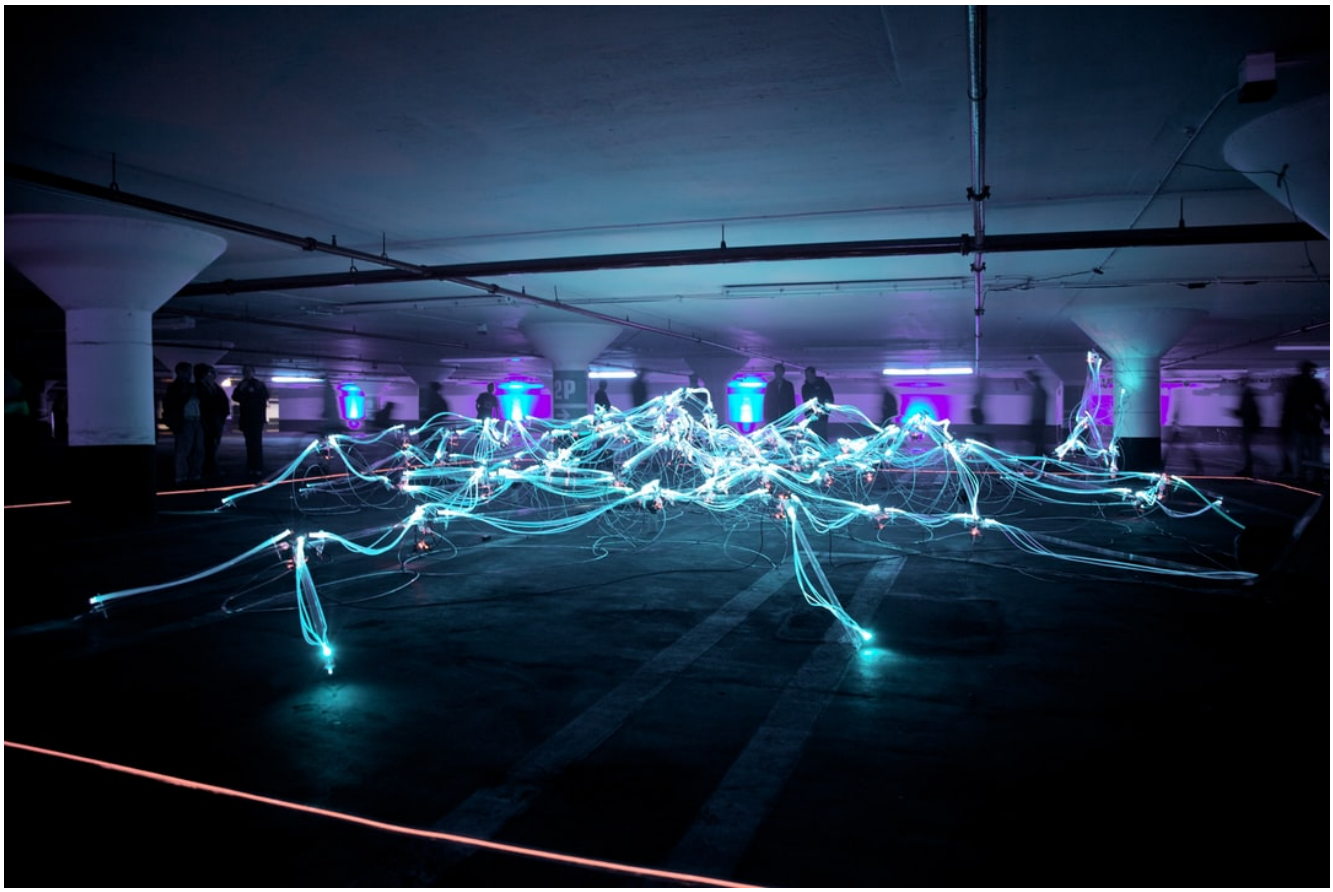
16 APRIL 2019 / #PROGRAMMING

An in-depth look at Database Indexing

Kousik Nath
Engineer @ PayPal, loves to have deep discussion on distributed and scalable systems, system architecture, design patterns, algorithmic problem solving. Linkedin: https://www.linkedin.com/in/kousikn/



Performance is extremely important in many consumer products like e-commerce, payment systems, gaming, transportation apps, etc. Although databases are internally optimised through multiple mechanisms to meet their performance requirements in the modern world, a lot depends on the application

what queries the application has to perform.

Developers who deal with relational databases have used or at least heard about indexing, and it's a very common concept in the database world. However, the most important part is to understand what to index & how the indexing is going to boost the query response time. For doing that you need to understand how you are going to query your database tables. A proper index can be created only when you know exactly what your query & data access patterns look like.

In simple terminology, an index maps search keys to corresponding data on disk by using different in-memory & on-disk data structures. Index is used to quicken the search by reducing the number of records to search for.

Mostly an index is created on the columns specified in the `WHERE` clause of a query as the database retrieves & filters data from the tables based on those columns. If you don't create an index, the database scans all the rows, filters out the matching rows & returns the result. With millions of records, this scan operation may take many seconds & this high response time makes APIs & applications slower & unusable. Let's see an example —

We will use MySQL with a default InnoDB database engine, although concepts explained in this article are more or less same in other database servers as well like Oracle, MSSQL etc.

Create a table called `index_demo` with the following schema:

```
CREATE TABLE index_demo (
    name VARCHAR(20) NOT NULL,
    age INT,
    pan_no VARCHAR(20),
```

How do we verify that we are using InnoDB engine?

Run the below command:

```
SHOW TABLE STATUS WHERE name = 'index_demo' \G;
```

```
mysql> SHOW TABLE STATUS WHERE name = 'index_demo' \G;
*************************** 1. row ***************************
           Name: index_demo
         Engine: InnoDB
        Version: 10
     Row_format: Dynamic
           Rows: 2
 Avg_row_length: 8192
    Data_length: 16384
Max_data_length: 0
   Index_length: 49152
      Data_free: 0
 Auto_increment: NULL
    Create_time: 2019-04-16 18:04:09
    Update_time: 2019-04-16 16:29:04
     Check_time: NULL
      Collation: utf8_general_ci
       Checksum: NULL
 Create_options:
        Comment:
1 row in set (0.00 sec)
```

The `Engine` column in the above screen shot represents the engine that is used to create the table. Here `InnoDB` is used.

