Indexes in MySQL

A database index is a data structure that improves the speed of search operations in a table. The largest data is in table, the slowest it is finding the specific data.

1. Why do we use indexes?

Indexes are a type of table that keep a primary key or index field and a pointer to each record into the actual table. The users cannot see the indexes, they are just used to speed up queries and will be used by the Database Search Engine to locate records very fast. MySQL uses indexes for multiple purposes:

- To run queries with `WHERE` clause faster.
- Fast retrieving of rows from other tables when joining them.
- Query optimization: GROUP BY, ORDER BY optimizations.
- Fast retrieving of rows while selecting them.
- Constraints
- MAX(), MIN()

2. Index Types

MySQL has 2 storing engines: InnoDB and MyISAM.

2.1. Creation of Indexes.

There are 3 different ways to create indexes in MySQL.

1. Using CREAT INDEX keyword:

```
CREATE INDEX index_name
ON table_name(index_column_1,index_column_2,...);
```

2. While creating table with `CREATE TABLE` keyword:

```
CREATE TABLE table_name(
  column_1 CHAR(30) NOT NULL,
  INDEX (column_1));
```

3. Using ALTER TABLE keyword:

```
ALTER TABLE table_name ADD INDEX (column_1,
column_2);
```

2.2. Types of Indexes.

There are 6 types of indexes in MySQL. Each of them serve for different purposes.

2.2.1 Unique

A **unique index** is a type of index in which all column values have to be unique. In a single column unique index there can be no duplication of values in the column being indexed. In a multi-column unique index the values can be duplicated in a single column, but the combination of column values in each row must be unique. You use a unique index to prevent duplicate values and you often define the index after a table has been created. Below is how you can create unique index:

```
1. CREATE UNIQUE INDEX index_name
   ON table_name(index_column_1,index_column_2,…);
Or
```

```
2. CREATE TABLE table_name(
   column_1 CHAR(30) NOT NULL,
   UNIQUE INDEX(column_2));
Or

3. ALTER TABLE table_name ADD UNIQUE INDEX(column_1, column_2)
```

2.2.2 Primary Key

A **primary key** is a unique index in which no value can be NULL. Every row must have a value for the column or combination of columns. You would usually define a primary key on the smallest number of columns possible because of this, and most of the time a primary key will be set on a single column. Also, once set the column values in the primary key can't be changed. Below is how you can create primary key index.

```
1. CREATE TABLE table_name(
   column_1 datatype PRIMARY KEY );
Or
2. CREATE TABLE table_name(
   column_1 CHAR(30) NOT NULL,
   column_2 CHAR(30) NOT NULL,
   PRIMARY KEY(column_1, column_2));
Or

3. ALTER TABLE table_name ADD PRIMARY KEY(column_1, column_2);
```

2.2.3 Simple, Regular, Normal

A **simple, regular, or normal index** is an index where the values don't need to be unique and they can be NULL. They're added simply to help the database search for things faster. Below is how you can create normal index.

```
1. CREATE INDEX index_name
   ON table_name(index_column_1,index_column_2,...);
Or
```

```
2. CREATE TABLE table_name(
   column_1 CHAR(30) NOT NULL,
   INDEX(column_1));
Or

3. ALTER TABLE table_name ADD INDEX(column_1, column_2);
```

2.2.4 Full-text

A **full-text index**, as the name implies, are used for full text searches. Sometimes you want to find the blob of text that contains a certain word or group of words or maybe you want to find a certain substring within the larger block of text. It is widely used in e-commerce and search engines. `FULLTEXT` indexes are supported only for `InnoDB` and `MyISAM` tables and can include only `CHAR`, `VARCHAR`, and `TEXT` columns.

Below is how you can create full-text index.

```
1. CREATE FULLTEXT INDEX index_name
   ON table_name(index_column_1,index_column_2,...);
Or

2. CREATE TABLE table_name(
   column_1 TEXT,
   FULLTEXT(column_1));
Or

3. ALTER TABLE table_name ADD FULLTEXT (column_1, column_2);
```

2.2.5 Spatial

Spatial indexes are new in MySQL and are not widely used. MySQL permits creation of `SPATIAL` indexes on `NOT NULL` geometry-valued columns. `SPATIAL INDEX` creates an R-tree index. For storage engines that support nonspatial indexing of spatial columns, the engine creates a B-tree index. A B-tree index on spatial values is useful for exact-value lookups, but not for range scans. The optimizer can use spatial indexes defined on columns that are SRID-restricted.

Below is how you can create spatial index.

```
1. CREATE SPATIAL INDEX index_name
   ON table_name(index_column_1);
Or

2. CREATE TABLE table_name(
   column_1 GEOMETRY NOT NULL SRID 4326,
   SPATIAL INDEX(column_2));
Or

3. ALTER TABLE table_name ADD SPATIAL INDEX(column_1, column_2);
```

2.2.6 Descending

A **descending index** which is available only 8+ versionof MySQL, is a regular index stored in reverse order. It’s helpful when you run queries for the most recently added data like you might to show your five most recent posts or the ten most recent comments on all your posts. Previously, indexes could be scanned in reverse order but performance was a problem. A descending index can be scanned in forward order, which is more efficient. Descending indexes also make it possible for the optimizer to use multiple-column indexes when the most efficient scan order mixes ascending order for some columns and descending order for others. Below is how you can create descending index.

```
1. CREATE INDEX index_name
   ON table_name(index_column_1 DESC);
Or

2. CREATE TABLE table_name(
   column_1 INT,
   column_2 INT,
   INDEX asc_index_1 (column_1 ASC, column_2 ASC),
   INDEX desc_index_1 (column_1 DESC, column_2 DESC));
Or
```

```
3. ALTER TABLE table_name ADD INDEX(column_1 DESC, column_2
ASC);
```

Introduction to MySQL USE INDEX hint

In MySQL, when you submit an SQL query, the query optimizer will try to make an optimal query execution plan.

