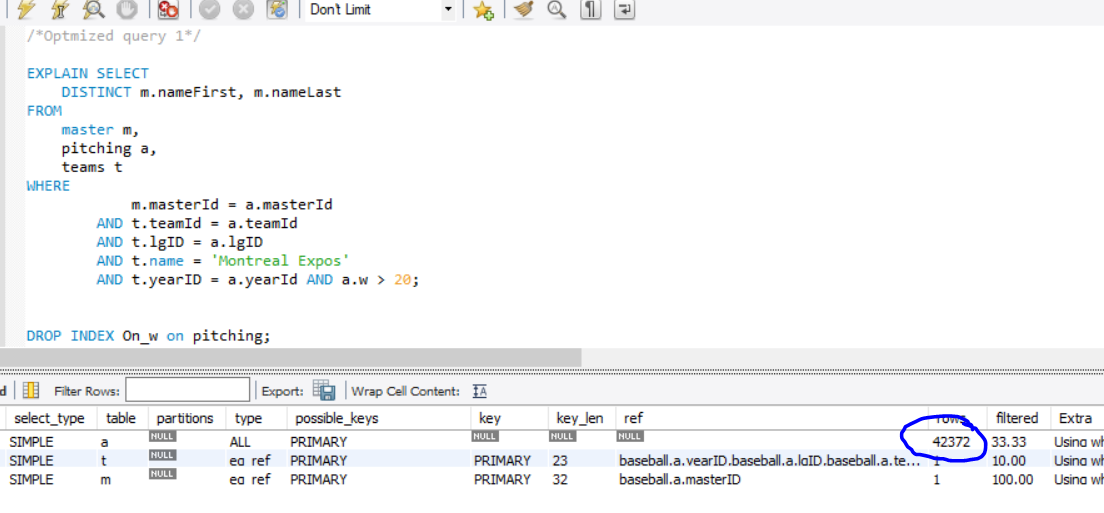
**Query 1:**

I executed the given query and used the explain keyword to check the number of rows that were being read. There were 42372 rows that were being read.



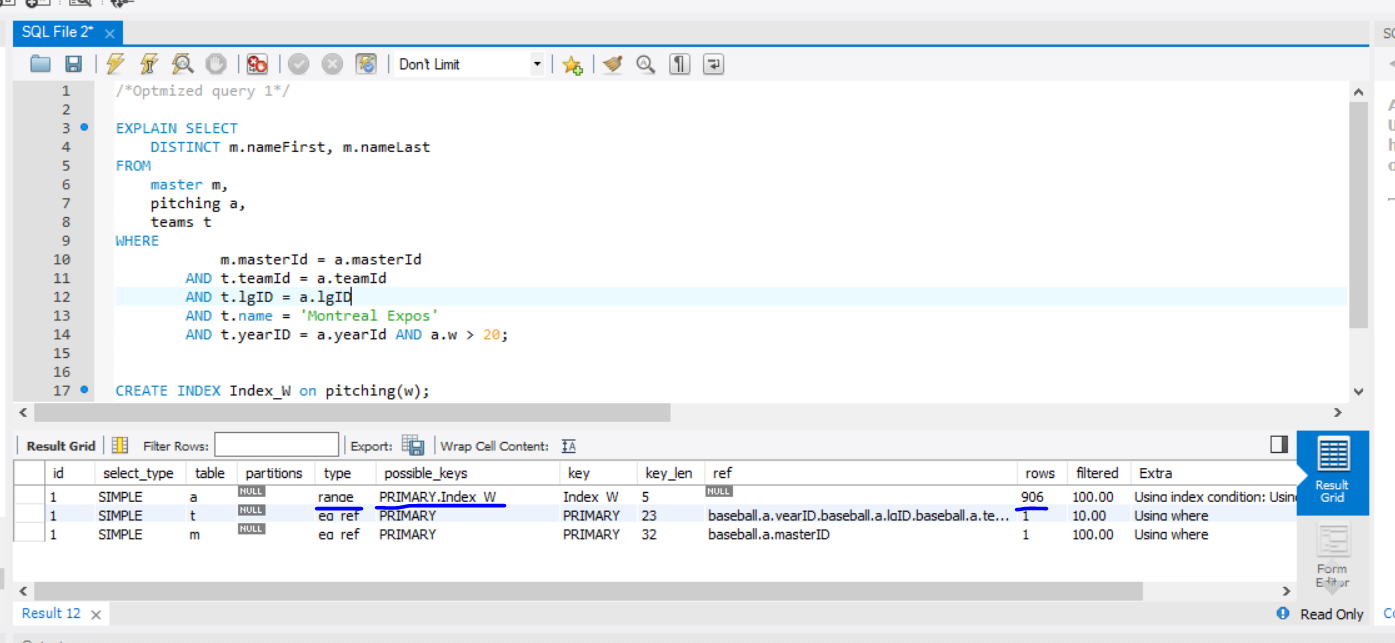
Firstly, I removed the like keyword and replaced it with =, which improves the efficiency of finding matching rows by a very slight margin.

I assessed the conditions under the where clause and I see that we’re trying to filter out rows using the a.w column. We can apply indexing on the “w” column of the pitching table, which can help us get lesser number of rows from the pitching table, which are then used to join with master and teams table.

I create an index using

CREATE INDEX Index\_W on pitching(w);

After creating this index the number of rows we get from the pitching table decreases to 906 rows which means we have to read through almost 50 times lesser number of rows.



