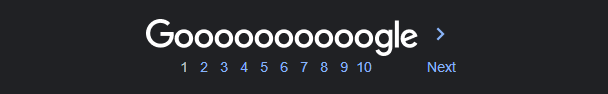
In mysql, limit 1000,10 and limt 10 have the same performance? How do we do pagination?

We often see pagination like this at the bottom of page when search some key words by google, :



Some guys think it is easy to implement it by using *limit* in mysql.

Assume we have a table like this:

*CREATE TABLE IF NOT EXISTS `EXMAPLE` (*

*`ID` BIGINT NOT NULL AUTO\_INCREMENT ,*

*`NAME` VARCHAR(128) NOT NULL,*

*`CONTENT` VARCHAR(128) NOT NULL,*

*PRIMARY KEY (`ID`),*

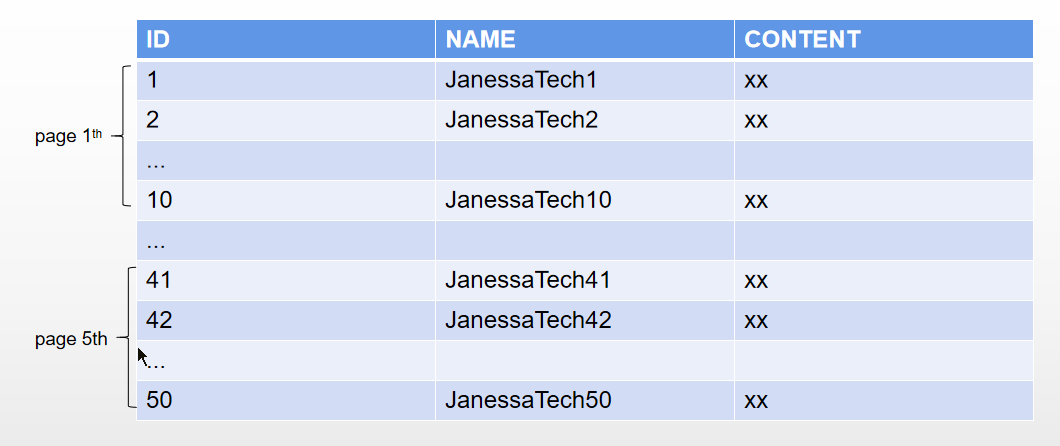
*INDEX `IDX\_NAME` (`NAME`)*

*)*

*ENGINE = InnoDB DEFAULT CHARSET=utf8;*

To paginate *`EXMAPLE`,* we will use the sql statement like this:

SELECT \* FROM *`*EXMAPLE` ORDER BY ID LIMIT {offset}, {size};



For the first page, the sql statement will be:

**SELECT \* FROM *`*EXMAPLE` ORDER BY ID LIMIT 0, 10;**

For the fifth page, the sql statement will be:

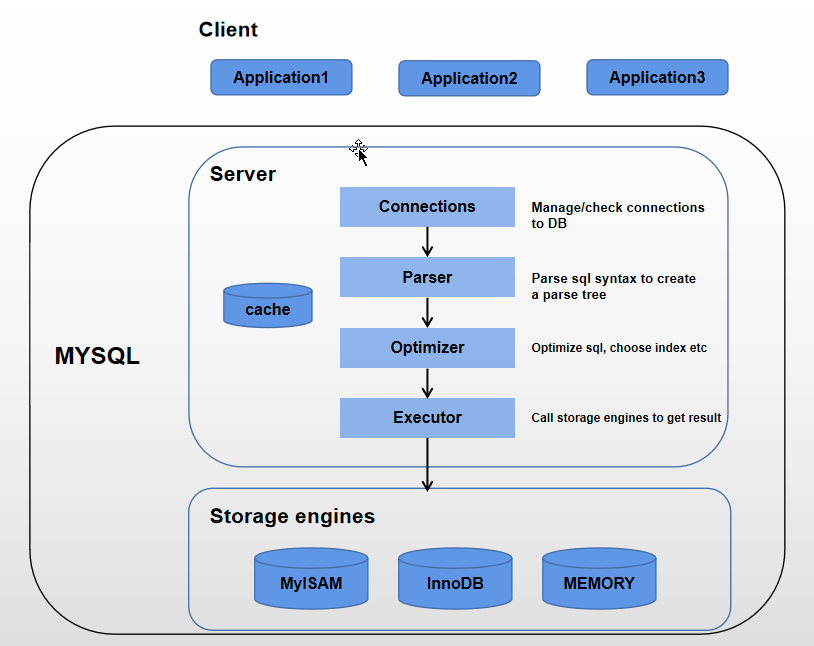
**SELECT \* FROM *`*EXMAPLE` ORDER BY ID LIMIT 40, 10;**

The question is:

Is the performance of the two sql statements the same considering they all fetch 10 items? If not, why?