Now Insert some random data in the table, my table with 5 rows looks like the following:

```
mysql> select * from index_demo;
+---------+------+------------+------------+
| name    | age  | pan_no     | phone_no   |
+---------+------+------------+------------+
| kousik  |   27 | IPOET0935V | 9281920292 |
| alex    |   26 | MNWTT0935V | 9281748482 |
| francis |   20 | OPETV0915E | 9281092482 |
| tom     |   40 | OPETV8935E | 9281072002 |
| harry   |   50 | IEYTV8935E | 0993372002 |
+---------+------+------------+------------+
5 rows in set (0.01 sec)
```

I have not created any index till now on this table. Let's verify this by the command: `SHOW INDEX` . It returns 0 results.

```
mysql> show index from index_demo;
Empty set (0.00 sec)
```

At this moment, if we run a simple `SELECT` query, since there is no user defined index, the query will scan the whole table to find out the result:

```
EXPLAIN SELECT * FROM index_demo WHERE name = 'alex';
```

```
+----+-------------+------------+------------+------+---------------+------+---------+------+------+----------+-------------+
| id | select_type | table      | partitions | type | possible_keys | key  | key_len | ref  | rows | filtered | Extra       |
+----+-------------+------------+------------+------+---------------+------+---------+------+------+----------+-------------+
|  1 | SIMPLE      | index_demo | NULL       | ALL  | NULL          | NULL | NULL    | NULL |    5 |    20.00 | Using where |
+----+-------------+------------+------------+------+---------------+------+---------+------+------+----------+-------------+
1 row in set, 1 warning (0.10 sec)
```

`EXPLAIN` shows how the query engine plans to execute the query. In the above screenshot, you can see that the `rows` column returns `5` & `possible_keys` returns `null`. `possible_keys` represents what all available indices are there which can be used in this query. The `key` column represents which index is actually going to be used out of all possible indices in this query.

Primary Key:

The above query is very in efficient. Let's optimise this query. We will make the `phone_no` column a `PRIMARY KEY` assuming that no two users can exist in our system with same phone number. Take the following into consideration when creating a primary key:

- A primary key should be part of many vital queries in your application.
- Primary key is a constraint that uniquely identifies each row in a table. If multiple columns are part of the primary key, that combination should be unique for each row.
- Primary key should be Non-null. Never make null-able fields your primary key. By ANSI SQL standards, primary keys should be comparable to each other, and you should definitely be able to tell whether the primary key column value for a particular row is greater, smaller or equal to the same from other row. Since `NULL` means an undefined value in SQL standards, you can't deterministically compare `NULL` with any other value, so logically `NULL` is not allowed.
- The ideal primary key type should be a number like `INT` or `BIGINT` because integer comparisons are faster, so

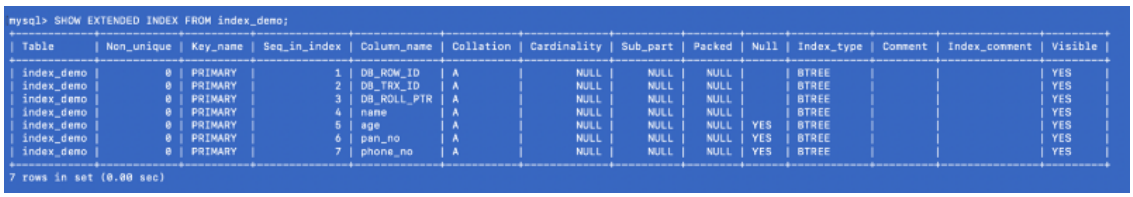
Often we define an `id` field as `AUTO INCREMENT` in tables & use that as a primary key, but the choice of a primary key depends on developers.

What if you don't create any primary key yourself?

It's not mandatory to create a primary key yourself. If you have not defined any primary key, InnoDB implicitly creates one for you because InnoDB by design must have a primary key in every table. So once you create a primary key later on for that table, InnoDB deletes the previously auto defined primary key.

Since we don't have any primary key defined as of now, let's see what InnoDB by default created for us:

```
SHOW EXTENDED INDEX FROM index_demo;
```



```
mysql> SHOW EXTENDED INDEX FROM index_demo;
+------------+------------+----------+--------------+-------------+-----------+-------------+----------+--------+------+------------+---------+---------------+---------+
| Table      | Non_unique | Key_name | Seq_in_index | Column_name | Collation | Cardinality | Sub_part | Packed | Null | Index_type | Comment | Index_comment | Visible |
+------------+------------+----------+--------------+-------------+-----------+-------------+----------+--------+------+------------+---------+---------------+---------+
| index_demo |          0 | PRIMARY  |            1 | DB_ROW_ID   | A         |        NULL |     NULL |   NULL |      | BTREE      |         |               | YES     |
| index_demo |          0 | PRIMARY  |            2 | DB_TRX_ID   | A         |        NULL |     NULL |   NULL |      | BTREE      |         |               | YES     |
| index_demo |          0 | PRIMARY  |            3 | DB_ROLL_PTR | A         |        NULL |     NULL |   NULL |      | BTREE      |         |               | YES     |
| index_demo |          0 | PRIMARY  |            4 | name        | A         |        NULL |     NULL |   NULL |      | BTREE      |         |               | YES     |
| index_demo |          0 | PRIMARY  |            5 | age         | A         |        NULL |     NULL |   NULL | YES  | BTREE      |         |               | YES     |
| index_demo |          0 | PRIMARY  |            6 | pan_no      | A         |        NULL |     NULL |   NULL | YES  | BTREE      |         |               | YES     |
| index_demo |          0 | PRIMARY  |            7 | phone_no    | A         |        NULL |     NULL |   NULL | YES  | BTREE      |         |               | YES     |
+------------+------------+----------+--------------+-------------+-----------+-------------+----------+--------+------+------------+---------+---------------+---------+
7 rows in set (0.00 sec)
```

`EXTENDED` shows all the indices that are not usable by the user but managed completely by MySQL.

Here we see that MySQL has defined a composite index (we will discuss composite indices later) on `DB_ROW_ID` , `DB_TRX_ID` , `DB_ROL`

defined primary key, this index is used to find records uniquely.