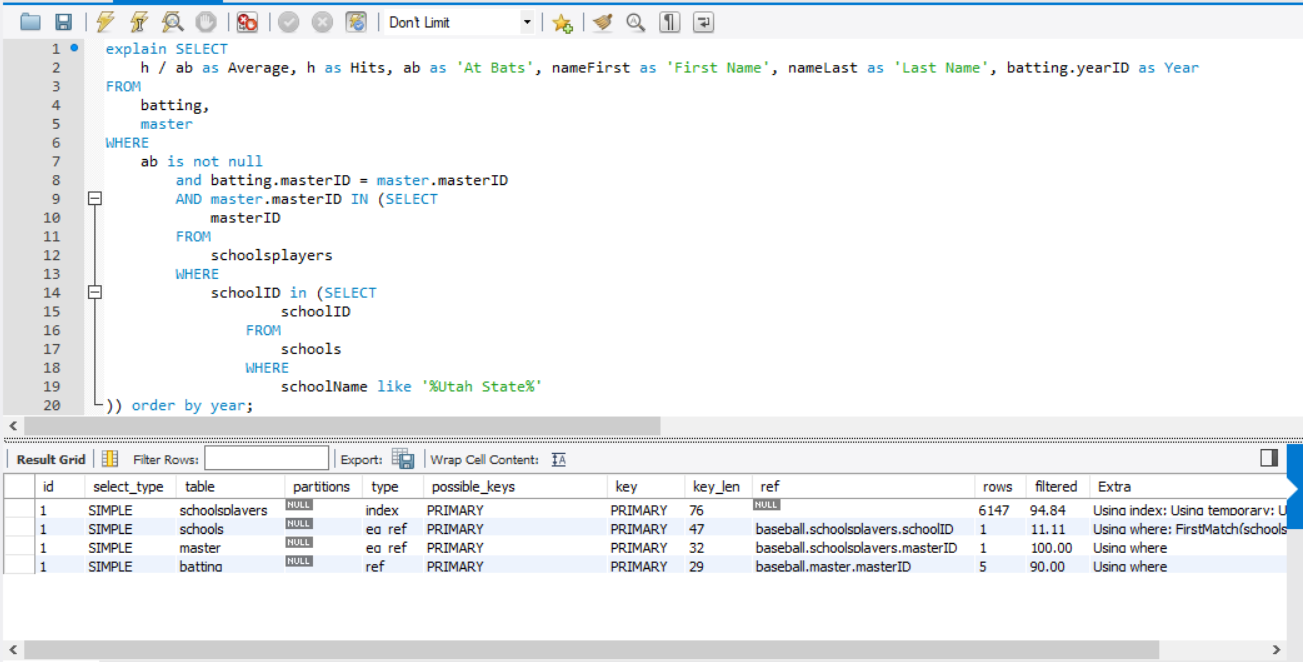
**Query 2**

I executed the below query using the explain function and it can be seen that schoolID from schoolsPlayers is what is being used to retrieve data for a school player.

Now, here if we assess the query, there can be multiple number of schoolIDs for a player belonging to Utah state university. In this case, once we retrieve the schoolID of Utah State University, 6147 rows are scanned to find out all the players that belong to Utah state university.

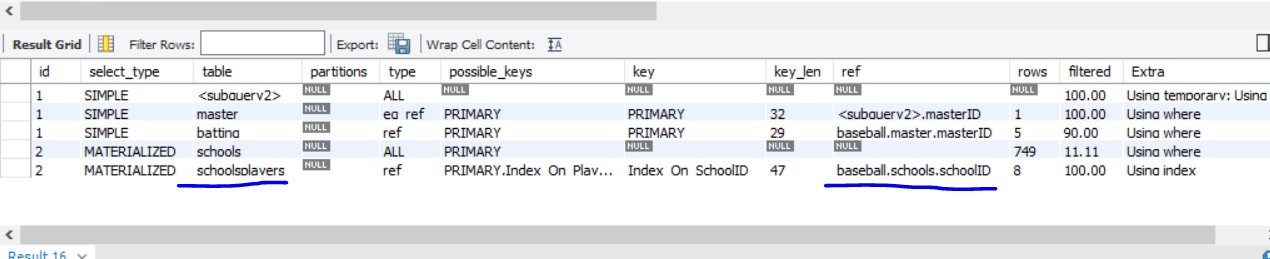
If we think about how we can optimize this, we could create an index on schoolID and schoolPlayers ID.

If we do that, then we would have much lesser rows to go through to find out schoolPlayers ID belonging to Utah State.



First I create an index on schoolID and test if there is an improvement in the number of rows we get.

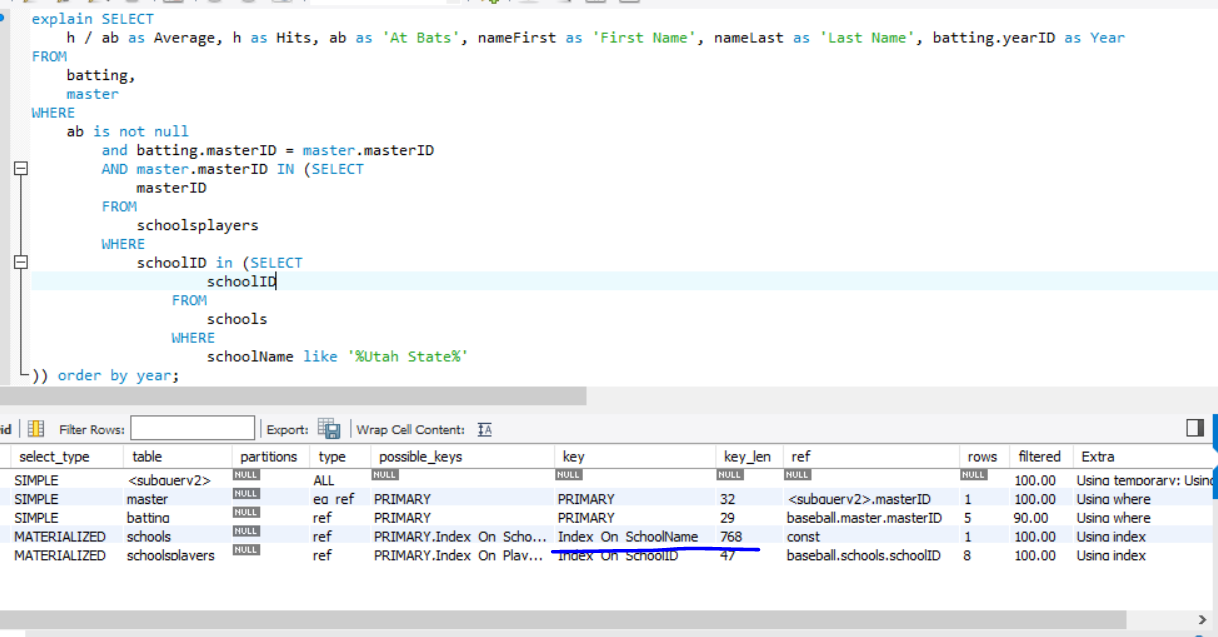
CREATE INDEX Index\_On\_SchoolID on schoolsPlayers(schoolID);



The rows count has now been decreased to 8 because we indexed the school players by their indexes, which makes it easier to find school players now.

