximum delay for flights on Sunday
* Group by airline

Our example query is:

select max(DepDelayMinutes),carrier, dayofweek

from ontime\_2012 where dayofweek = 7

group by Carrier

**Explain Output**

type: ALL

possible\_keys: NULL

key: NULL

key\_len: NULL

ref: NULL

rows: 4833086

Extra: Using where; Using temporary; Using filesort

As we can see, MySQL does not use any index and have to scan ~4M rows. In addition, it will have to create a large temporary table. If we will create an index on “**dayofweek**” it will only filter out some rows and MySQL will still need to create a temporary table:

**Solution**

However, we can create a covered index on (dayofweek, Carrier, DepDelayMinutes) in this particular order. In this case MySQL will be able to use this index and avoid creating a temporary table:

alter table ontime\_2012

add key covered(dayofweek, Carrier, DepDelayMinutes);

EXPLAIN OUTPUT

possible\_keys: DayOfWeek,covered

         key: covered

     key\_len: 2

         ref: const

        rows: 905138

       Extra: Using where; Using index

As we can see from the explain, MySQL will use our index and will avoid creating a temporary table. This is the fastest possible solution.

Note that MySQL will also be able to use non-covered index on (dayofweek, Carrier) and avoid creating a temporary table. However, covered index will be faster as MySQL will be able to satisfy the whole query by just reading the index.

**Challenges**

MySQL will not be able to use an index and avoid a filesort if we have a “range” scan in the where clause. Here’s an example

explain select max(DepDelayMinutes), Carrier, dayofweek from ontime\_2012

where dayofweek > 5 group by Carrier, dayofweekG

type: range

possible\_keys: covered

         key: covered

     key\_len: 2

         ref: NULL

        rows: 2441781

       Extra: Using where; Using index; Using temporary; Using filesort

MySQL will still have to create a temporary table. To fix that we can use a simple trick and rewrite the query into 2 parts with UNION

(select max(DepDelayMinutes), Carrier, dayofweek

from ontime\_2012

where dayofweek = 6

group by Carrier, dayofweek)

union

(select max(DepDelayMinutes), Carrier, dayofweek

from ontime\_2012

where dayofweek = 7

group by Carrier, dayofweek)

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 1. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

       table: ontime\_2012

         key: covered

...

       Extra: Using where; Using index

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 2. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

       table: ontime\_2012

         key: covered

...

       Extra: Using where; Using index

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 3. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

          id: NULL

 select\_type: UNION RESULT

       table: <union1,2>

        type: ALL

possible\_keys: NULL

         key: NULL

     key\_len: NULL

         ref: NULL

        rows: NULL

       Extra: Using temporary

As we can see, MySQL uses our covered index for each of the 2 queries. It will still have to create a temporary table to merge the results, however, it will probably be much smaller temporary table as it will only need to store the resultsets of 2 queries.

**Example 2**

Also, in some cases, it makes sense to also add the columns from the SELECT clause to the index, to create covering index. This is only relevant if the index isn't already 'too large'. What's too large? Well, no official rule of thumb here, but I usually go with up to 5 columns in an index. Creating a covering index allows the database to not only filter using the index, but to also fetch the information required by the SELECT clause directly from the index, which saves precious I/O operations for fetching the data of the filtered rows from the table's data.

SELECT id,first\_name,last\_name,age FROM employees

WHERE first\_name = 'John' AND last\_name = 'Brack' AND age > 25

ORDER BY age ASC;

For this query, we'll start with adding the columns first\_name and last\_name, which are compared with an equality operator. Then, we'll add the age column which is compared with a range condition. No need to have the ORDER BY clause indexed here, as the age column is already in the index. Last but not least, we'll add id from the SELECT clause to the index to have a covering index.

### loose index

### Multi-column indexes

The engine will create multi-column index key values by concatenating the column values in the order they are specified for the index. MySQL allows you to create a composite index that consists of up to 16 columns. If you specify the columns in the right order in the index definition, a single composite index can speed up these kinds of queries on the same table.