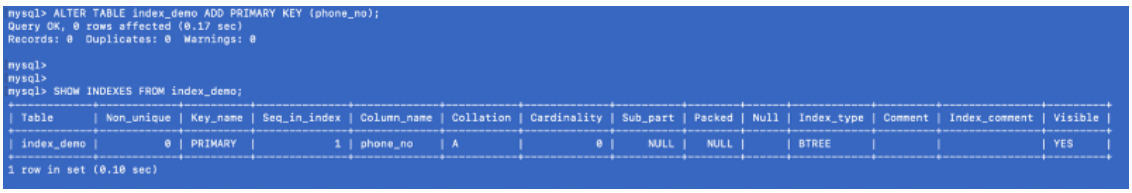
What is the difference between key & index?

Although the terms `key` & `index` are used interchangeably, `key` means a constraint imposed on the behaviour of the column. In this case, the constraint is that primary key is non null-able field which uniquely identifies each row. On the other hand, `index` is a special data structure that facilitates data search across the table.

Let's now create the primary index on `phone_no` & examine the created index:

```
ALTER TABLE index_demo ADD PRIMARY KEY (phone_no);
SHOW INDEXES FROM index_demo;
```

Note that `CREATE INDEX` can not be used to create a primary index, but `ALTER TABLE` is used.



In the above screenshot, we see that one primary index is created on the column `phone_no` . The columns of the following images are described as follows:

`Table` : The table on which the index is created.

`Non_unique` : If the value is 1, the index is not unique, if the value is 0, the index is unique.

index is always `PRIMARY` in MySQL, irrespective of if you have provided any index name or not while creating the index.

`Seq_in_index` : The sequence number of the column in the index. If multiple columns are part of the index, the sequence number will be assigned based on how the columns were ordered during the index creation time. Sequence number starts from 1.

`Collation` : how the column is sorted in the index. `A` means ascending, `D` means descending, `NULL` means not sorted.

`Cardinality` : The estimated number of unique values in the index. More cardinality means higher chances that the query optimizer will pick the index for queries.

`Sub_part` : The index prefix. It is `NULL` if the entire column is indexed. Otherwise, it shows the number of indexed bytes in case the column is partially indexed. We will define partial index later.

`Packed` : Indicates how the key is packed; `NULL` if it is not.

`Null` : `YES` if the column may contain `NULL` values and blank if it does not.

`Index_type` : Indicates which indexing data structure is used for this index. Some possible candidates are — `BTREE` , `HASH` , `RTREE` , or `FULLTEXT` .

`Comment` : The information about the index not described in its own column.

`Index_comment` : The comment for the index specified when you created the index with the `COMMENT` attribute.

searched for a given `phone_no` in the `WHERE` clause of a query.

```
EXPLAIN SELECT * FROM index_demo WHERE phone_no = '9281072002';
```

```
mysql> EXPLAIN SELECT * FROM index_demo WHERE phone_no = '9281072002';
+----+-------------+------------+------------+-------+---------------+---------+---------+-------+------+----------+-------+
| id | select_type | table      | partitions | type  | possible_keys | key     | key_len | ref   | rows | filtered | Extra |
+----+-------------+------------+------------+-------+---------------+---------+---------+-------+------+----------+-------+
|  1 | SIMPLE      | index_demo | NULL       | const | PRIMARY       | PRIMARY | 22      | const |    1 |   100.00 | NULL  |
+----+-------------+------------+------------+-------+---------------+---------+---------+-------+------+----------+-------+
1 row in set, 1 warning (0.00 sec)
```

In this snapshot, notice that the `rows` column has returned `1` only, the `possible_keys` & `key` both returns `PRIMARY` . So it essentially means that using the primary index named as `PRIMARY` (the name is auto assigned when you create the primary key), the query optimizer just goes directly to the record & fetches it. It's very efficient. This is exactly what an index is for — to minimize the search scope at the cost of extra space.

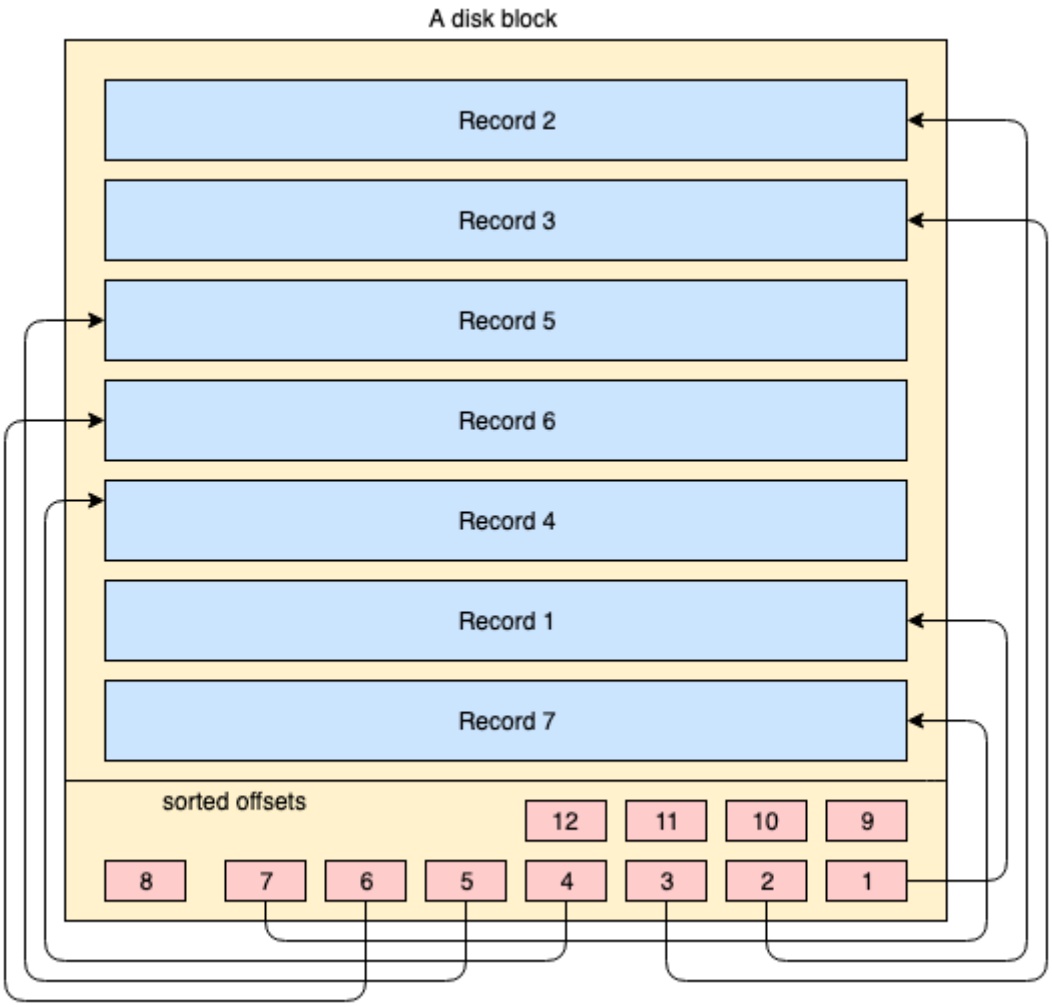
Clustered Index:

