Indexing in Database Management System

Introduction on Database Management System

Indexing is a technique to optimize our performance or processing speed of querying records in the database by minimizing the number of searches or scans required. It is a data structure technique that is used to quickly locate and access the data in a database. Users cannot see the indexes, they are just used to speed searches and queries.

Why Database Management System?

As mentioned before indexing is needed to optimize our processing time. We can experience it once we are handling millions of data. It is nothing but like our book indexes, it does not totally scan our entire rows, it just searches for an index range and finds the result within the index. No worries we can understand clearly once we continue this article.

Advantage of Indexing

CREATE INDEX statement in SQL is used to create indexes in tables. The indexes are used to retrieve data from the database more quickly than others. The user can not see the indexes, and they are just used to speed up queries /searches.

Disadvantage of Indexing

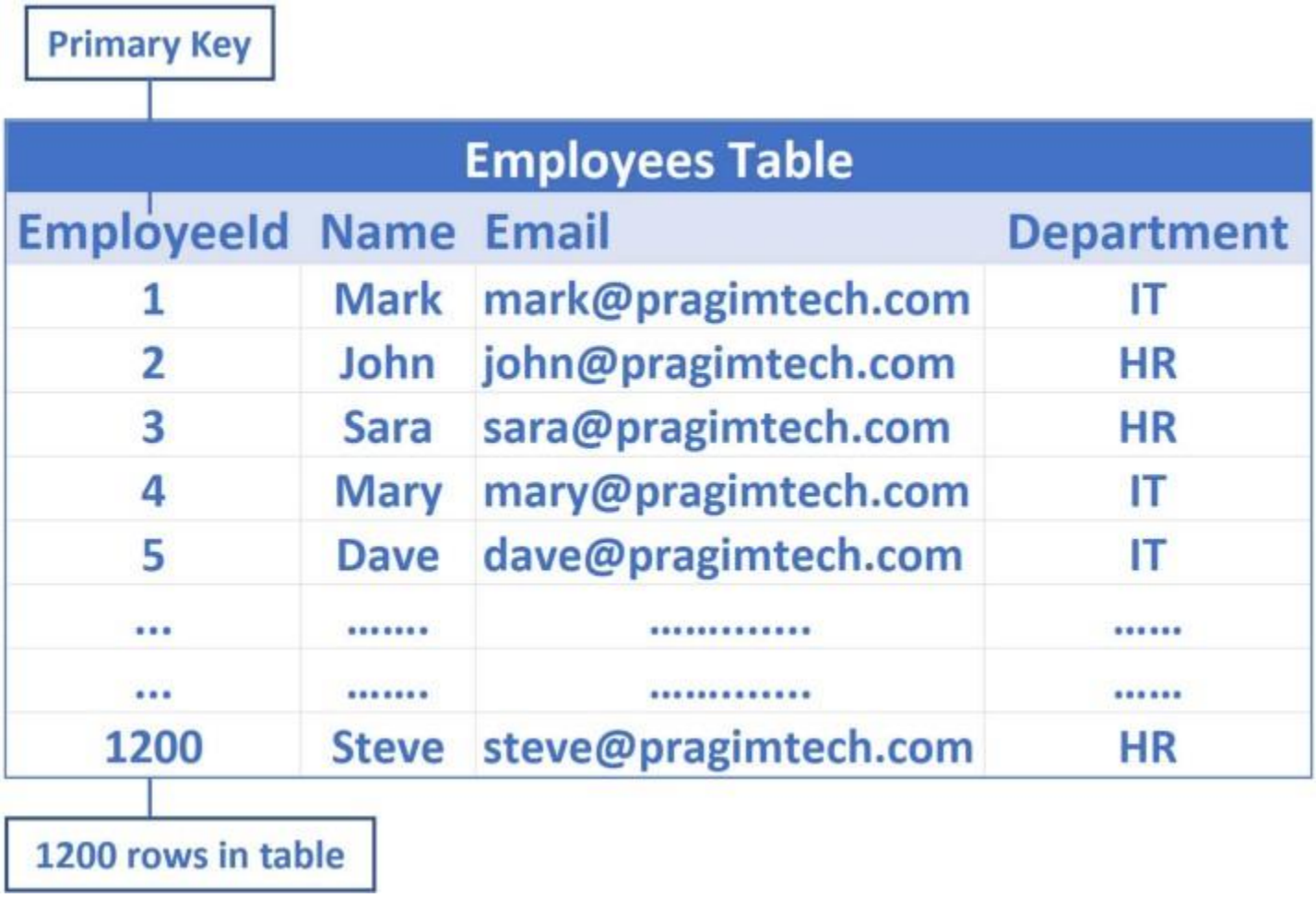
Updating the table with indexes takes a lot of time than updating a table without indexes. It is because the indexes also need an update. So, only create indexes on those columns that will be frequently searched items.