Therefore, only queries that conform to the following criteria may use the index:

* queries that test all columns used in the index
* queries that test just the first column
* queries that test the first two columns
* queries that test the first three columns
* and so on...

**Example**

index(station, windangle)

SELECT \* FROM indexed\_samples WHERE station\_name = “test 1” and windangle = 180

SELECT \* FROM indexed\_samples WHERE station\_name = “test 1”

But, below two queries will not use the index

SELECT \* FROM indexed\_samples WHERE windangle = 180

SELECT \* FROM indexed\_samples WHERE station\_name = "test 1" OR windangle = 180;

If we rewrite the query like so, it can use the index:

SELECT \* FROM indexed\_samples WHERE station\_name = "test 1" OR

(station\_name = "test 2" AND station\_name = "test 3" AND windangle = 180);

**Example2**

CREATE INDEX index\_name ON table\_name(c2,c3,c4);

Notice that if you have a composite index on (c1,c2,c3), you will have indexed search capabilities on one the following column combinations:

(c1)

(c1,c2)

(c1,c2,c3)

Optimizer can not use index for below Query

SELECT \* FROM table\_name WHERE c1 = v1 AND c3 = v3;

#### order of index columns

SELECT first\_name,last\_name FROM contacts

WHERE first\_name = 'John';

Having the index contacts (first\_name, last\_name) is ideal here, because the index starts with our filtering condition and ends with another column in the SELECT clause.

But, having the reverse index contacts (last\_name, first\_name) is rather useless, as the database can't use the index for filtering, as the column we need is second in the index and not first.

<https://blog.nodeswat.com/making-slow-queries-fast-with-composite-indexes-in-mysql-eb452a8d6e46>

explore above link

### Index Merge

<https://dev.mysql.com/doc/refman/8.0/en/index-merge-optimization.html>

The Index Merge access method retrieves rows with multiple [range](https://dev.mysql.com/doc/refman/8.0/en/explain-output.html#jointype_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.

In [EXPLAIN](https://dev.mysql.com/doc/refman/8.0/en/explain.html) output, the Index Merge method appears as [index\_merge](https://dev.mysql.com/doc/refman/8.0/en/explain-output.html#jointype_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. Extra field of [EXPLAIN](https://dev.mysql.com/doc/refman/8.0/en/explain.html) output as below

* Using intersect(...)
* Using union(...)
* Using sort\_union(...)

It is one of the most common question about indexing: is it better to create one index for each column or a single index for all columns of a where clause? The answer is very simple in most cases: one index with multiple columns is better—that is, a concatenated or compound index.

The other option is to use two separate indexes, one for each column. Then the database must scan both indexes first and then combine the results. Additionally, the database needs a lot of memory and CPU time to combine the intermediate results.

Databases use two methods to combine indexes. Firstly there is the index join. The second approach makes use of functionality from the data warehouse world i.e Bitmap index.

The optimizer chooses between different possible Index Merge algorithms

1. Index Merge Intersection Access Algorithm
2. Index Merge Union Access Algorithm
3. Index Merge Sort-Union Access Algorithm
4. 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](https://dev.mysql.com/doc/refman/8.0/en/logical-operators.html#operator_and)

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](https://dev.mysql.com/doc/refman/8.0/en/explain.html) 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;

**Example**

PRIMARY KEY(user\_id),

INDEX `parent\_id` (parent\_id),

INDEX `status` (status),

INDEX `user\_type` (user\_type)

Mysql> EXPLAIN SELECT user\_id FROM user WHERE user\_type=2 AND status=1 AND parent\_id=0 AND user\_id > 2938575 ORDER BY user\_id

Explain Output…………………………………………………………………………………………………………………………………

id: 1

select\_type: SIMPLE

table: users

type: index\_merge

possible\_keys: PRIMARY,parent\_id,status,user\_type

key: user\_type,status,parent\_id

key\_len: 2,2,4

ref: NULL

rows: 8100

**Extra**: Using intersect(user\_type,status,parent\_id); Using where; Using index; Using filesort

End\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

#### 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](https://dev.mysql.com/doc/refman/8.0/en/logical-operators.html#operator_or).