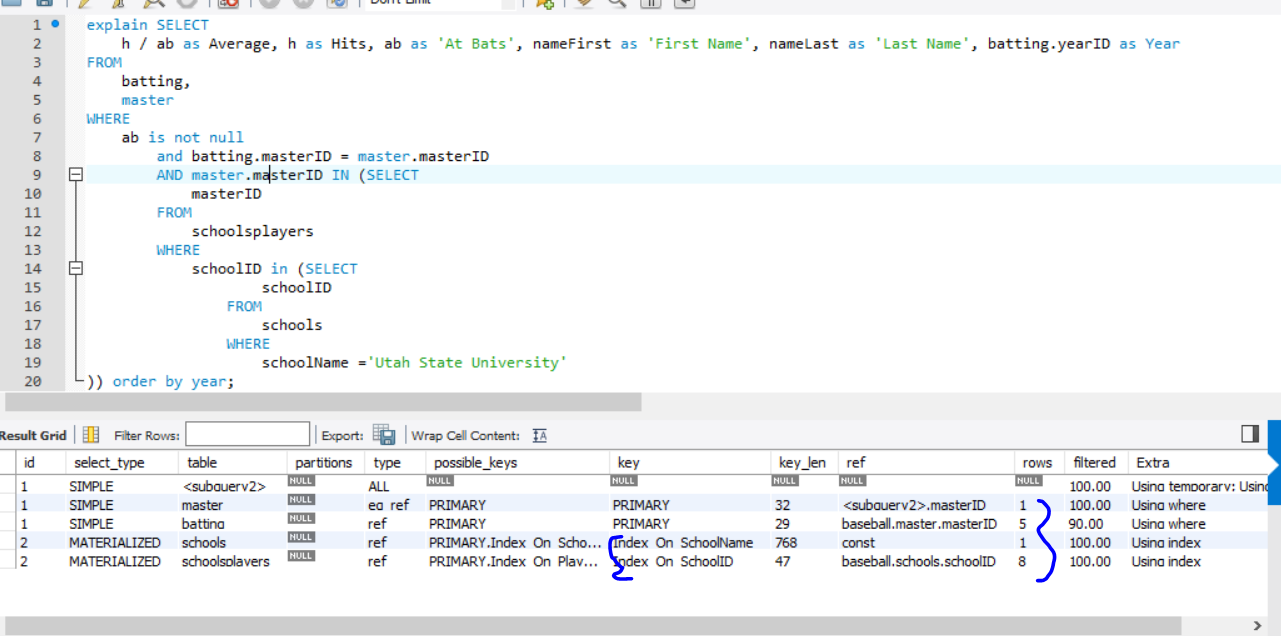
We could also apply another index to the schools table on schoolName column to reduce the current rows count 749.

CREATE INDEX Index\_On\_SchoolName on schools(schoolName);



If you notice, it is still the same. That is because we have used schoolName like ‘%Utah State%’

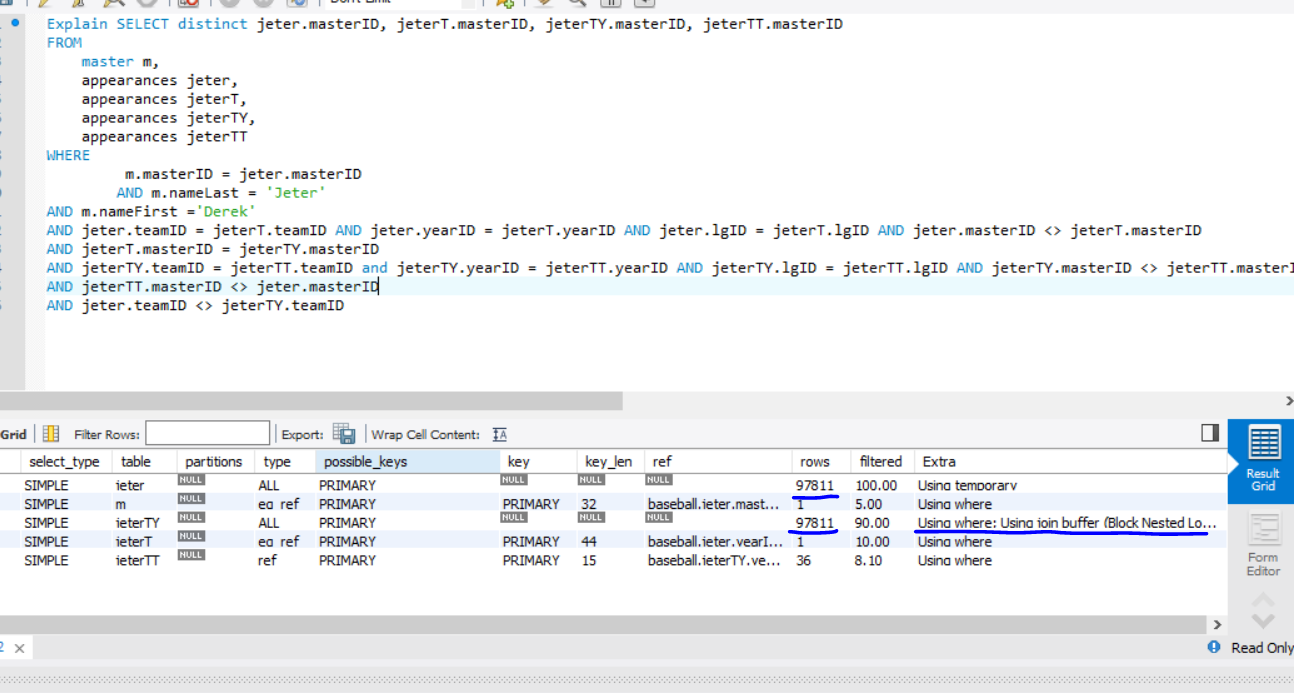
If we know the name of the university we can improve the performance of the query by replacing it by schoolName=’Utah State University’ and below will be the result-