In this article, we will discuss how indexes actually work and help improve the performance of our SQL queries. We will discuss how both the index types work—Clustered and Non-clustered.

A clustered index is an index that stores the actual data & a non clustered index is just a pointer to a data. This data is available in its leaf nodes. A table can only have **one clustered index** and **up to 249 non clustered indexes**. If a table does not have a clustered index it is referred to as a heap.

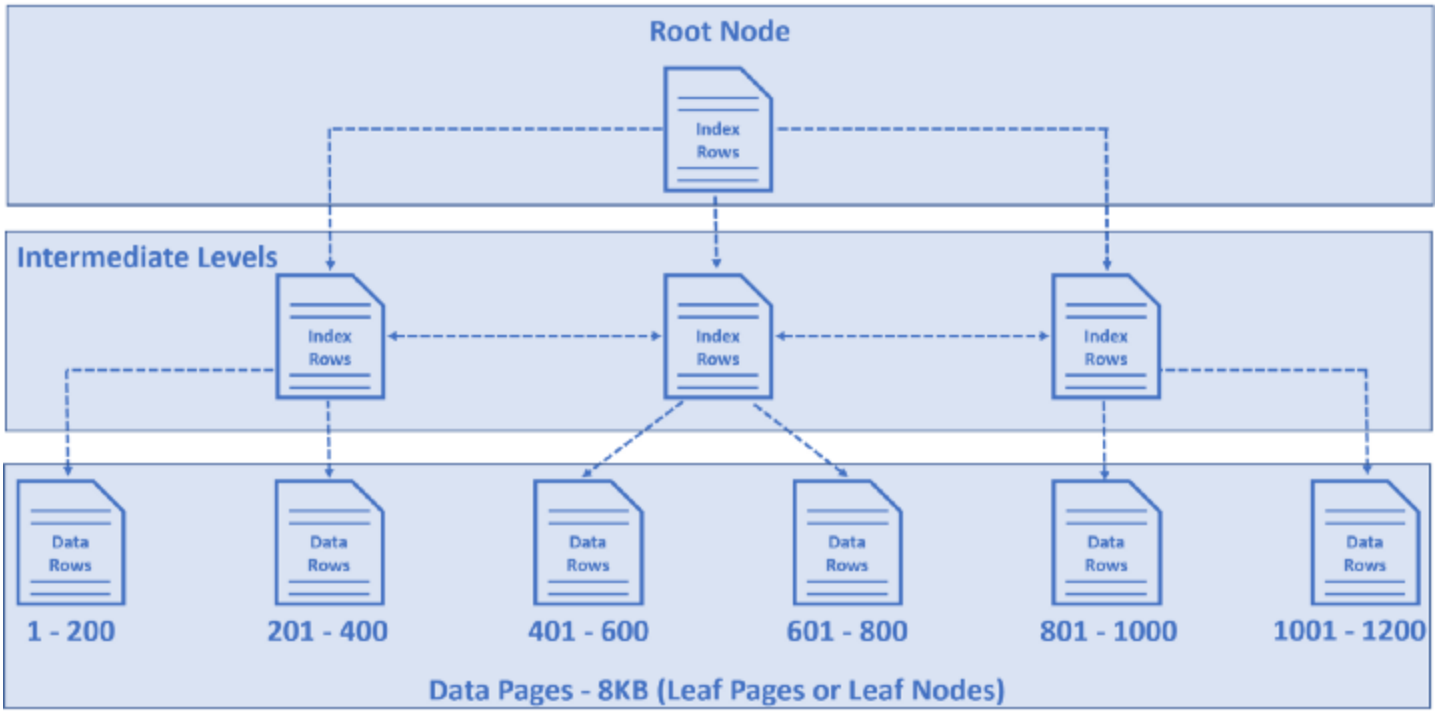
Clustered Index Structure

Lets look into the following Employee Table.