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](https://dev.mysql.com/doc/refman/8.0/en/logical-operators.html#operator_or), but the Index Merge union algorithm is not applicable.

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.

### ACCESS TYPES in EXPLAIN Keyword

When you’re thinking about the cost of a query, consider the cost of finding a single row in a table. MySQL can use several access methods to find and return a row.

The access method(s) appear in the type column in EXPLAIN’s output. The access types range from a

* **full table scan (ALL)**
* **index scans**,
* **range scans**,
* **unique index lookups**,
* **constants** (ref)

Each of these is faster than the one before it, because it requires reading less data. You don’t need to memorize the access types, but you should understand the general concepts of scanning a table, scanning an index, range accesses, and single-value accesses.

**SELECT \* FROM sakila.film\_actor WHERE film\_id = 1;**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 1. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

id: 1

select\_type: SIMPLE

table: film\_actor

type: ref

possible\_keys: idx\_fk\_film\_id

key: idx\_fk\_film\_id

key\_len: 2

ref: const

rows: 10

Extra:

EXPLAIN shows that MySQL estimated it needed to access only 10 rows. In other words, the query optimizer knew the chosen access type could satisfy the query efficiently. What would happen if there were no suitable index for the query? MySQL would have to use a less optimal access type, as we can see if we drop the index and run the query again:

mysql> **ALTER TABLE sakila.film\_actor DROP FOREIGN KEY fk\_film\_actor\_film;**

mysql> **ALTER TABLE sakila.film\_actor DROP KEY idx\_fk\_film\_id;**

mysql> **EXPLAIN SELECT \* FROM sakila.film\_actor WHERE film\_id = 1\G**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 1. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

id: 1

select\_type: SIMPLE

table: film\_actor

type: ALL

possible\_keys: NULL

key: NULL

key\_len: NULL

ref: NULL

rows: 5073

Extra: Using where

Predictably, the access type has changed to a full table scan (ALL), and MySQL now estimates it’ll have to examine 5,073 rows to satisfy the query. The “Using where” in the Extra column shows that the MySQL server is using the WHERE clause to discard rows after the storage engine reads them.

### Partitions

<https://severalnines.com/database-blog/using-explain-improve-sql-query-performance>

Partitions are great tool to manage large amounts of data. If we partition our table using temporal columns, we can easily rotate old data away and keep the recent data close together for better performance and lower fragmentation. Partitions can also be used to speed up queries - this process is called partition pruning. As long as we have in a WHERE clause the column which is used to partition a table, it may be possible to use that condition to access only relevant partitions.

CREATE TABLE `titles` (

  `emp\_no` int(11) NOT NULL,

  `title` varchar(50) NOT NULL,

  `from\_date` date NOT NULL,

  `to\_date` date DEFAULT NULL,

  PRIMARY KEY (`emp\_no`,`title`,`from\_date`)

) ENGINE=InnoDB DEFAULT CHARSET=latin1

/\*!50500 PARTITION BY RANGE  COLUMNS(from\_date)

(

PARTITION p10 VALUES LESS THAN ('1994-12-31') ENGINE = InnoDB,

PARTITION p11 VALUES LESS THAN ('1995-12-31') ENGINE = InnoDB,

PARTITION p12 VALUES LESS THAN ('1996-12-31') ENGINE = InnoDB,

 PARTITION p13 VALUES LESS THAN ('1997-12-31') ENGINE = InnoDB)

**EXPLAIN** **PARTITIONS** SELECT e.first\_name, e.last\_name FROM employees AS e JOIN titles AS t ON e.emp\_no = t.emp\_no WHERE from\_date > '1994-10-31' AND from\_date < '1996-03-31' AND t.title='Technique Leader'\G