A `clustered index` is collocated with the data in the same table space or same disk file. You can consider that a clustered index is a `B-Tree` index whose leaf nodes are the actual data blocks on disk, since the index & data reside together. This kind of index physically organizes the data on disk as per the logical order of the index key.

What does physical data organization mean?

Physically, data is organized on disk across thousands or millions of disk / data blocks. For a clustered index, it's not mandatory that all the disk blocks are contagiously stored. Physical data blocks are all

necessary. A database system does not have any absolute control over how physical data space is managed, but inside a data block, records can be stored or managed in the logical order of the index key. The following simplified diagram explains it:



- The yellow coloured big rectangle represents a disk block / data block
- the blue coloured rectangles represent data stored as rows inside that block
- the footer area represents the index of the block where red coloured small rectangles reside in sorted order of a particular key. These small blocks are nothing but sort of pointers pointing to offsets of the records.

Records are stored on the disk block in any arbitrary order. Whenever new records are added, they get added in the next available space. Whenever an existing record is updated, the OS decides whether that record can still fit into the same position or a new position has to be allocated for that record.

So position of records are completely handled by OS & no definite relation exists between the order of any two records. In order to fetch the records in the logical order of key, disk pages contain an index section in the footer, the index contains a list of offset pointers in the order of the key. Every time a record is altered or created, the index is adjusted.

In this way, you really don't need to care about actually organizing the physical record in a certain order, rather a small index section is maintained in that order & fetching or maintaining records becomes very easy.

Advantage of Clustered Index:

This ordering or co-location of related data actually makes a clustered index faster. When data is fetched from disk, the complete block containing the data is read by the system since our disk IO system writes & reads data in blocks. So in case of range queries, it's

you fire the following query:

```
SELECT * FROM index_demo WHERE phone_no > '9010000000' AND phone_
```

A data block is fetched in memory when the query is executed. Say the data block contains `phone_no` in the range from `9010000000` to `9030000000` . So whatever range you requested for in the query is just a subset of the data present in the block. If you now fire the next query to get all the phone numbers in the range, say from `9015000000` to `9019000000` , you don't need to fetch any more blocks from the disk. The complete data can be found in the current block of data, thus `clustered_index` reduces the number of disk IO by collocating related data as much as possible in the same data block. This reduced disk IO causes improvement in performance.

So if you have a well thought of primary key & your queries are based on the primary key, the performance will be super fast.

Constraints of Clustered Index:

Since a clustered index impacts the physical organization of the data, there can be only one clustered index per table.

Relationship between Primary Key & Clustered Index:

You can't create a clustered index manually using InnoDB in MySQL. MySQL chooses it for you. But how does it choose? The following excerpts are from MySQL documentation:

> When you define a `PRIMARY KEY` on your table, `InnoDB` uses it as the clustered index. Define a primary key for each table that you create. If there is no logical unique and non-null

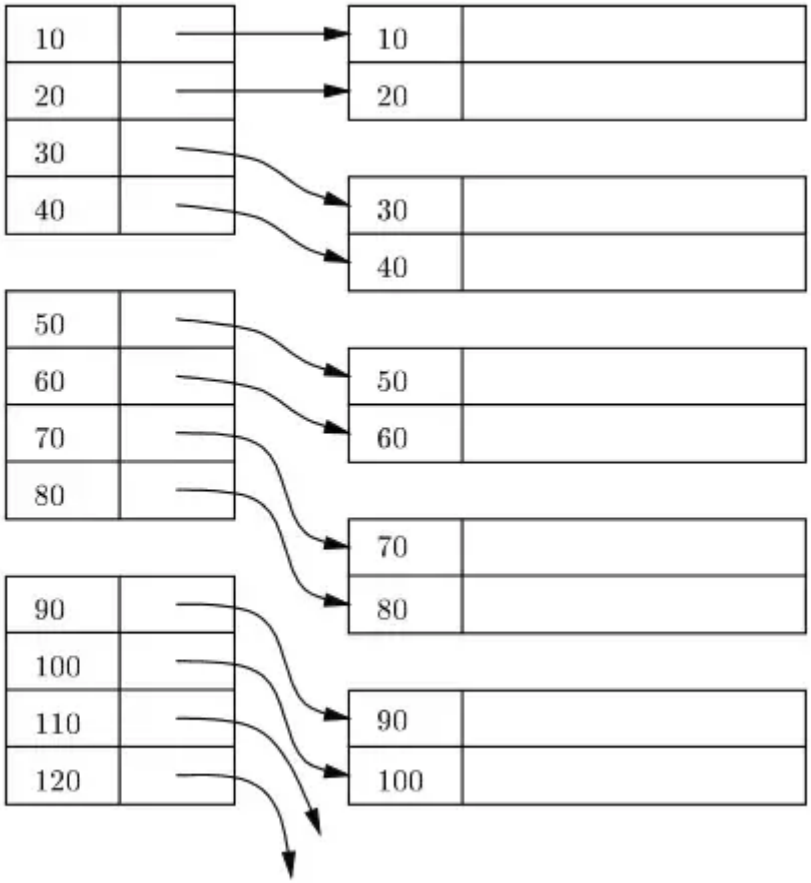
> whose values are filled in automatically.
>
> If you do not define a `PRIMARY KEY` for your table, MySQL locates the first `UNIQUE` index where all the key columns are `NOT NULL` and `InnoDB` uses it as the clustered index.
>
> If the table has no `PRIMARY KEY` or suitable `UNIQUE` index, `InnoDB` internally generates a hidden clustered index named `GEN_CLUST_INDEX` on a synthetic column containing row ID values. The rows are ordered by the ID that `InnoDB` assigns to the rows in such a table. The row ID is a 6-byte field that increases monotonically as new rows are inserted. Thus, the rows ordered by the row ID are physically in insertion order.