**Query 3**

The below query is quite slow uses block nested loop to run. The appearances table is being read in a nested way for the tables jeter and jeterTY and both tables are being read for around 97k rows,

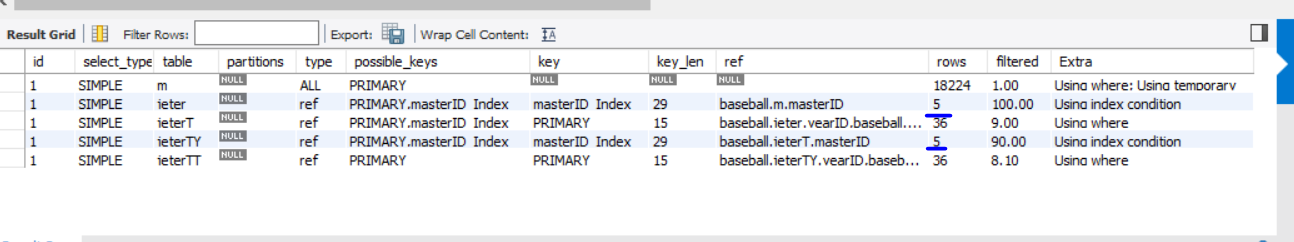
97000\*97000 amounts to a huge amount of rows and that is why this query ends up taking about 6 seconds to execute.



We can create an index and the masterID column in the appearances table as it is being heavily used to join the jeter and jeterTY table and improve the cost.

CREATE INDEX masterID\_Index ON appearances(masterID);

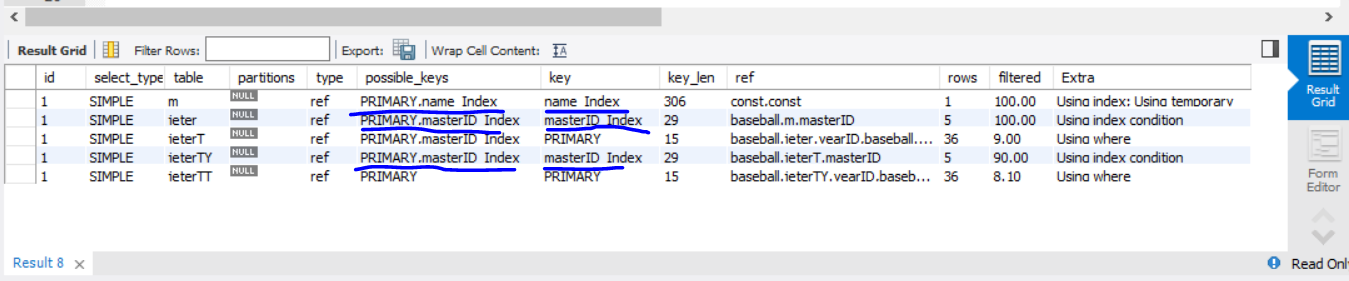
The cost improvement after doing this is pretty good-



Now that we have improved the cost for reading rows from the appearances table. We can notice that the master table(m) has quite a lot of cost we can improve on.

We can apply another index on the nameFirst and nameLast column as it is being used in the where clause to find the player named Jason Derek in query.

CREATE INDEX name\_Index ON master(nameFirst,nameLast);



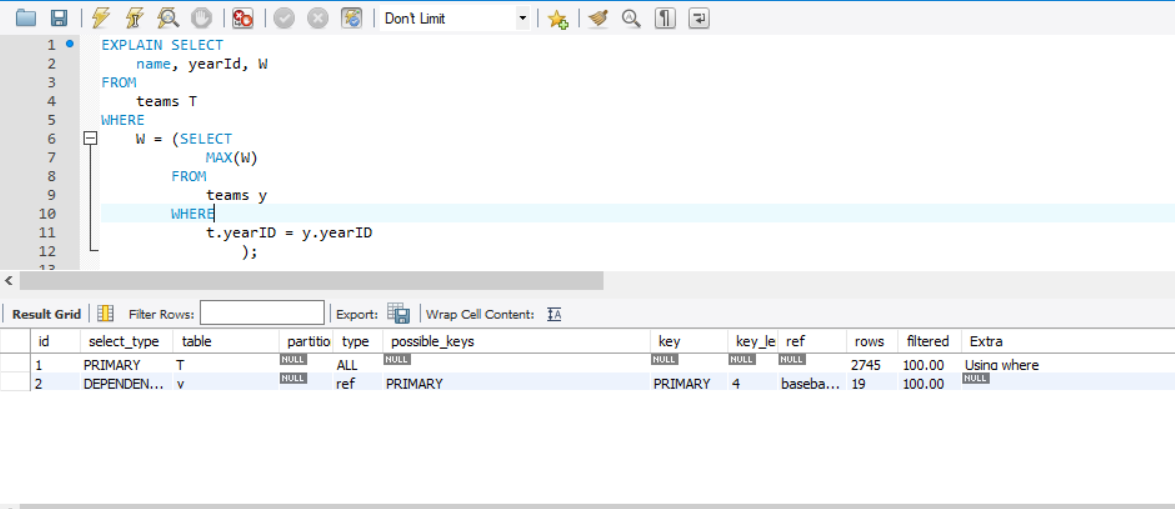
This results in significant improvement. As we can see, the number of rows it has to read now is way lesser than we had earlier.

Also, the query after this optimization runs in 1.1 seconds as compared to earlier which was ~6.5 seconds.

**Query 4**

The below query cost comes out to be 2745 rows for the team table. Here I can try the following approaches-

1. Apply an index on W since it is being used in the where clause
2. Apply a composite index on yearID, W since we first use yearID to find a max W then we use W in the query.
3. Try out a join instead of a subquery and then apply a composite index on W, yearid.