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 1. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

           id: 1

  select\_type: SIMPLE

        table: t

   partitions: p10,p11,p12

         type: index

possible\_keys: PRIMARY

          key: PRIMARY

      key\_len: 59

          ref: NULL

         rows: 99697

        Extra: Using where; Using index

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 2. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

           id: 1

  select\_type: SIMPLE

        table: e

   partitions: NULL

         type: eq\_ref

possible\_keys: PRIMARY

          key: PRIMARY

      key\_len: 4

          ref: employees.t.emp\_no

         rows: 1

        Extra: NULL

As you can see, one new column (‘partitions’) was added. It contains values ‘p10,p11,p12’. This means that optimizer, based on the WHERE condition, decided that relevant data is located only in those partitions. Others won’t be accessed. Given the ‘type: index’, the table will be scanned using PRIMARY key to locate matching rows. As a next step, WHERE condition will be applied to locate rows with title='Technique Leader'. Finally, ‘employees’ table will be joined using it’s PK.

# MySql Keyword

CONCAT\_WS()  
This function in MySQL helps in joining two or more strings along with a separator. The separator must be specified by the user and it can also be a string. If the separator is NULL, then the result will also be NULL.

CONCAT\_WS(separator, string1, string2, ...)

Ex

SELECT CONCAT\_WS("@ ", "Geek ", "Vansh ", 13 ) AS ConcatWsStr;

Geek @ Vansh @ 13

## ISNULL()

The ISNULL() function returns 1 or 0 depending on whether an expression is NULL.

If expression is NULL, this function returns 1. Otherwise, it returns 0.

[SELECT](https://sercoaams-test.appiancloud.com/database/url.php?url=https://dev.mysql.com/doc/refman/5.5/en/select.html) ISNULL(350),ISNULL(""),ISNULL(NULL)

o/p--- 0 0 1

## IF(condition, true\_value, false\_value)

The MySQL IF() function is used for validating a condition. The IF() function returns a value if the condition is TRUE and another value if the condition is FALSE.

Example

SELECT IF(5<12, 'TRUE', 'FALSE');

**Output:** TRUE

Example2

IF(compliment = 'set' OR compliment = 'Y' OR compliment = 1, 'Y', 'N') AS customer\_compliment

Equivalent

IF(compliment IN('set','Y',1), 'Y', 'N') AS customer\_compliment

## CASE

CASE  
    WHEN condition1 THEN result1  
    WHEN condition2 THEN result2  
    WHEN conditionN THEN resultN  
    ELSE result  
END;

Example

SELECT CustomerName, City, Country FROM Customers  
ORDER BY  
(CASE     WHEN City IS NULL THEN Country  
    ELSE City  
END);

# [Exporting from MySQL](https://chartio.com/resources/tutorials/importing-from-and-exporting-to-files-using-the-mysql-command-line/" \l "exporting-from-mysql)

$ mysqldump -u my\_username -p database\_name > output\_file\_path

The options in use are:

* The -u flag indicates that the MySQL username will follow.
* The -p flag indicates we should be prompted for the password associated with the above username.
* database\_name is of course the exact name of the database to export.
* The > symbol is a Unix directive for [STDOUT](https://www.unix.com/man-page/linux/3/stdout/), which allows Unix commands to output the text results of the issued command to another location. In this case, that output location is a file path, specified by output\_file\_path.

# Reset root password

We can change the MySQL root password using the below statement in the new notepad file and save it with an appropriate name:

**ALTER** USER 'root'@'localhost' IDENTIFIED **BY** 'NewPassword';

Next, open a Command Prompt and navigate to the MySQL directory. Now, copy the following folder and paste it in our DOS command and press the Enter key.