In short, the MySQL InnoDB engine actually manages the primary index as clustered index for improving performance, so the primary key & the actual record on disk are clustered together.

Structure of Primary key (clustered) Index:

An index is usually maintained as a B+ Tree on disk & in-memory, and any index is stored in blocks on disk. These blocks are called index blocks. The entries in the index block are always sorted on the index/search key. The leaf index block of the index contains a row locator. For the primary index, the row locator refers to virtual address of the corresponding physical location of the data blocks on disk where rows reside being sorted as per the index key.

In the following diagram, the left side rectangles represent leaf level index blocks, and the right side rectangles represent the data blocks. Logically the data blocks look to be aligned in a sorted order, but as already described earlier, the actual physical locations may be scattered here & there.



Is it possible to create a primary index on a non-primary key?

In MySQL, a primary index is automatically created, and we have already described above how MySQL chooses the primary index. But in the database world, it's actually not necessary to create an

on any non primary key column as well. But when created on the primary key, all key entries are unique in the index, while in the other case, the primary index may have a duplicated key as well.

Is it possible to delete a primary key?

It's possible to delete a primary key. When you delete a primary key, the related clustered index as well as the uniqueness property of that column gets lost.

```
ALTER TABLE `index_demo` DROP PRIMARY KEY;

- If the primary key does not exist, you get the following error:

"ERROR 1091 (42000): Can't DROP 'PRIMARY'; check that column/key
```

Advantages of Primary Index:

- Primary index based range queries are very efficient. There might be a possibility that the disk block that the database has read from the disk contains all the data belonging to the query, since the primary index is clustered & records are ordered physically. So the locality of data can be provided by the primary index.
- Any query that takes advantage of primary key is very fast.

Disadvantages of Primary Index:

- Since the primary index contains a direct reference to the data block address through the virtual address space & disk blocks are physically organized in the order of the index key, every time the OS does some disk page split due to `DML` operations like `INSERT` / `UPDATE` / `DELETE` , the primary index

pressure on the performance of the primary index.

Secondary Index:

Any index other than a clustered index is called a secondary index. Secondary indices does not impact physical storage locations unlike primary indices.

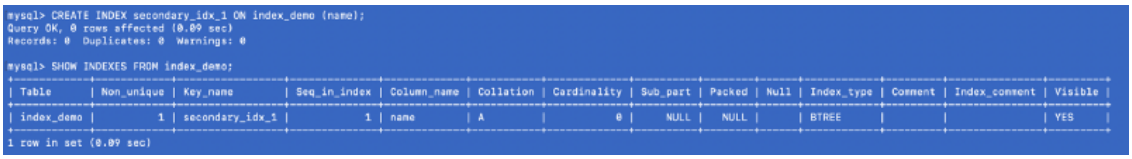
When do you need a Secondary Index?

You might have several use cases in your application where you don't query the database with a primary key. In our example `phone_no` is the primary key but we may need to query the database with `pan_no` , or `name` . In such cases you need secondary indices on these columns if the frequency of such queries is very high.

How to create a secondary index in MySQL?

The following command creates a secondary index in the `name` column in the `index_demo` table.

```
CREATE INDEX secondary_idx_1 ON index_demo (name);
```

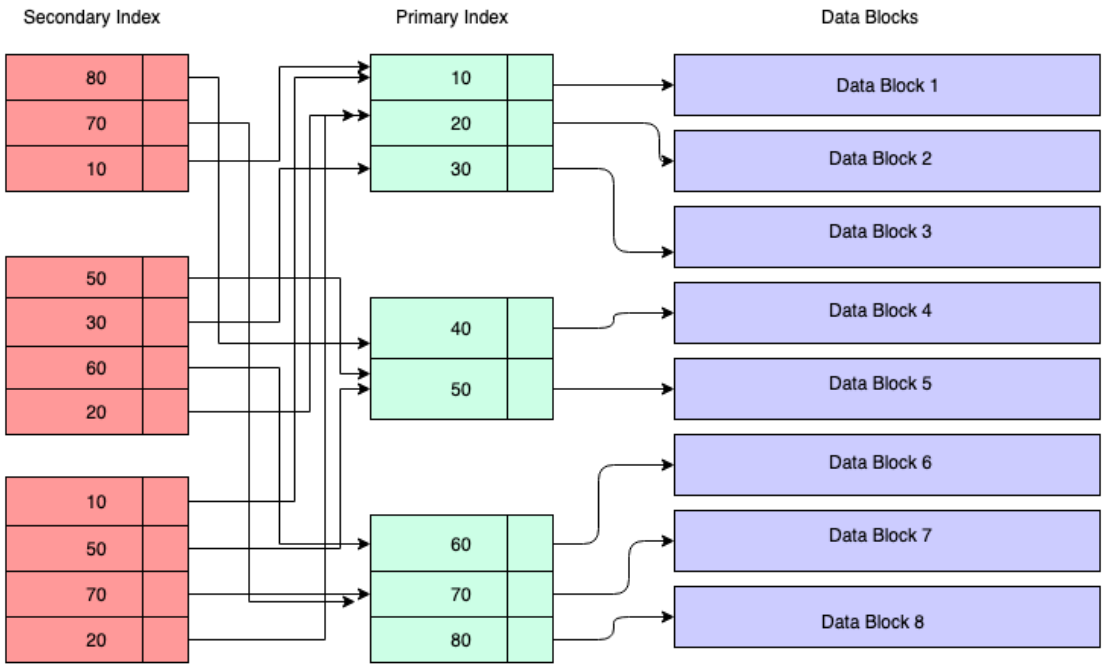


Structure of Secondary Index:

In the diagram below, the red coloured rectangles represent secondary index blocks. Secondary index is also maintained in the

created. The leaf nodes contain a copy of the key of the corresponding data in the primary index.