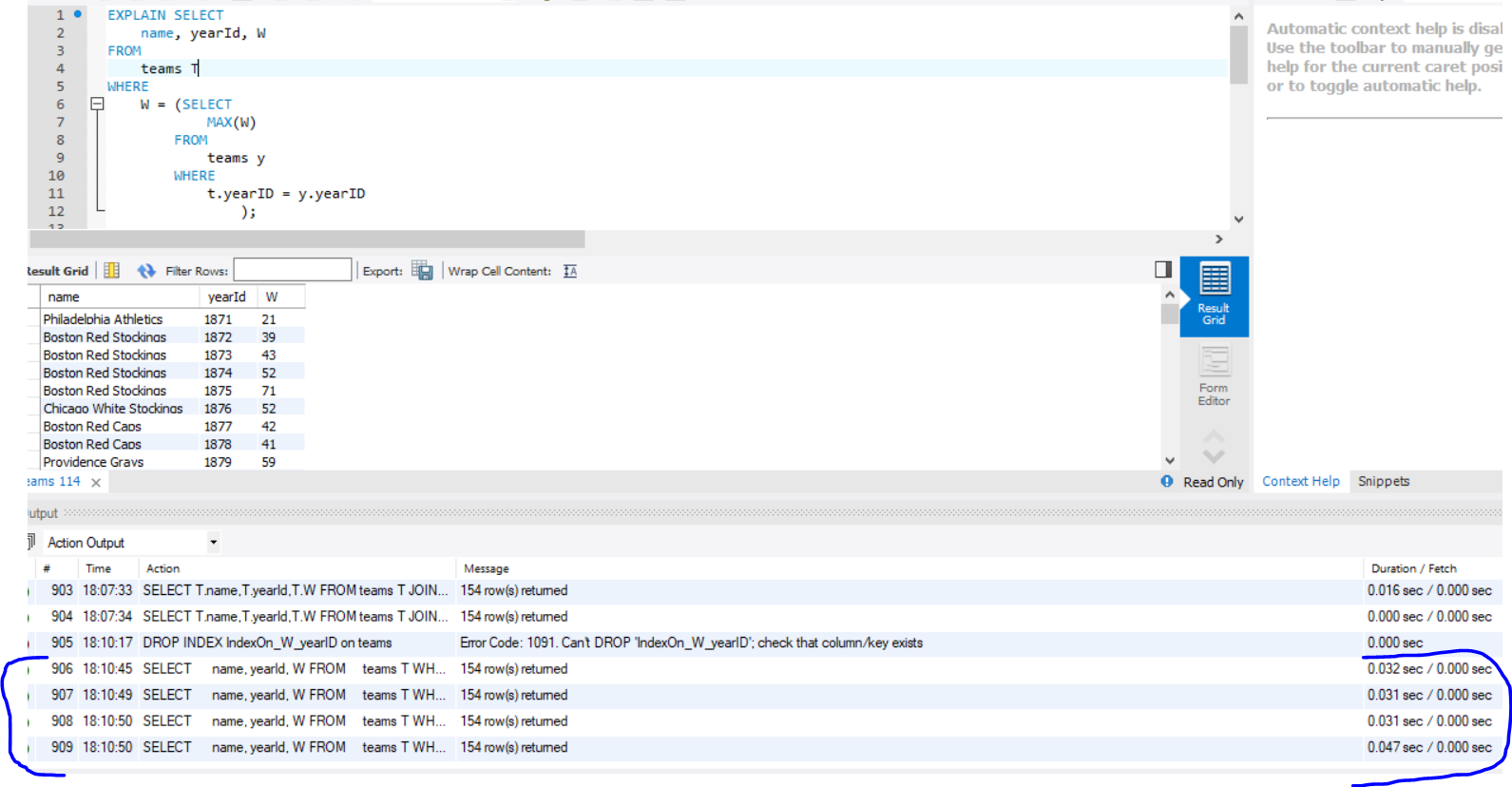


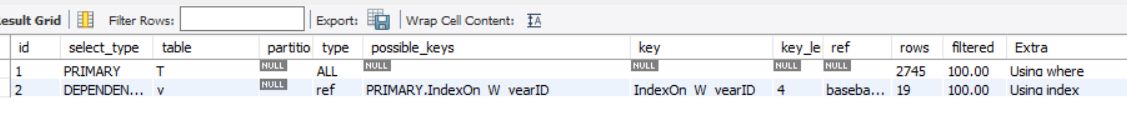
Approach 1 – This didn’t work out because it still has to go through all the rows in the subquery to find out a valid yearId, and then it finds the max. The index did not improve any performance here.

Approach 2 – This slightly improved the performance. The performance improved by 40% on average.

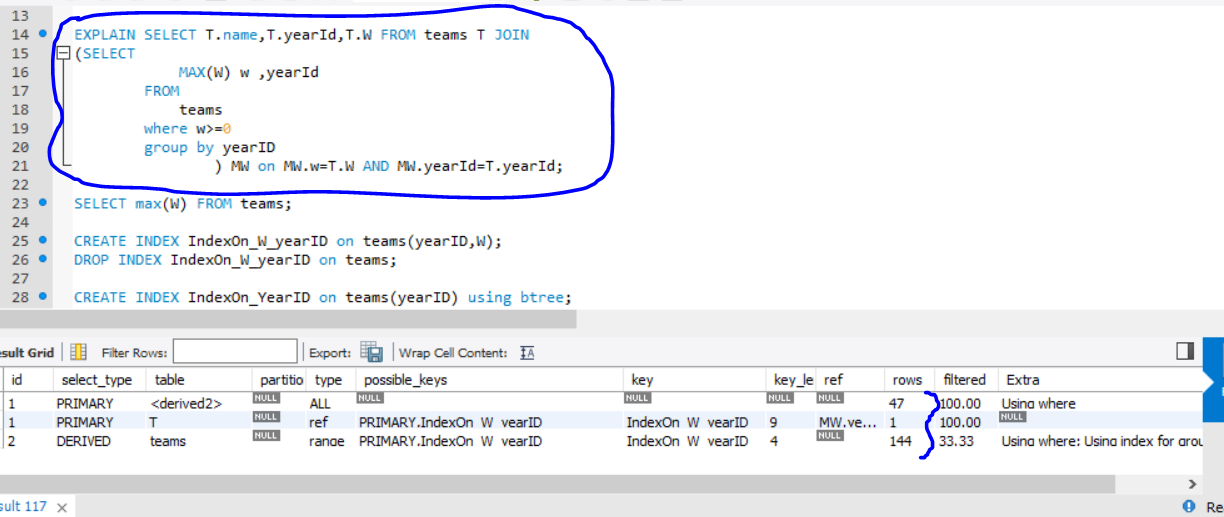
Approach 3 – This significantly improved the performance. The performance differences are as below