Here, EmployeeId is the primary key, as by default a **clustered index** on the EmployeeId column is created. This means employee data is sorted by the EmployeeId column and physically stored in a series of data pages in a tree-like structure that looks like the following.

Let's look at this Simple Architecture to get a better understanding.

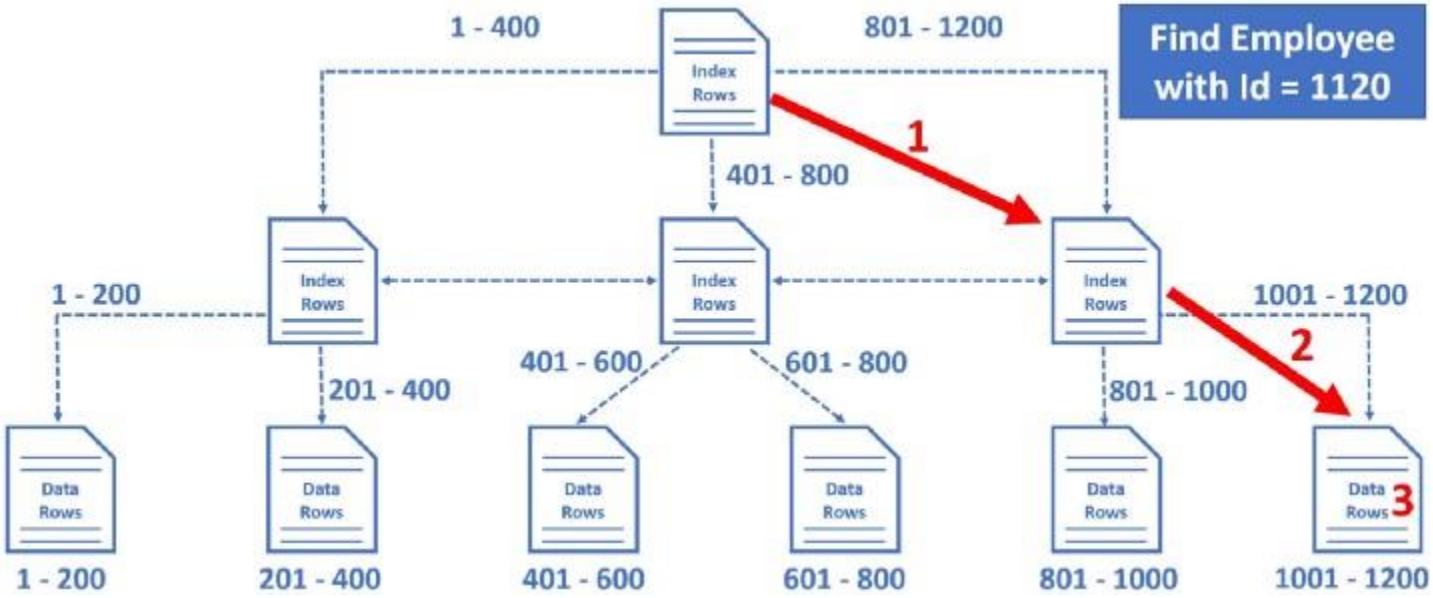


- The nodes at the bottom layer of the tree are called data pages or leaf nodes. This contains the actual data rows, in our example employee rows.
- By default a clustered index on this column is created by the primary key. In our case is EmployeeId. These employee rows are sorted by the EmployeeId column.
- For our example, let's say in the Employees table we have 1200 rows, and let's assume in each data page we have 200 rows.
- So, in the first data page, we have 1 to 200 rows, in the second 201 to 400, in the third 401 to 600, and so on and so forth.

- The node at the top of the tree is called the Root Node.
- The nodes between the root node and the leaf nodes are called intermediate levels.
- The root and the intermediate level nodes contain index rows.
- Each index row contains a key value (in our case Employee Id) and a pointer to either an intermediate level page in the B-tree, or a data row in the leaf node.
- So this tree-like structure has a series of pointers that helps the query engine find data quickly.