To determine the best possible plan, the query optimizer makes use of many parameters. One of the most important parameters for choosing which index to use is stored key distribution which is also known as cardinality.

The cardinality, however, may be not accurate for example in case the table has been modified heavily with many inserts or deletes.

To solve this issue, you should run the ANALYZE TABLE statement periodically to update the cardinality.

In addition, MySQL provides an alternative way that allows you to recommend the indexes that the query optimizer should by using an index hint called USE INDEX.

The following illustrates syntax of the MySQL USE INDEX hint:

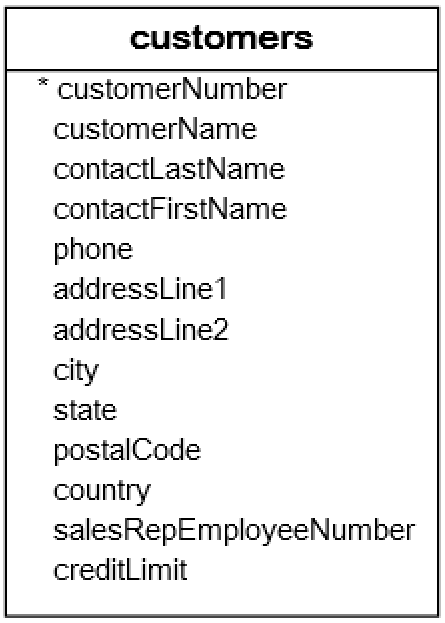
```
SELECT select_list
FROM table_name USE INDEX(index_list)
WHERE condition;
```
Code language: SQL (Structured Query Language) (sql)

In this syntax, the USE INDEX instructs the query optimizer to use one of the named indexes to find rows in the table.

Notice that when you recommend the indexes to use, the query optimizer may either decide to use them or not depending on the query plan that it comes up with.

MySQL USE INDEX example

We will use the `customers` table from the sample database for the demonstration.



customers

* customerNumber
customerName
contactLastName
contactFirstName
phone
addressLine1
addressLine2
city
state
postalCode
country
salesRepEmployeeNumber
creditLimit

First, use the r statement to display all indexes of the customers table:

```sql
SHOW INDEXES FROM customers;
```

Code language: SQL (Structured Query Language) (sql)

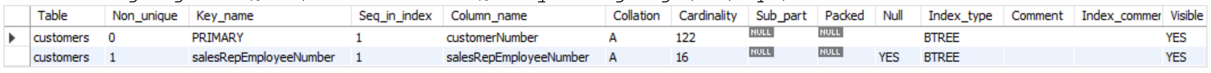


	Table	Non_unique	Key_name	Seq_in_index	Column_name	Collation	Cardinality	Sub_part	Packed	Null	Index_type	Comment	Index_commer	Visible
▶	customers	0	PRIMARY	1	customerNumber	A	122	NULL	NULL		BTREE			YES
	customers	1	salesRepEmployeeNumber	1	salesRepEmployeeNumber	A	16	NULL	NULL	YES	BTREE			YES

Second, create four indexes as follows:

```sql
CREATE INDEX idx_c_ln  ON customers(contactLastName);
CREATE INDEX idx_c_fn ON customers(contactFirstName);
CREATE INDEX idx_name_fl  ON customers(contactFirstName,contactLastName);
CREATE INDEX idx_name_lf  ON customers(contactLastName,contactFirstName);
```

Code language: SQL (Structured Query Language) (sql)

Third, find customers whose contact first name or contact last name starts with the letter A. Use the EXPLAIN statement check which indexes are used:

```sql
EXPLAIN SELECT *
FROM
  customers
WHERE
  contactFirstName LIKE 'A%'
    OR contactLastName LIKE 'A%';
```

The following shows the output of the statement:

```
           id: 1
  select_type: SIMPLE
        table: customers
   partitions: NULL
         type: index_merge
possible_keys: idx_c_ln,idx_c_fn,idx_name_fl,idx_name_lf
          key: idx_c_fn,idx_c_ln
```

```
     key_len: 52,52
         ref: NULL
        rows: 16
    filtered: 100.00
       Extra: Using sort_union(idx_c_fn,idx_c_ln); Using where
1 row in set, 1 warning (0.00 sec)
```
Code language: SQL (Structured Query Language) (sql)

As you can see, the Query Optimizer used the idx_c_fn and idx_c_ln indexes.

Fourth, if you think that it is better to use the idx_c_fl and idx_c_lf indexes, you use the USE INDEX clause as follows:

```
EXPLAIN SELECT *
FROM
    customers
USE INDEX (idx_name_fl, idx_name_lf)
WHERE
    contactFirstName LIKE 'A%'
        OR contactLastName LIKE 'A%';
```

Notice that this is just for the demonstration purposes, not the best choice though.

The following illustrates the output:

```
           id: 1
  select_type: SIMPLE
        table: customers
   partitions: NULL
         type: index_merge
possible_keys: idx_name_fl,idx_name_lf
          key: idx_name_fl,idx_name_lf
      key_len: 52,52
          ref: NULL
         rows: 16
     filtered: 100.00
        Extra: Using sort_union(idx_name_fl,idx_name_lf); Using where
1 row in set, 1 warning (0.00 sec)
```
Code language: SQL (Structured Query Language) (sql)

These are the changes:

- The possible_keys column only lists the indexes specified in the USE INDEX clause.
- The key column has both idx_name_fl and idx_name_lf. It means that the Query Optimizer used the recommended indexes instead.

The USE INDEX is useful in case the EXPLAIN shows that the Query Optimizer uses the wrong index from the list of possible indexes.