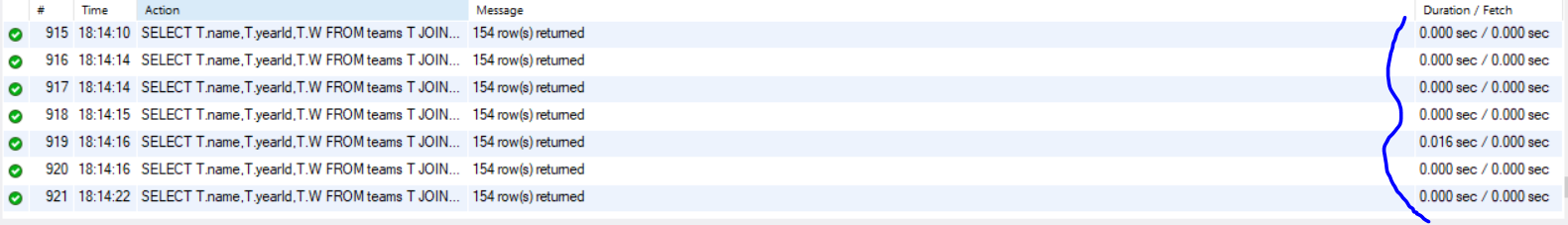
Original Query-





Modified Query

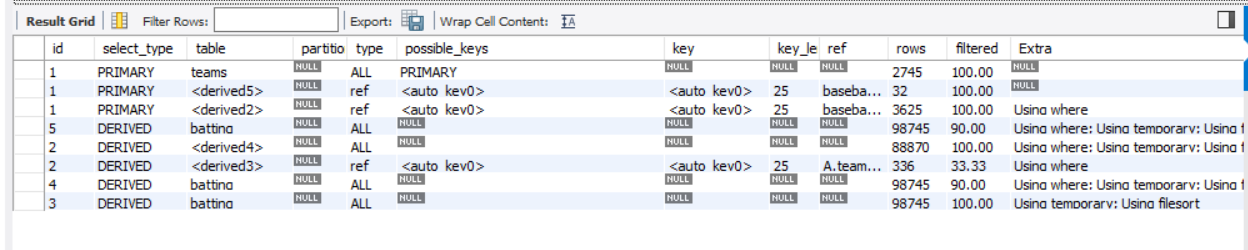




So, using a join instead of a subquery and then creating a composite index on yearId,W helped improve the query performance overall.

**Query 5**

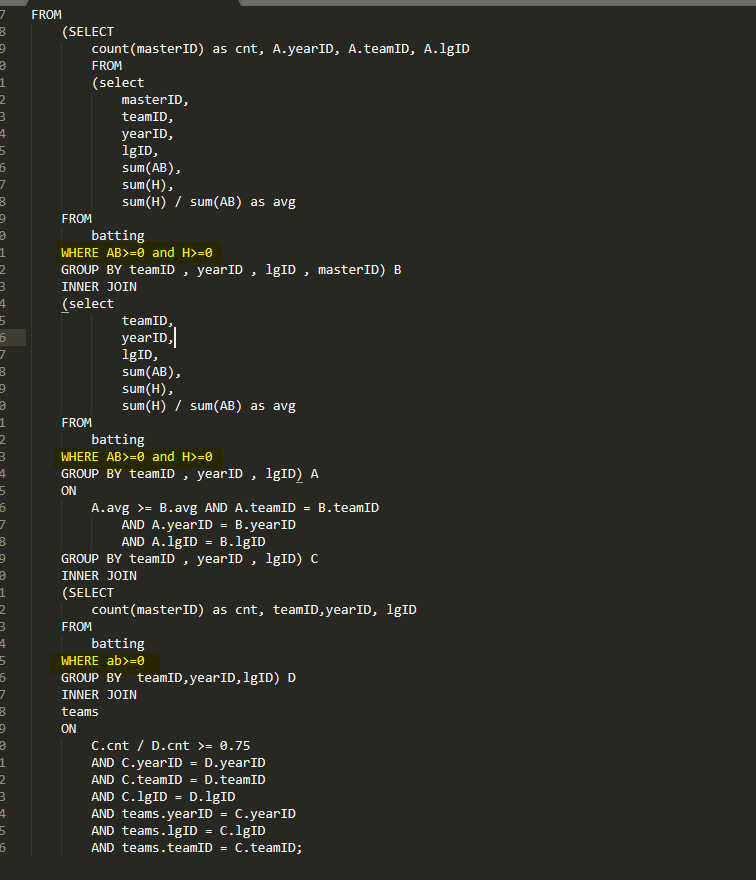
In this query, I observe the following things-



The use of AB and H is excessive from the batting table. Although we’re grouping by teamID, yearID and lgID, even if we apply a composite index on these three, the performance degrades, because it has to look through the batting table anyway to group the tuples.

What we can do here is reduce the number of tuples before we group them. We are summing up AB and H and also using ‘AB is not null’ in the where clause. Now, while grouping them, we can ignore all the tuples that do not contribute to this aggregation by applying a where clause in such a way that we don’t have to go through all these non-contributing tuples.