How SQL Server finds a row by ID

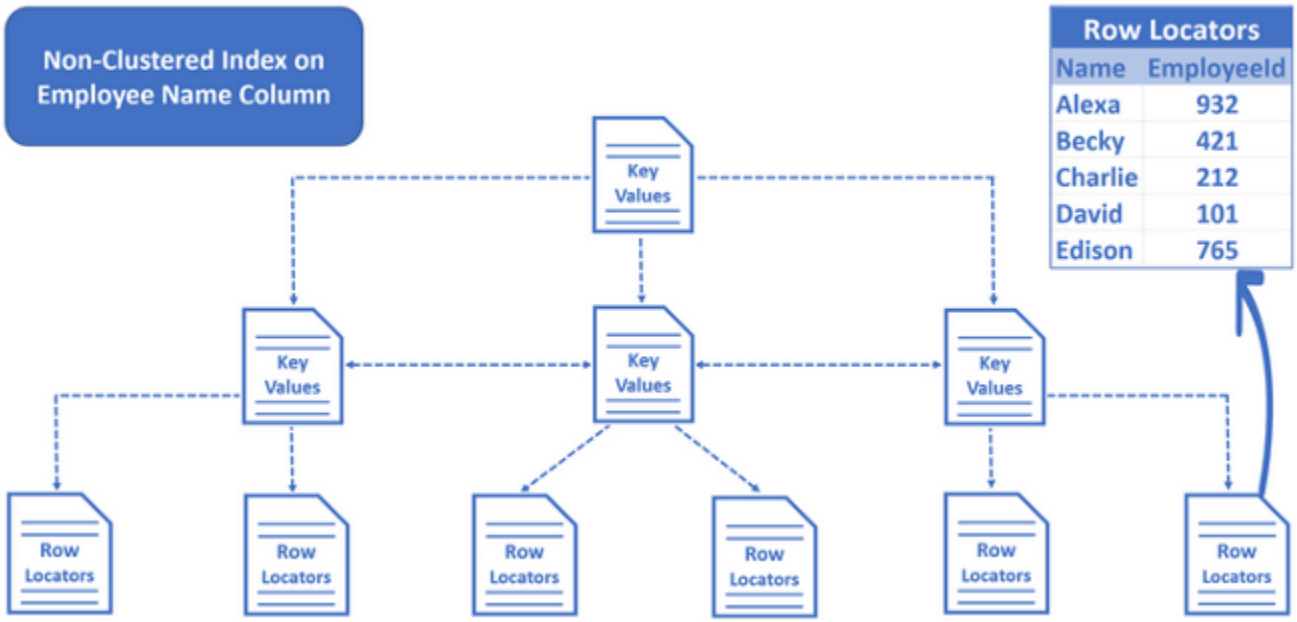
For example, let's say we want to find Employee row with EmployeeId = 1120



1. So the database engine starts at the root node and it picks the index node. In our case, the root node is on the right, because the database engine knows it is this node that contains employee IDs from 801 to 1200.
2. From there, it picks the leaf node that is present on the extreme right, because employee data rows from 1001 to 1200 are present in this leaf node.
3. The data rows in the leaf node are sorted by Employee ID, so it's easy for the database engine to find the employee row with Id = 1120.

Notice in just 3 operations, SQL Server is able to find the data we are looking for. It's making use of the clustered index we have on the table.

Non-Clustered Index in MySQL Server



- In a non-clustered index we do not have table data. We have key values and row locators.
- We created a non-clustered index on the Name column, so the key values, in this case, Employee names are sorted and stored in alphabetical order.
- The row locators at the bottom of the tree contain Employee Names and the cluster key of the row. In our example, Employee Id is the cluster key.

Practical 1

This practice is getting practice about creating the single, combined, show, drop the index.