So to understand, you can assume that the secondary index has reference to the primary key's address, although it's not the case. Retrieving data through the secondary index means you have to traverse two B+ trees — one is the secondary index B+ tree itself, and the other is the primary index B+ tree.



Logically you can create as many secondary indices as you want. But in reality how many indices actually required needs a serious thought process since each index has its own penalty.

Disadvantages of a Secondary Index:

With `DML` operations like `DELETE / INSERT` , the secondary index also needs to be updated so that the copy of the primary key column can be deleted / inserted. In such cases, the existence of lot of secondary indexes can create issues.

Also, if a primary key is very large like a `URL` , since secondary indexes contain aa copy of the primary key column value, it can be inefficient in terms of storage. More secondary keys means a greater number of duplicate copies of the primary key column value, so more storage in case of a large primary key. Also the primary key itself stores the keys, so the combined effect on storage will be very high.

Consideration before you delete a Primary Index:

In MySQL, you can delete a primary index by dropping the primary key. We have already seen that a secondary index depends on a primary index. So if you delete a primary index, all secondary indices have to be updated to contain a copy of the new primary index key which MySQL auto adjusts.

This process is expensive when several secondary indexes exist. Also other tables may have a foreign key reference to the primary key, so you need to delete those foreign key references before you delete the primary key.

When a primary key is deleted, MySQL automatically creates

UNIQUE Key Index:

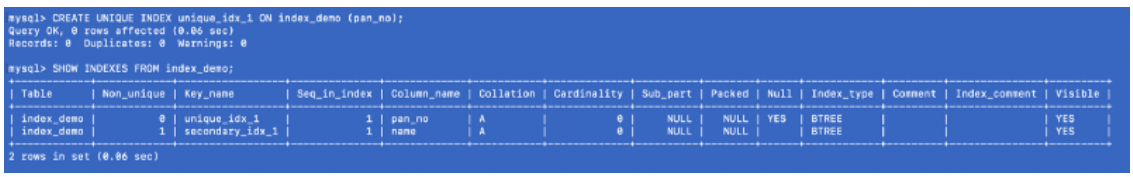
Like primary keys, unique keys can also identify records uniquely with one difference — the unique key column can contain `null` values.

Unlike other database servers, in MySQL a unique key column can have as many `null` values as possible. In SQL standard, `null` means an undefined value. So if MySQL has to contain only one `null` value in a unique key column, it has to assume that all null values are the same.

But logically this is not correct since `null` means undefined — and undefined values can't be compared with each other, it's the nature of `null` . As MySQL can't assert if all `null` s mean the same, it allows multiple `null` values in the column.

The following command shows how to create a unique key index in MySQL:

```
CREATE UNIQUE INDEX unique_idx_1 ON index_demo (pan_no);
```



Composite Index:

MySQL lets you define indices on multiple columns, up to 16 columns. This index is called a Multi-column / Composite /

Let's say we have an index defined on 4 columns — `col1`, `col2`, `col3`, `col4`. With a composite index, we have search capability on `col1`, `(col1, col2)`, `(col1, col2, col3)`, `(col1, col2, col3, col4)`. So we can use any left side prefix of the indexed columns, but we can't omit a column from the middle & use that like — `(col1, col3)` or `(col1, col2, col4)` or `col3` or `col4` etc. These are invalid combinations.

The following commands create 2 composite indexes in our table:

```
CREATE INDEX composite_index_1 ON index_demo (phone_no, name, age

CREATE INDEX composite_index_2 ON index_demo (pan_no, name, age);
```

If you have queries containing a `WHERE` clause on multiple columns, write the clause in the order of the columns of the composite index. The index will benefit that query. In fact, while deciding the columns for a composite index, you can analyze different use cases of your system & try to come up with the order of columns that will benefit most of your use cases.

Composite indices can help you in `JOIN` & `SELECT` queries as well. Example: in the following `SELECT *` query, `composite_index_2` is used.

```
mysql> EXPLAIN SELECT * FROM index_demo;
+----+-------------+------------+------------+-------+---------------+-------------------+---------+------+------+----------+-------------+
| id | select_type | table      | partitions | type  | possible_keys | key               | key_len | ref  | rows | filtered | Extra       |
+----+-------------+------------+------------+-------+---------------+-------------------+---------+------+------+----------+-------------+
|  1 | SIMPLE      | index_demo | NULL       | index | NULL          | composite_index_2 | 130     | NULL |    2 |   100.00 | Using index |
+----+-------------+------------+------------+-------+---------------+-------------------+---------+------+------+----------+-------------+
1 row in set, 1 warning (0.00 sec)
```

When several indexes are defined, the MySQL query optimizer chooses that index which eliminates the greatest number of rows or

Why do we use composite indices? Why not define multiple secondary indices on the columns we are interested in?

MySQL uses only one index per table per query except for UNION. (In a UNION, each logical query is run separately, and the results are merged.) So defining multiple indices on multiple columns does not guarantee those indices will be used even if they are part of the query.

MySQL maintains something called index statistics which helps MySQL infer what the data looks like in the system. Index statistics is a generilization though, but based on this meta data, MySQL decides which index is appropriate for the current query.

How does composite index work?

The columns used in composite indices are concatenated together, and those concatenated keys are stored in sorted order using a B+ Tree. When you perform a search, concatenation of your search keys is matched against those of the composite index. Then if there is any mismatch between the ordering of your search keys & ordering of the composite index columns, the index can't be used.

In our example, for the following record, a composite index key is formed by concatenating `pan_no` , `name` , `age` — `HJKXS9086Wkousik 28` .

```
+--------+------+------------+------------+
 name
 age
 pan_no
 phone_no
```

```
  28
HJKXS9086W
9090909090
```

How to identify if you need a composite index:

- Analyze your queries first according to your use cases. If you see certain fields are appearing together in many queries, you may consider creating a composite index.
- If you are creating an index in `col1` & a composite index in (`col1` , `col2`), then only the composite index should be fine. `col1` alone can be served by the composite index itself since it's a left side prefix of the index.
- Consider cardinality. If columns used in the composite index end up having high cardinality together, they are good candidate for the composite index.