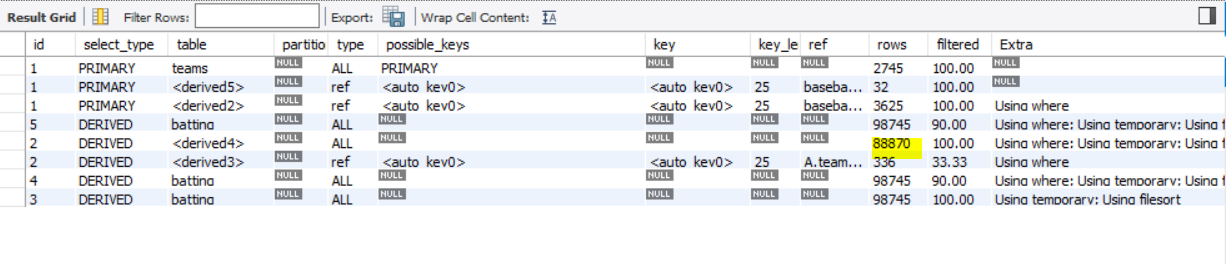
Here is the re-written query-



I’ve highlighted the changes I’ve made in yellow. These changes help in reducing the number of tuples we need to group.

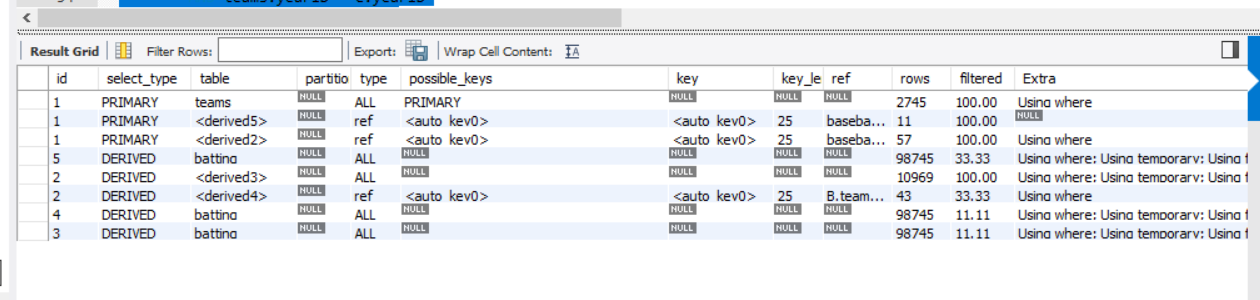
Below are the performance statistics-

Original Query





Modified Query





I also tried applying index on AB and H so that it is easier to search the when we use the where clause but the performance didn’t speed up. This is one of the cases where using an index does not increase performance.

As it can be seen from the screenshots, the query performance is 14 times better now.

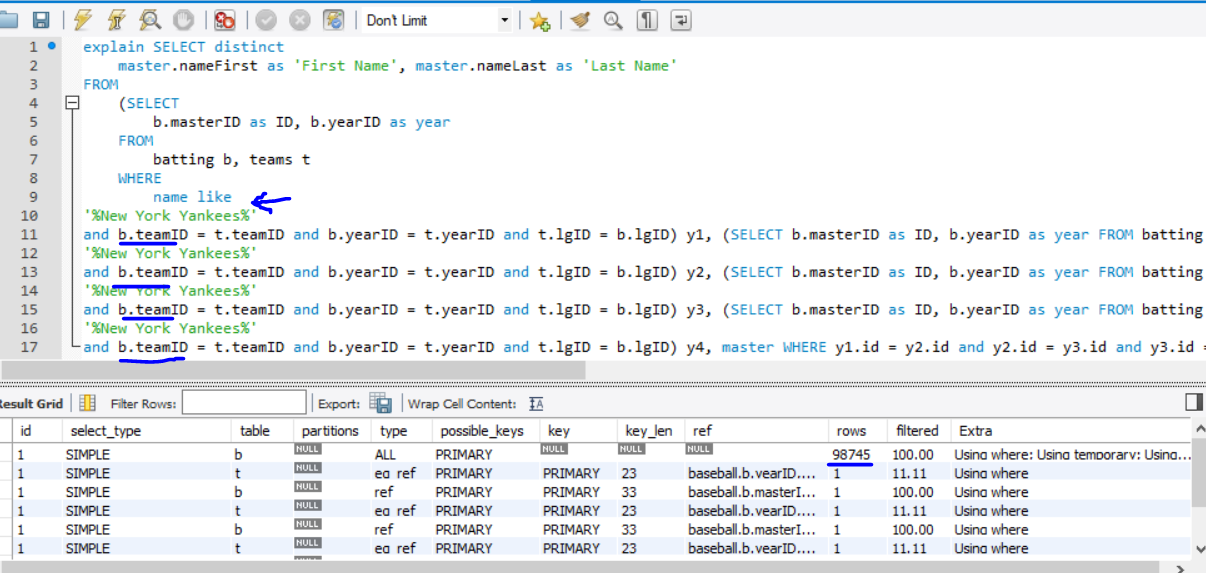
**Query 6**

In this query, I observe three things-

One, that the cost of scanning the batting table b is quite a lot(98745 rows). We can try adding an index on the teamID, yearID and leagueID columns, to improve cost.

Second, instead of using the like keyword on the name column in teams table, we can use =

Third, Since we are trying to find a specific team name, we could use an index on the name column of the teams table.



What I did

Created an index on teamID, yearID and lgID columns in the batting table.