To answer this question, let’s take a look at how ‘limit’ works internally in mysql:

MySQL consists two parts: **Server** and **storage engines**.

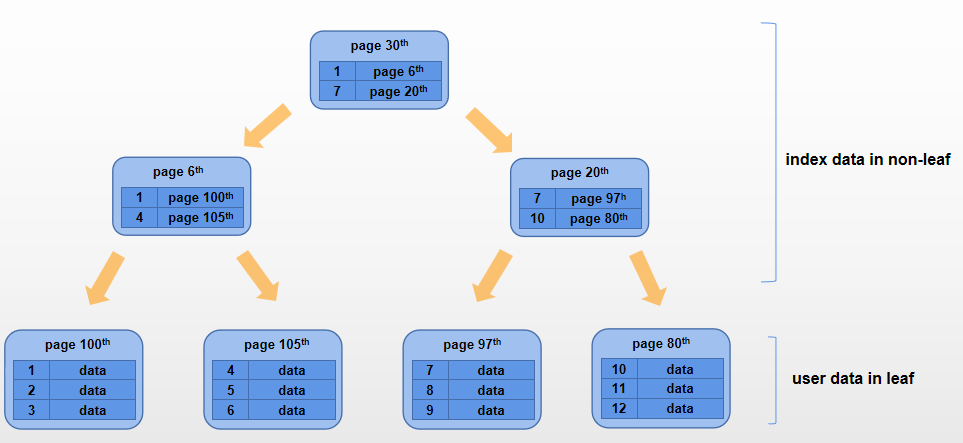


Server consists many modules. Let’s focus on **Executor** which will interact with storage engines. **Executor** fetches data by calling API interfaces provided by **storage engines** and then puts result into cache. Finally, the result is returned to Client(An application written by Java, go etc).

Next, let’s take a look at another important concept: B+ tree

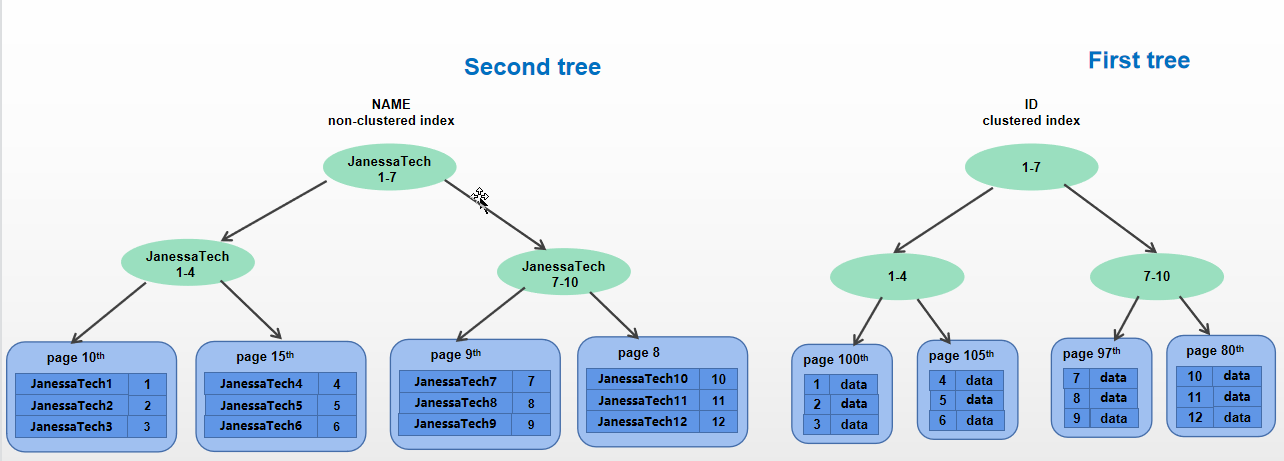
Let’s take a quick look-back if you guys are not familiar with B+ tree.