Covering Index:

A covering index is a special kind of composite index where all the columns specified in the query somewhere exist in the index. So the query optimizer does not need to hit the database to get the data — rather it gets the result from the index itself. Example: we have already defined a composite index on `(pan_no, name, age)` , so now consider the following query:

```
SELECT age FROM index_demo WHERE pan_no = 'HJKXS9086W' AND name =
```

The columns mentioned in the `SELECT` & `WHERE` clauses are part of

the `age` column from the composite index itself. Let's see what the `EXPLAIN` command shows for this query:

```
EXPLAIN FORMAT=JSON SELECT age FROM index_demo WHERE pan_no = 'H
```

```
mysql> EXPLAIN FORMAT=JSON SELECT age FROM index_demo WHERE pan_no = 'HJKXS9086W' AND name = '111kousik1';
+--------------------------------------------------------------------------------------------------------------
--------------------------------------------------------------------------------------------------------------
--------------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------+
| EXPLAIN


                                                                               |
+--------------------------------------------------------------------------------------------------------------
--------------------------------------------------------------------------------------------------------------
--------------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------+
| {
  "query_block": {
    "select_id": 1,
    "cost_info": {
      "query_cost": "0.35"
    },
    "table": {
      "table_name": "index_demo",
      "access_type": "ref",
      "possible_keys": [
        "composite_index_2"
      ],
      "key": "composite_index_2",
      "used_key_parts": [
        "pan_no",
        "name"
      ],
      "key_length": "125",
      "ref": [
        "const",
        "const"
      ],
      "rows_examined_per_scan": 1,
      "rows_produced_per_join": 1,
      "filtered": "100.00",
      "using_index": true,
      "cost_info": {
        "read_cost": "0.25",
        "eval_cost": "0.10",
        "prefix_cost": "0.35",
        "data_read_per_join": "192"
      },
      "used_columns": [
        "name",
        "age",
        "pan_no"
      ]
    }
  }
} |
```

In the above response, note that there is a key — `using_index` which is set to `true` which signifies that the covering index has been used to answer the query.

I don't know how much covering indices are appreciated in production environments, but apparently it seems to be a good optimization in case the query fits the bill.

Partial Index:

We already know that Indices speed up our queries at the cost of space. The more indices you have, the more the storage requirement. We have already created an index called `secondary_idx_1` on the column `name`. The column `name` can contain large values of any length. Also in the index, the row locators' or row pointers' metadata have their own size. So overall, an index can have a high storage & memory load.

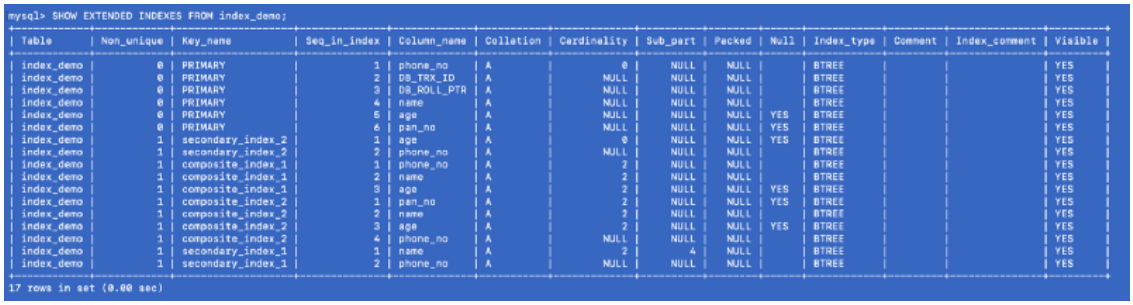
In MySQL, it's possible to create an index on the first few bytes of data as well. Example: the following command creates an index on the first 4 bytes of name. Though this method reduces memory overhead by a certain amount, the index can't eliminate many rows, since in this example the first 4 bytes may be common across many names. Usually this kind of prefix indexing is supported on `CHAR`, `VARCHAR`, `BINARY`, `VARBINARY` type of columns.

```
CREATE INDEX secondary_index_1 ON index_demo (name(4));
```

define an index?

Let's run the `SHOW EXTENDED` command again:

```
SHOW EXTENDED INDEXES FROM index_demo;
```



```
mysql> SHOW EXTENDED INDEXES FROM index_demo;
+------------+------------+-------------------+--------------+-------------+-----------+-------------+----------+--------+------+------------+---------+---------------+---------+
| Table      | Non_unique | Key_name          | Seq_in_index | Column_name | Collation | Cardinality | Sub_part | Packed | Null | Index_type | Comment | Index_comment | Visible |
+------------+------------+-------------------+--------------+-------------+-----------+-------------+----------+--------+------+------------+---------+---------------+---------+
| index_demo |          0 | PRIMARY           |            1 | phone_no    | A         |           0 |     NULL |   NULL |      | BTREE      |         |               | YES     |
| index_demo |          0 | PRIMARY           |            2 | DB_TRX_ID   | A         |        NULL |     NULL |   NULL |      | BTREE      |         |               | YES     |
| index_demo |          0 | PRIMARY           |            3 | DB_ROLL_PTR | A         |        NULL |     NULL |   NULL |      | BTREE      |         |               | YES     |
| index_demo |          0 | PRIMARY           |            4 | name        | A         |        NULL |     NULL |   NULL |      | BTREE      |         |               | YES     |
| index_demo |          0 | PRIMARY           |            5 | age         | A         |        NULL |     NULL |   NULL | YES  | BTREE      |         |               | YES     |
| index_demo |          0 | PRIMARY           |            6 | pan_no      | A         |        NULL |     NULL |   NULL | YES  | BTREE      |         |               | YES     |
| index_demo |          1 | secondary_index_2 |            1 | age         | A         |           0 |     NULL |   NULL | YES  | BTREE      |         |               | YES     |
| index_demo |          1 | secondary_index_2 |            2 | phone_no    | A         |        NULL |     NULL |   NULL |      | BTREE      |         |               | YES     |
| index_demo |          1 | composite_index_1 |            1 | phone_no    | A         |           2 |     NULL |   NULL |      | BTREE      |         |               | YES     |
| index_demo |          1 | composite_index_1 |            2 | name        | A         |           2 |     NULL |   NULL |      | BTREE      |         |               | YES     |
| index_demo |          1 | composite_index_1 |            3 | age         | A         |           2 |     NULL |   NULL | YES  | BTREE      |         |               | YES     |
| index_demo |          1 | composite_index_2 |            1 | pan_no      | A         |           2 |     NULL |   NULL | YES  | BTREE      |         |               | YES     |
| index_demo |          1 | composite_index_2 |            2 | name        | A         |           2 |     NULL |   NULL |      | BTREE      |         |               | YES     |
| index_demo |          1 | composite_index_2 |            3 | age         | A         |           2 |     NULL |   NULL | YES  | BTREE      |         |               | YES     |
| index_demo |          1 | composite_index_2 |            4 | phone_no    | A         |        NULL |     NULL |   NULL |      | BTREE      |         |               | YES     |
| index_demo |          1 | secondary_index_1 |            1 | name        | A         |           2 |        4 |   NULL |      | BTREE      |         |               | YES     |
| index_demo |          1 | secondary_index_1 |            2 | phone_no    | A         |        NULL |     NULL |   NULL |      | BTREE      |         |               | YES     |
+------------+------------+-------------------+--------------+-------------+-----------+-------------+----------+--------+------+------------+---------+---------------+---------+
17 rows in set (0.00 sec)
```