1. **Create an index**

```
create index index_name on Employees(name);
```

2. **Describe Tables**

```
desc Employees;
```

3. **Show Index**

```
show index from Employees;
```

4. **Create an index combining multiple columns**

```
create index index_name_department on Employees(name, department);
```

5. **Drop index**

```
alter table Employees drop index index_name_department;
```

Practical 2

This practice is based on the game play history of game play users. It has 4701404 records. So here when we query data for some requirements there will be considerable execution time. Application wise the milliseconds we need to be accountable. Our ultimate plan is to create an index for the game_history table by using the score column.

1. Get all results in the game_history table, the primary key is game_history_id. It returns 4701404 records.

game_history_id	score	time	game_id	leaderboard_id	user_id
4701672	0	2022-05-29 09:38:57	204	174	183733
4701673	98	2022-05-29 09:40:26	204	174	6861
4701674	0	2022-05-29 09:40:34	198	169	183733
4701675	116	2022-05-29 09:43:44	204	174	6861
4701676	0	2022-05-29 09:46:00	198	169	183733
4701677	10	2022-05-29 09:47:13	198	169	183733
4701678	109	2022-05-29 09:48:13	204	174	6861
4701679	19	2022-05-29 09:48:45	198	169	183733
4701680	1	2022-05-29 09:49:07	198	169	183733
4701681	13	2022-05-29 09:50:42	198	169	183733

game_history Table

Returns 4701404 records

2. Now let's show the index that we have a default. You can see the default indexes. Here only the primary key is known as a clustered index, the rest are called non clustered index.

```
use imi_local;
show index from game_history;
```

Table	Non_unique	Key_name	Seq_in_index	Column_name	Collation	Cardinality	Sub_part	Packed	Null	Index_type
game_history	0	PRIMARY	1	game_history_id	A	4675633	NULL	NULL		BTREE
game_history	1	GHISTORY_USER_INDEX	1	user_id	A	83484	NULL	NULL	YES	BTREE
game_history	1	GHISTORY_LEADERBOARD_INDEX	1	leaderboard_id	A	198	NULL	NULL	YES	BTREE
game_history	1	GHISTORY_GAME_USER_LEADERBOARD_INDEX	1	game_id	A	9220	NULL	NULL	YES	BTREE
game_history	1	GHISTORY_GAME_USER_LEADERBOARD_INDEX	2	user_id	A	382247	NULL	NULL	YES	BTREE
game_history	1	GHISTORY_GAME_USER_LEADERBOARD_INDEX	3	leaderboard_id	A	382247	NULL	NULL	YES	BTREE

3. Before getting into adding an index let's have a play with the default index that game_history_id.

Clustered Index Seek

```
SELECT * FROM imi_local.game_history where game_history_id=44227;
```

game_history_id	score	time	game_id	leaderboard_id	user_id	x_platform
44227	18	2018-11-05 16:14:58	2	2	2801	NULL

Output

#	Time	Action	Message	Duration / Fetch
1	16:00:36	SELECT * FROM imi_local.game_history where game_history_id=44227	1 row(s) returned	0.000 sec / 0.000 sec

Notice, that the operation is **Clustered Index Seek**, meaning the database engine is using the clustered index on the game_history_id column to find the game_history row with game_history_id = 44227

- Number of rows read = 1
- Actual number of rows for all executions = 1

The number of rows reads is the number of rows the SQL server has to read to produce the query result. In our case game_history_id is unique, so we expect 1 row and that is represented by an Actual number of rows for all executions.