C:\Users\javatpoint> CD C:\Program Files\MySQL\MySQL Server 8.0\bin

Next, enter this statement to change the password:

mysqld --init-file=C:\\mysql-notepadfile.txt

Finally, we can log into the MySQL server as root using this new password. After launches the MySQL server, it is to delete the C:\myswl-init.txt file to ensure the password change.

# Trigger

<https://www.mysqltutorial.org/create-the-first-trigger-in-mysql.aspx>

A trigger is a named MySQL object that **activates when an event occurs in a table**. Triggers are a particular type of stored procedure associated with a specific table.

**Syntax**

CREATE TRIGGER trigger\_name

{BEFORE | AFTER} {INSERT | UPDATE| DELETE }

ON table\_name FOR EACH ROW

……

………

trigger\_body;

* the trigger name must be unique within a database.
* specify the trigger action time which can be either BEFORE or AFTER which indicates that the trigger is invoked before or after each row is modified.
* specify the operation that activates the trigger, which can be [INSERT](https://www.mysqltutorial.org/mysql-insert-statement.aspx), [UPDATE](https://www.mysqltutorial.org/mysql-update-data.aspx), or [DELETE](https://www.mysqltutorial.org/mysql-delete-statement.aspx).
* specify the name of the table to which the trigger belongs after the ON keyword
* specify the statement to execute when the trigger activates. If you want to execute multiple statements, you use the BEGIN END compound statement.
* To distinguish between the value of the columns BEFORE and AFTER the DML has fired, you use the **NEW** and **OLD** modifiers.

|  |  |  |
| --- | --- | --- |
| **Trigger Event** | **OLD** | **NEW** |
| INSERT | No | Yes |
| UPDATE | Yes | Yes |
| DELETE | Yes | No |

**Example Update Trigger**:

CREATE TABLE EMPLOYEE

(eid int,

ename varchar(50)

)

CREATE TABLE employees\_audit (

id INT AUTO\_INCREMENT PRIMARY KEY,

employeeNumber INT NOT NULL,

lastname VARCHAR(50) NOT NULL,

changedat DATETIME DEFAULT NULL,

action VARCHAR(50) DEFAULT NULL

);

CREATE TRIGGER before\_employee\_update

BEFORE UPDATE ON EMPLOYEE

FOR EACH ROW

INSERT INTO employees\_audit(employeeNumber,lastname,changedat,action) values(OLD.eid,OLD.ename,NOW(),'update');

INSERT INTO EMPLOYEE VALUES(1,'khalid Anwar')

INSERT INTO EMPLOYEE VALUES(2,'khalid Anwar');

INSERT INTO EMPLOYEE VALUES(3,'khalid Anwar');

update EMPLOYEE set ename='Shahela3' where eid=3

SELECT \* FROM employees\_audit

**Example DELETE Trigger**:

CREATE TABLE employees\_DELETE (

id INT AUTO\_INCREMENT PRIMARY KEY,

employeeNumber INT NOT NULL,

lastname VARCHAR(50) NOT NULL,

changedat DATETIME DEFAULT NULL,

action VARCHAR(50) DEFAULT NULL

);

CREATE TRIGGER before\_employee\_DELETE

BEFORE **DELETE** ON EMPLOYEE

FOR EACH ROW

INSERT INTO employees\_audit(employeeNumber,lastname,changedat,action) values(OLD.eid,OLD.ename,NOW(),'insert');

**Example INSERT Trigger**:

CREATE TABLE employees\_INSERT (

id INT AUTO\_INCREMENT PRIMARY KEY,

employeeNumber INT NOT NULL,

lastname VARCHAR(50) NOT NULL,

changedat DATETIME DEFAULT NULL,

action VARCHAR(50) DEFAULT NULL

);