We defined `secondary_index_1` on `name`, but MySQL has created a composite index on (`name`, `phone_no`) where `phone_no` is the primary key column. We created `secondary_index_2` on `age` & MySQL created a composite index on (`age`, `phone_no`). We created `composite_index_2` on (`pan_no`, `name`, `age`) & MySQL has created a composite index on (`pan_no`, `name`, `age`, `phone_no`). The composite index `composite_index_1` already has `phone_no` as part of it.

So whatever index we create, MySQL in the background creates a backing composite index which in-turn points to the primary key. This means that the primary key is a first class citizen in the MySQL indexing world. It also proves that all the indexes are backed by a copy of the primary index —but I am not sure whether a single copy of the primary index is shared or different copies are used for different indexes.

There are many other indices as well like Spatial index and Full Text Search index offered by MySQL. I have not yet experimented with

General Indexing guidelines:

- Since indices consume extra memory, carefully decide how many & what type of index will suffice your need.
- With `DML` operations, indices are updated, so write operations are quite costly with indexes. The more indices you have, the greater the cost. Indexes are used to make read operations faster. So if you have a system that is write heavy but not read heavy, think hard about whether you need an index or not.
- Cardinality is important — cardinality means the number of distinct values in a column. If you create an index in a column that has low cardinality, that's not going to be beneficial since the index should reduce search space. Low cardinality does not significantly reduce search space.
  Example: if you create an index on a boolean (`int 1` or `0` only) type column, the index will be very skewed since cardinality is less (cardinality is 2 here). But if this boolean field can be combined with other columns to produce high cardinality, go for that index when necessary.
- Indices might need some maintenance as well if old data still remains in the index. They need to be deleted otherwise memory will be hogged, so try to have a monitoring plan for your indices.

In the end, it's extremely important to understand the different aspects of database indexing. It will help while doing low level system designing. Many real-life optimizations of our applications depend on knowledge of such intricate details. A carefully chosen index will surely help you boost up your application's performance.

article. :)

References:

1. https://dev.mysql.com/doc/refman/5.7/en/innodb-index-types.html
2. https://www.quora.com/What-is-difference-between-primary-index-and-secondary-index-exactly-And-whats-advantage-of-one-over-another
3. https://dev.mysql.com/doc/refman/8.0/en/create-index.html
4. https://www.oreilly.com/library/view/high-performance-mysql/0596003064/ch04.html
5. http://www.unofficialmysqlguide.com/covering-indexes.html
6. https://dev.mysql.com/doc/refman/8.0/en/multiple-column-indexes.html
7. https://dev.mysql.com/doc/refman/8.0/en/show-index.html
8. https://dev.mysql.com/doc/refman/8.0/en/create-index.html

Countinue reading about

Programming

550+ Free Online Programming & Computer Science Courses You Can Start This October

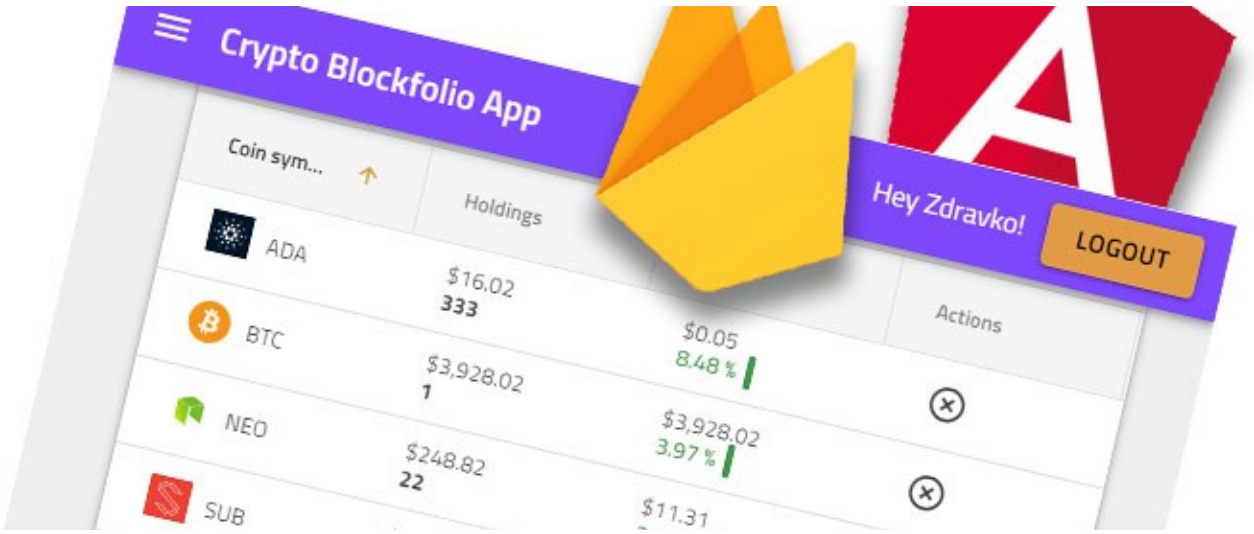
A Walkthrough of the FreeCodeCamp Telephone Validator Project

See all 2737 posts →

#MACHINE LEARNING

How I planned my meals with Reinforcement Learning on a budget

6 MONTHS AGO



#TECH

How to build a Firebase Angular app with auth and a real-time database

6 MONTHS AGO