The B+ tree would look like below:



Mysql will create several B+ trees for one table (assume with primary key and non-primary key)

Take the table `**EXMAPLE**` in this article as an example, mysql will construct 2 B+ trees. First B+ tree is constructed using primary key “**ID**”(often referred to as Clustered index) as index, second B+ tree is constructed using “**IDX\_NAME**”(often referred to as Non-clustered index) as index. The importance to notice is that: For the first tree, all pages at the bottom of the tree contain complete row data in each entry. For the second tree, all pages at the bottom of the tree contain primary key only in each entry

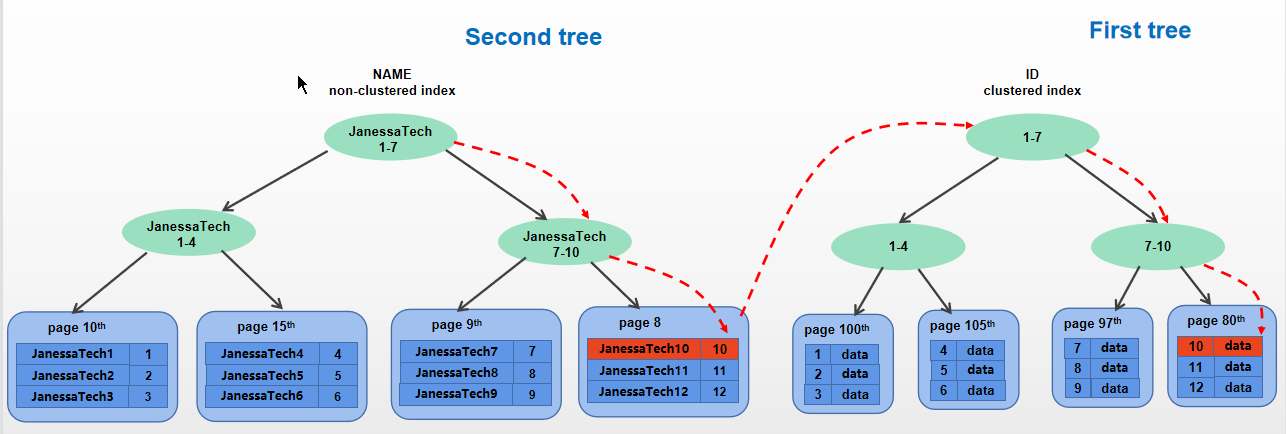


mysql will search these B+ trees to get result when a select statement is executed,

Which B+ tree will be searched through depends on how the select statement filters data.

When executing **SELECT \* FROM *`*EXMAPLE` WHERE ID=10**, mysql will search the first tree only from the root all the way down to the leaf page 105th which contains the item with ID being 10.

When executing **SELECT \* FROM EXMAPLE WHERE NAME=’JanessaTech10’,** mysql will search the second tree first, from the root all the way down to the leaf page 9th, and finds that the primary key of the item containing **’JanessaTech10’** in page 9th is 10, then it goes to the root of first tree, traverses until ends up at page 105th, returns the row data of the key being 10.



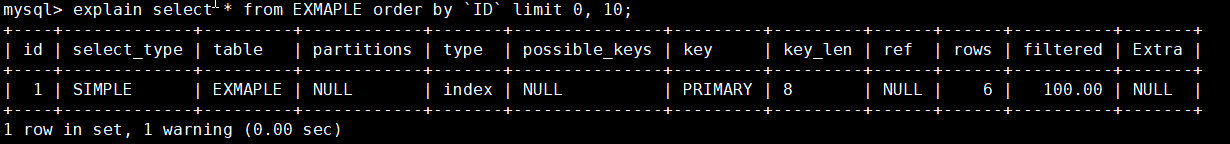
Two sql statements we analyzed look for the same item, obviously, the performance of **SELECT \* FROM *`*EXMAPLE` WHERE ID=10** is much better than **SELECT \* FROM EXMAPLE WHERE NAME=’JanessaTech10’,** because mysqltraversed two B+ trees for the second sql, whereas only one traversal for the first sql.