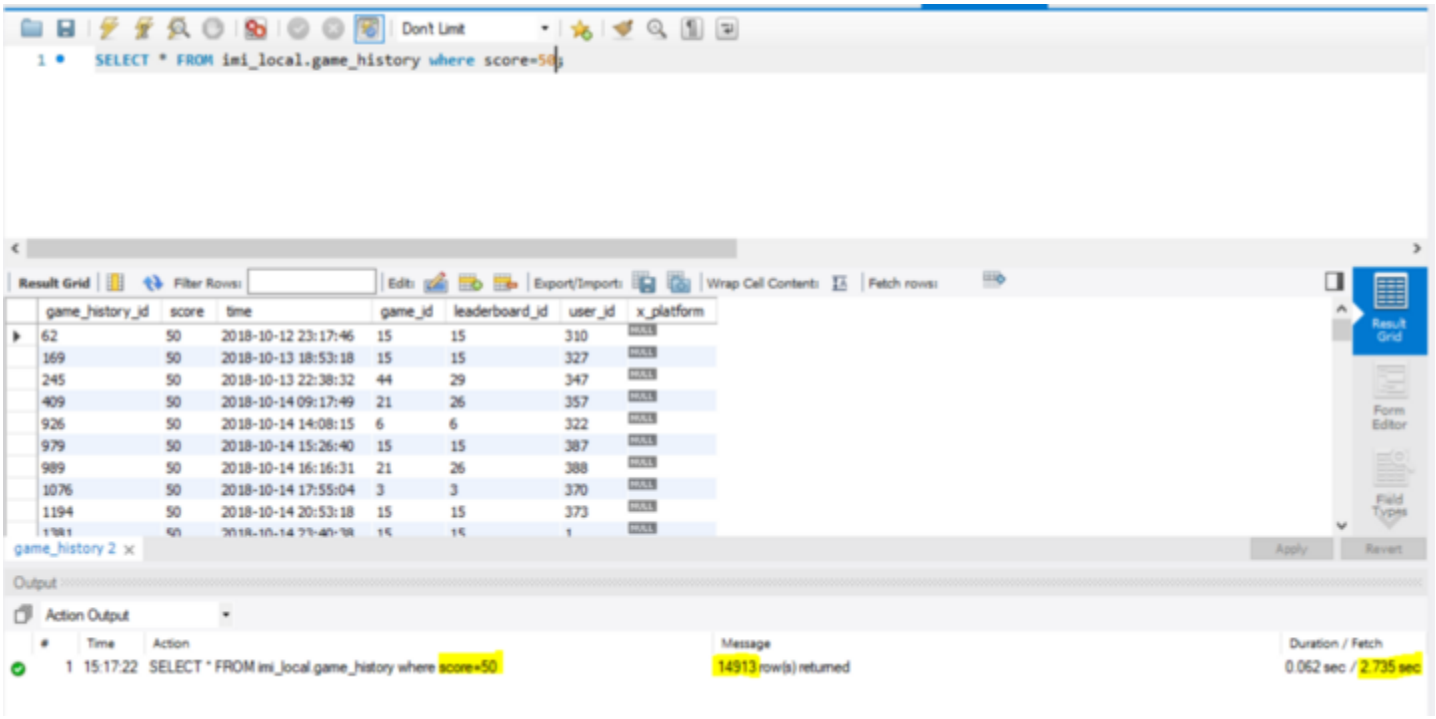
With the help of the default index, the SQL server is able to directly read that 1 specific game_history row we want. Hence, both, Number of rows read and the Actual number of rows for all executions is 1.

So the point is, if there are thousands or even millions of records, the SQL server can easily and quickly find the data we are looking for, provided there is an index that can help the query find data.

Clustered Index Scan

In this example, there is a clustered index on the game_history_id column, so when we search by game_history_id, SQL Server can easily and quickly find the data we are looking for. What if we search by score? At the moment, there is no index on the score column, so there is no easy way for the SQL server to find the data we are looking for. SQL Server has to read every record in the table which is extremely inefficient from a performance standpoint.

1. We know we don’t have any indexes for retrieving scores from the table. so at first to see the execution time, we get the results of a score that equals 50.



Notice, that the operation is **Clustered Index scan**. Since there is no proper index to help this query, the database engine has no other choice than to read every record in the table. This is exactly the reason why the number of rows reads is 4701404, i.e every row in the table.

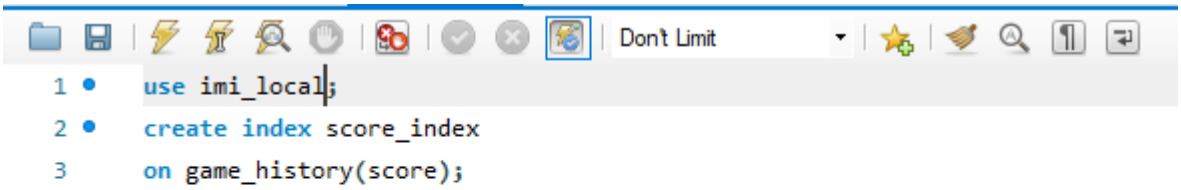
- Number of rows read = 4701404
- Actual number of rows for all executions = 14913