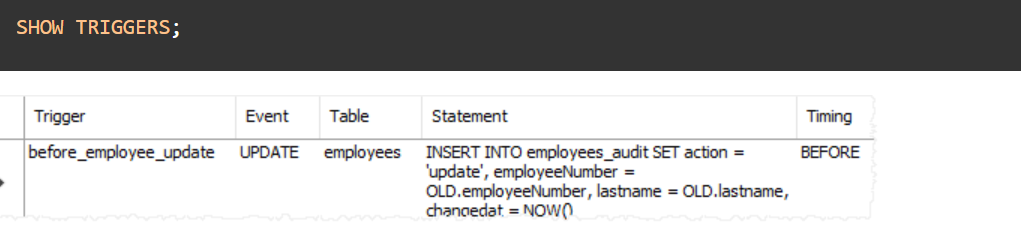
CREATE TRIGGER before\_employee\_DELETE

BEFORE **DELETE** ON EMPLOYEE

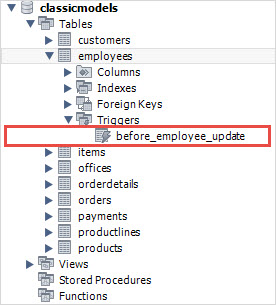
FOR EACH ROW

INSERT INTO employees\_audit(employeeNumber,lastname,changedat,action) values(new.eid,new.ename,NOW(),'delete');

## View Triggers



In addition, if you look at the schema using MySQL Workbench under the **employees > triggers**, you will see the before\_employee\_update trigger as shown in the screenshot below:



## Drop trigger

DROP TRIGGER <trigger name>;

## Create Multiple Triggers

MySQL does not support having multiple triggers fire at the same time. However, adding multiple logical operations to the same trigger is possible. Use the **BEGIN** and **END** delimiters to indicate the trigger body:

--DROP TRIGGER before\_employee\_update

DELIMITER //

CREATE TRIGGER before\_employee\_update

BEFORE UPDATE ON EMPLOYEE

FOR EACH ROW

BEGIN

INSERT INTO employees\_audit(employeeNumber,lastname,changedat,action) values(OLD.eid,OLD.ename,NOW(),'Old Value');

INSERT INTO employees\_audit(employeeNumber,lastname,changedat,action) values(new.eid,new.ename,NOW(),'new Value');

END //

# **MySQL Delimiter**

A MySQL client program such as MySQL Workbench or mysql program uses the delimiter (;) to separate statements and executes each statement separately.

However, a stored procedure consists of multiple statements separated by a semicolon (;).If you use a MySQL client program to define a stored procedure that contains semicolon characters, the MySQL client program will not treat the whole stored procedure as a single statement, but many statements. Therefore, you must redefine the delimiter temporarily so that you can pass the whole stored procedure to the server as a single statement.

To redefine the default delimiter, you use the DELIMITER command:

DELIMITER delimiter\_character e.g., // or $$

Example

DELIMITER //

CREATE PROCEDURE sp\_name()

BEGIN

statements1;

statements2;

END //

# Stored Procedures

A stored procedure is a prepared SQL code that you can save, so the code can be reused over and over again. So if you have an SQL query that you write over and over again, save it as a stored procedure, and then just call it to execute it.

**Create procedure**

DELIMITER $$

CREATE PROCEDURE procedure\_name  
BEGIN  
sql\_statement1;

sql\_statement2;  
END$$

DELIMITER ;

**Call procedure**

call proc\_name

# Transation

By default, MySQL automatically commits the changes permanently to the database. To force MySQL not to commit changes automatically, you use the following statement:

SET autocommit = 0; -- not to commit

SET autocommit = 1; -- auto commit

-->

SET autocommit =0

START TRANSACTION;

DELETE FROM DEPT WHERE deptid=6;

commit;

//OR

ROLLBACK;

## Savepoint

START transaction;

INSERT INTO DEPT VALUES(5,'D5'); --this will be inserted

SAVEPOINT s5;

INSERT INTO DEPT VALUES(6,'D6'); -- this will be ignored

ROLLBACK TO SAVEPOINT s5;

INSERT INTO DEPT VALUES(7,'D7'); --this will be inserted

COMMIT;

# How to export Log File

