Indexing is an important feature of MySQL. It supports different type of indexes like primary key index, unique index, normal index and full-text index. Indexes help to speed up the retrieval of data from MySQL database server. When retrieving the data from a database table, MySQL first checks whether the index of table exists; If yes it will use index to select exact physical corresponding rows without scanning the whole table.

**Creating Indexes**

Mostly we create index when creating table. Any column in creating table statement declared as PRIMARY KEY, KEY, UNIQUE or INDEX will be indexed automatically by MySQL. In addition, you can add indexes to the tables which has data. The statement to create index in MySQL is as follows:  
CREATE [UNIQUE|FULLTEXT|SPATIAL] INDEX index\_name  
USING [BTREE | HASH | RTREE]  
ON table\_name (column\_name [(length)] [ASC | DESC],…)  
  
In above statement UNIQUE specify that MySQL will create a constraint that all values in the index must be distinct. Duplicated NULL is allowed in all storage engine except BDB.

The FULLTEXT index is supported only by MyISAM storage engine and only accepted columns which have data type is CHAR,VARCHAR or TEXT.

The SPATIAL index supports spatial column and available in MyISAM storage engine. In addition, the column value must not be NULL.

Then you name the index using index types such as BTREE, HASH or RTREE also based on storage engine

In below index create statement, I have created index to emp\_no column on employees table to make the record retreaval faster. The SQL statement to create index is as follows  
CREATE INDEX emp\_no ON employees(emp\_no)

**Advantages of MySQL Indexes**

1- Indexes make search queries much faster.

2- Indexes like primary key index and unique index help to avoid duplicate row data.

3- Full-text indexes in MySQL, users have the opportunity to optimize searching against even large amounts of text located in any field indexed as such.

**Disadvantages of MySQL indexes**

Actually a separate file created when a new index created on the table column. that file stored only the field you’re interested in sorting on. So when we create index, it takes up disk space. but because of creating index on every column in every possible combination, the index file would grow much more quickly than the data file. In the case when a table is of large table size, the index file could reach the operating system’s maximum file size.

The index also slow down the speed of writing queries, such as INSERT, UPDATE and DELETE. AS MySQL has to internally maintain the pointers to the inserted rows in the actual data file, so there is a performance price to pay in case of above said writing queries because every time a record is changed, the indexes must be updated.

So Indexes are important to speed in large MySQL databases. it doesn’t matter how small your table, a 100000-row table scan will never be fast. So If you have a site with a 100000-row table, you should really spend time analyzing possible indexes and possibly consider rewriting queries to optimize your application.

**What is an index?**

*So, what is an index? Well, an index is a data structure (most commonly a B- tree) that stores the values for a specific column in a table. An index is created on a column of a table. So, the key points to remember are that an index consists of column values from one table, and that those values are stored in a data structure. The index is a data structure – remember that.*

Let’s start out our tutorial and explanation of why you would need a database index by going through a very simple example. Suppose that we have a database table called Employee with three columns – Employee\_Name, Employee\_Age, and Employee\_Address. Assume that the Employee table has thousands of rows.

Now, let’s say that we want to run a query to find all the details of any employees who are named ‘Jesus’? So, we decide to run a simple query like this:

SELECT \* FROM Employee

WHEREEmployee\_Name = 'Jesus'

**What would happen without an index on the table?**

Once we run that query, what exactly goes on behind the scenes to find employees who are named Jesus? Well, the database software would literally have to look at every single row in the Employee table to see if the Employee\_Name for that row is ‘Jesus’. And, because we want every row with the name ‘Jesus’ inside it, we can not just stop looking once we find just one row with the name ‘Jesus’, because there could be other rows with the name Jesus. So, every row up until the last row must be searched` – which means thousands of rows in this scenario will have to be examined by the database to find the rows with the name ‘Jesus’. This is what is called a full table scan.

**How a database index can help performance**

You might be thinking that doing a full table scan sounds inefficient for something so simple – shouldn’t software be smarter? It’s almost like looking through the entire table with the human eye – very slow and not at all sleek. But, as you probably guessed by the title of this article, this is where indexes can help a great deal. The whole point of having an index is to speed up search queries by essentially cutting down the number of records/rows in a table that need to be examined.

**What kind of data structure is an index?**

*B- trees are the most commonly used data structures for indexes. The reason B- trees are the most popular data structure for indexes is due to the fact that they are time efficient – because look-ups, deletions, and insertions can all be done in logarithmic time. And, another major reason B- trees are more commonly used is because the data that is stored inside the B- tree can be sorted. The RDBMS typically determines which data structure is actually used for an index. But, in some scenarios with certain RDBMS’s, you can actually specify which data structure you want your database to use when you create the index itself.*

**How does an index improve performance?**

*Because an index is basically a data structure that is used to store column values, looking up those values becomes much faster. And, if an index is using the most commonly used data structure type – a B- tree – then the data structure is also sorted. Having the column values be sorted can be a major performance enhancement – read on to find out why. Let’s say that we create a B- tree index on the Employee\_Name column This means that when we search for employees named “Jesus” using the SQL we showed earlier, then the entire Employee table does not have to be searched to find employees named “Jesus”. Instead, the database will use the index to find employees named Jesus, because the index will presumably be sorted alphabetically by the Employee’s name. And, because it is sorted, it means searching for a name is a lot faster because all names starting with a “J” will be right next to each other in the index! It’s also important to note that the index also stores pointers to the table row so that other column values can be retrieved – read on for more details on that.*

**How to create an index in SQL:**

*Here’s what the actual SQL would look like to create an index on the Employee\_Name column from our example earlier:*

CREATEINDEXname\_index

ON Employee (Employee\_Name)

**How to create a multi-column index in SQL:**

*We could also create an index on two of the columns in the Employee table , as shown in this SQL:*

CREATEINDEXname\_index

ON Employee (Employee\_Name, Employee\_Age)

# MySQL Composite Index (Multiple-Column Indexes)

# Keep the following in mind when creating composite indices for MySQL tables:

* A composite index is by MySQL to optimise queries for columns used in WHERE, ORDER BY, GROUP BY and MIN/MAX.
* A three-column index (col1, col2, col3) can be used for search capabilities on (col1), (col1, col2) and (col1, col2, col3) (ordering doesn't matter). It cannot be used for (col2) or (col2, col3) or (col3). If the table has a multiple-column index, any leftmost prefix of the index can be used by the optimizer to find rows.
* Once you have decided you need a composite index (col1, col2, col3), the preferred ordering is to put the higher cardinality columns first. If column 2 has the highest cardinality, then you may want to create an index (col2,col1, col3) but keep in mind this cannot be used for optimising searches on just (col1), etc.
* MySQL uses only one index per query. Hence, composite indices come in handy. You can use a sub-query to work around this if needed but ideally you should have a composite index with with the relevant columns.
* If you have two separate indices on col1 and col2, MySQL will prefer the one with the highest cardinality. MySQL keeps stats on the cardinality of each index. An index with the most discriminating power allows MySQL to eliminate the maximum number of rows in the quickest time.
* If all the fields selected in a query are covered by an index, MySQL never has to hit the DB and can fetch the fields directly from the index (applies only to InnoDB). So if you select col1 from table1 and col1 is indexed, MySQL will use a covering index and never hit the table.