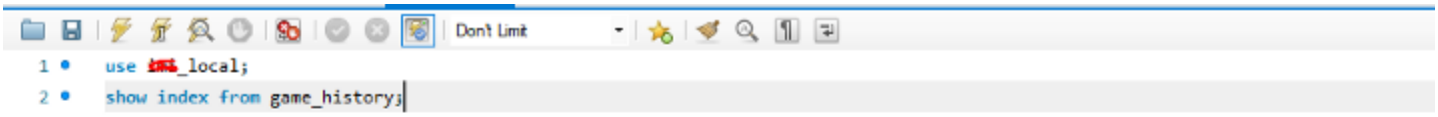
Here we can see we are getting 14913 results, the execution time is 2.735 seconds without a score index.

How many rows are we expecting in the result? Well, only 14913 rows because there are some game_history which score=50. So, to produce these 14913 rows as the result, the SQL Server has to read all the 4701404 rows from the table because there is no index to help this query. This is called Index Scan and in general, Index Scans are bad for performance.

3. Now let’s add the index for the score. If we add the index here, it will consider as non clustered index.

Note: You don’t need to specify that an index is **NON-CLUSTERED** in MySQL, it’s implicit from the table design. Only the PRIMARY KEY or the first non-NULL UNIQUE KEY can be the clustered index, and they will be the clustered index without your choosing. All the other indexes in the table are implicitly non-clustered.





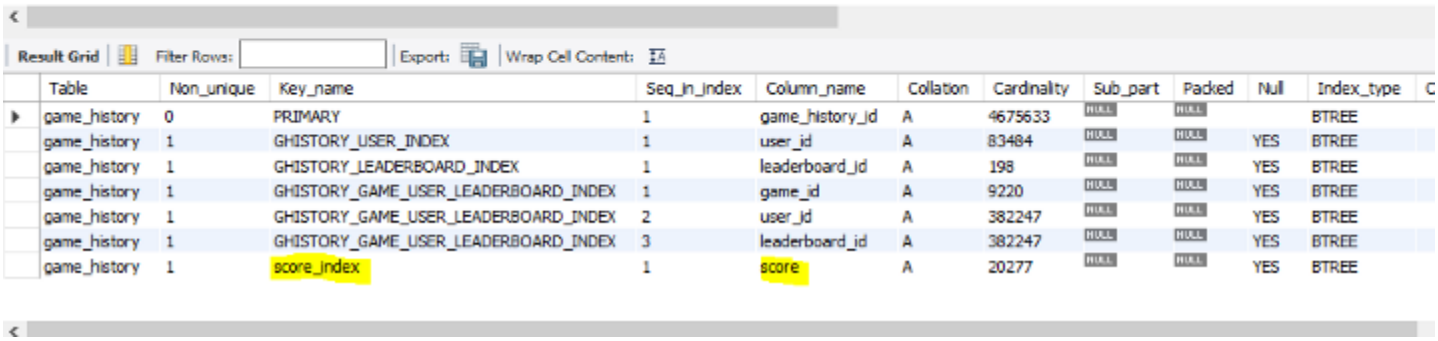
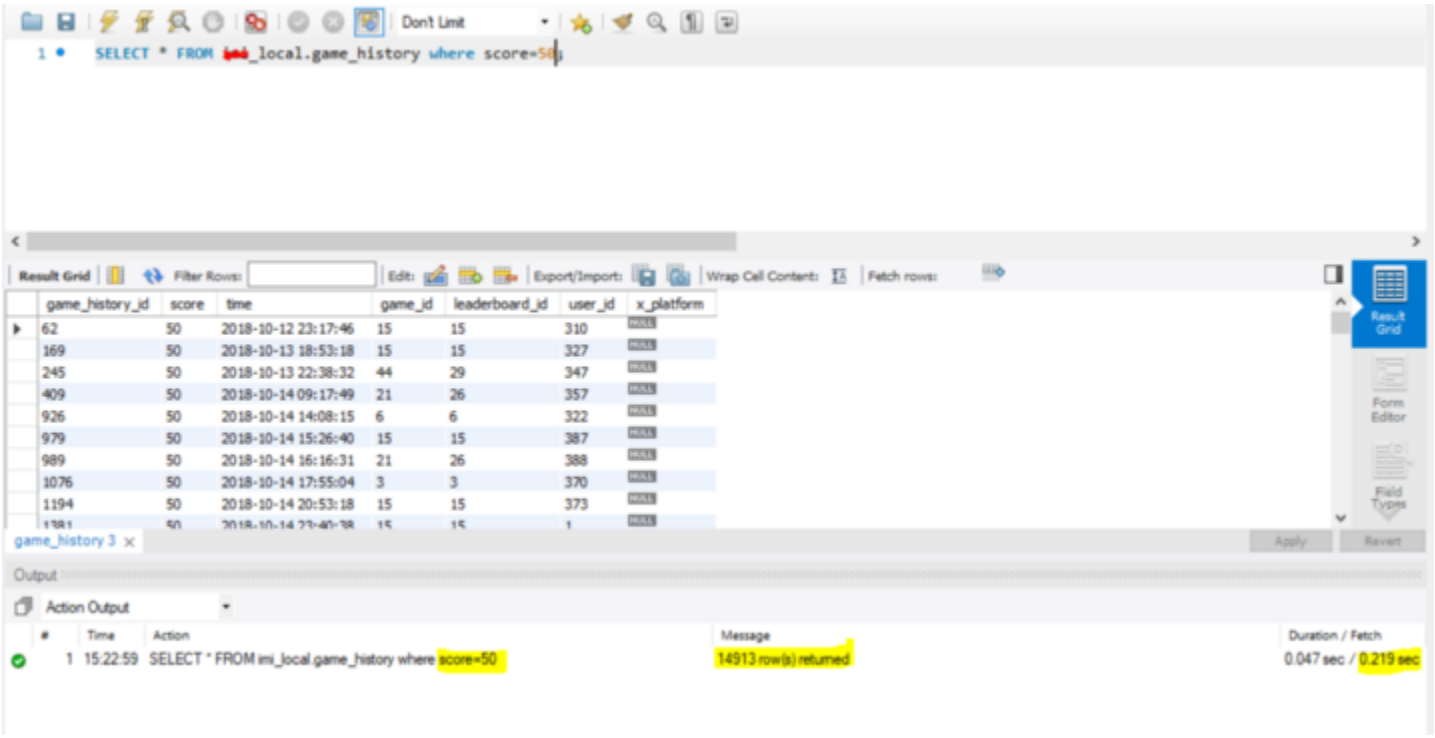