Back to the sqls of our original question: which B+ tree mentioned will go through or both?

**SELECT \* FROM *`*EXMAPLE` ORDER BY ID LIMIT 0, 10;**

**SELECT \* FROM *`*EXMAPLE` ORDER BY ID LIMIT 40, 10;**

It is easy to know that by running **explain SELECT \* FROM *`*EXMAPLE` ORDER BY ID LIMIT 0, 10;**



The value of key is **PRIMARY**, which means mysql only traverses the B+ tree constructed using primary key as index(the first B+ tree).

OK. So far, we know that for the sqls below, mysql only goes through first B+ tree,

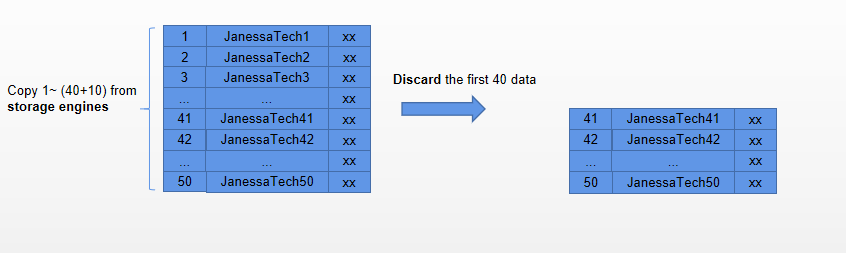
**SELECT \* FROM *`*EXMAPLE` ORDER BY ID LIMIT 0, 10;**

**SELECT \* FROM *`*EXMAPLE` ORDER BY ID LIMIT 40, 10;**

We notice that two sqls require all columns by using \* in select.

When the first sql is executed, internally, **Server** will call **storage engines**(innodb for example), then storage engines will fetch 10 full rows by going through the first B+ tree using the primary key 0 to 10, then give the result to Server. Server will return the result directly to client.

When the second sql is executed, internally, storage engines will fetch 50 full row from **1 to 40 + 10**, then give the result to **Server. Server will discard the first 40 full rows, leaving only the last 10 rows, then return the final result to client.**



We notice that the data **storage engines** gives to **Server** contains too much useless data which will be discarded by **Server** when offset is not 0. Too much time is spent on fetching the useless data. The larger the offset is, the more time spent, especial when the offset is a large number saying 1000000.

Now we know the answer of our original question: **SELECT \* FROM *`*EXMAPLE` ORDER BY ID LIMIT 40, 10** is better than **SELECT \* FROM *`*EXMAPLE` ORDER BY ID LIMIT 0, 10**, because storage engines spent some time on fetching some useless data which will be discarded by **Server**. **SELECT \* FROM *`*EXMAPLE` ORDER BY ID LIMIT 0, 10** has no such an issue.

Is there any way to optimize **SELECT \* FROM *`*EXMAPLE` ORDER BY ID LIMIT 40, 10**?

We know that Serverwhen offset is not 0 and we execute sql starting with “select \*”,

Let’s take a look at the first sql statement:

**SELECT \* FROM *`*EXMAPLE` ORDER BY ID LIMIT 0, 10;** (Let’s call it **SQL1**)

Let’s use **explain** to analyze **SQL1**

We see that the value of key column is **PRIMARY**. **PRIMARY indicates mysql uses** **primary key as index** to fetch data when executing **SQL1**

mysql will create an B+ tree using **primary key as index** when the target sql statement uses primary key as filter. For example SQL1, it uses **ID as filter, mysql will create an B+ tree using ID as index .**

mysql will create an B+ tree using **non-primary key as index** when the target sql statement uses **non-primary key** as filter. For example, **select \* from EXMAPLE where NAME=’JanessaTech1’.**(You could try to run **explain select \* from EXMAPLE where NAME=’JanessaTech1’** to see the value of key)

In the B+ tree above, the content in all leaves depends on

- In case the value of key column is **PRIMARY**: For example when executing **select \* from EXMAPLE where ID=1,** the complete row data will be put into each leaf

- In case the value of key column is **non PRIMARY**: only primary key will be put into each leaf. In this case, mysql needs to search two B+ trees. For the first time, mysql searchesWe call the process of going through the B+ tree constructed using primary key as index as