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

Table	Non_unique	Key_name	Seq_in_index	Column_name	Collation	Cardinality	Sub_part	Packed	Null	Index_type
game_history	0	PRIMARY	1	game_history_id	A	4675633	NULL	NULL		BTREE
game_history	1	GHISTORY_USER_INDEX	1	user_id	A	83484	NULL	NULL	YES	BTREE
game_history	1	GHISTORY_LEADERBOARD_INDEX	1	leaderboard_id	A	198	NULL	NULL	YES	BTREE
game_history	1	GHISTORY_GAME_USER_LEADERBOARD_INDEX	1	game_id	A	9220	NULL	NULL	YES	BTREE
game_history	1	GHISTORY_GAME_USER_LEADERBOARD_INDEX	2	user_id	A	382247	NULL	NULL	YES	BTREE
game_history	1	GHISTORY_GAME_USER_LEADERBOARD_INDEX	3	leaderboard_id	A	382247	NULL	NULL	YES	BTREE
game_history	1	score_index	1	score	A	20277	NULL	NULL	YES	BTREE

4. After adding the index, execute the same query to get the results of a score that equals 50.



Without index: Execution time is 2.735

With index: Execution time is 0.219

Conclusion on Database Management System

Finally, what we learned so far is, Indexing the Database Management System is the best technique for querying the largest set of data by reducing the execution time multiple times. Ideally, this article provides much conceptual and practical knowledge about indexing. I hope this article explains the indexing method using clustered and non-clustered techniques. Also keep in mind when you are going to index the MySQL Database, If the table has the primary key that will be considered the clustered indexing, All the other indexing is known as non clustered indexing. We will see it in another article.