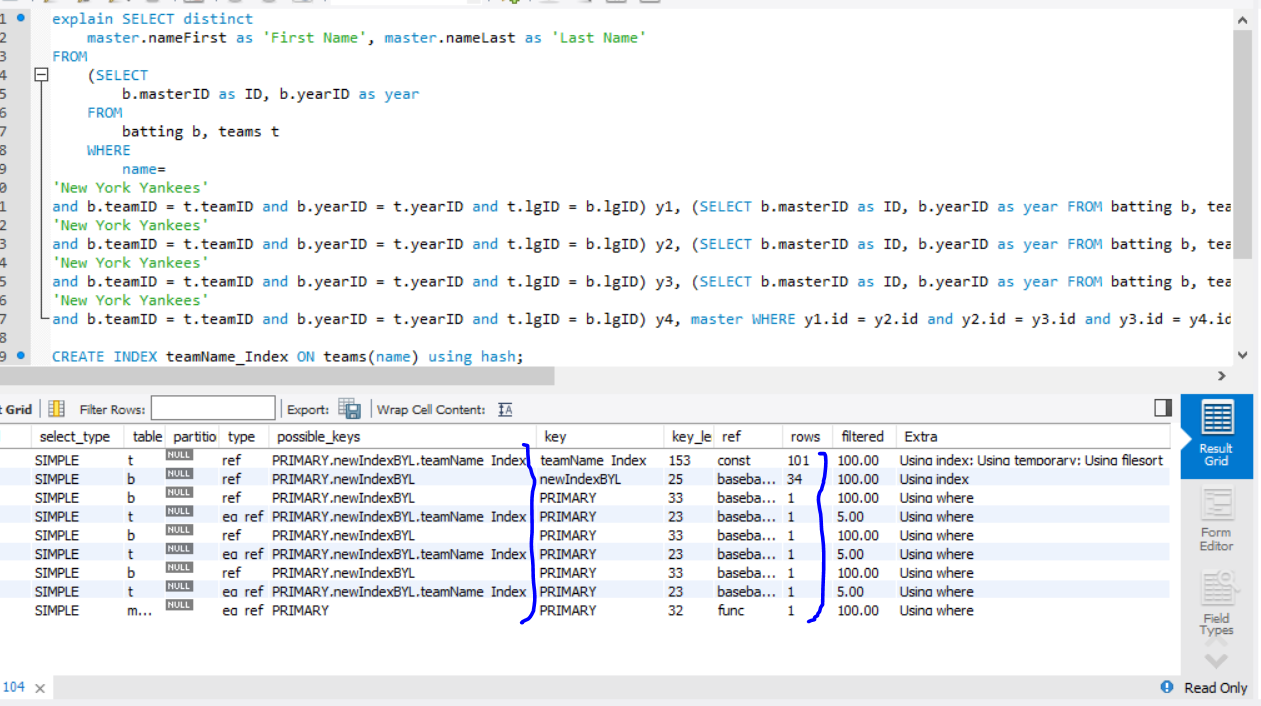
Created an index on teamID, yearID and lgID columns in the teams table.

Removed the like keyword and replaced it with =

Created a hash index on teamName column in the teams table.

Improvement

The cost came down from 98745 rows to 101 rows.

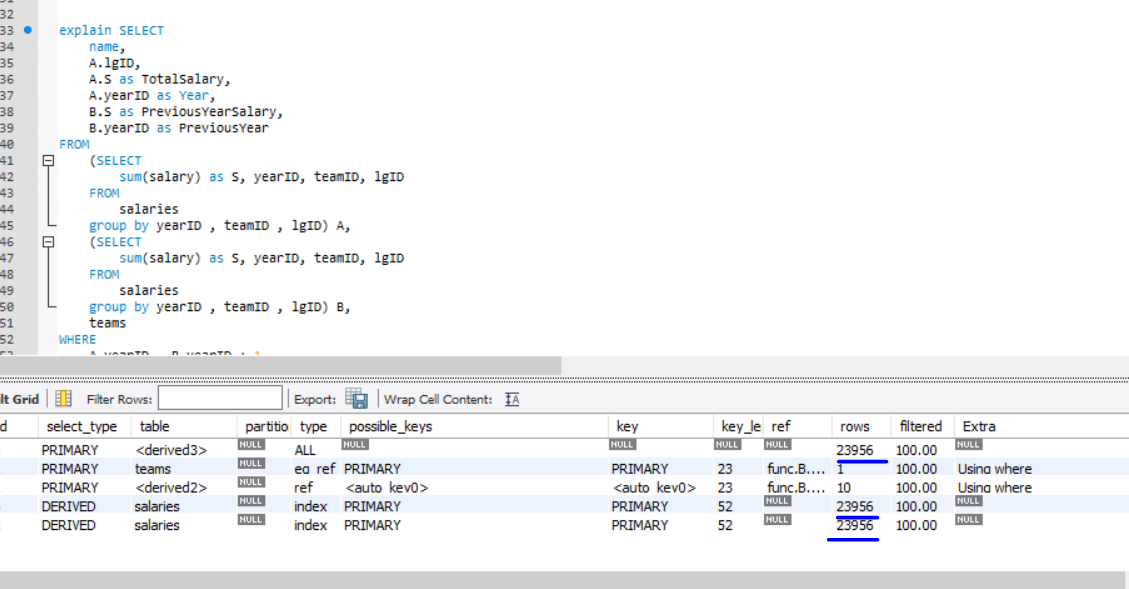


**Query 7**

In the given query, the current cost is 23956 rows. This cost stems from the grouping that is being applied on yearID,teamID and lgID 2 times to find out the sum of the salaries.

So, the two derived tables can be optimized by the following approach-

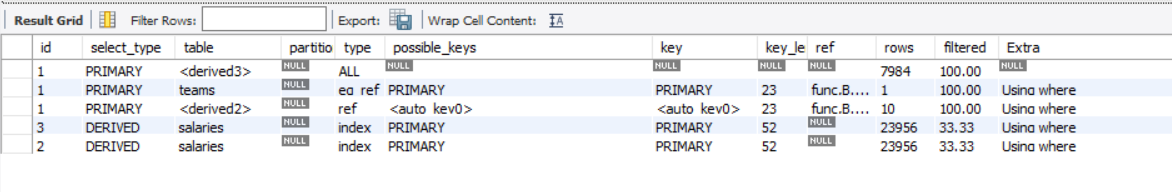
1. Use where before the group by clause. Apply where clause on salary. This way we will have lesser number of rows to group.
2. Try creating a composite index using yearID,teamID,lgID and salary.
3. Apply indexing on the salary column of the salaries table, so that when we use the where clause, we can reduce the cost by reducing the number of rows we need to traverse.



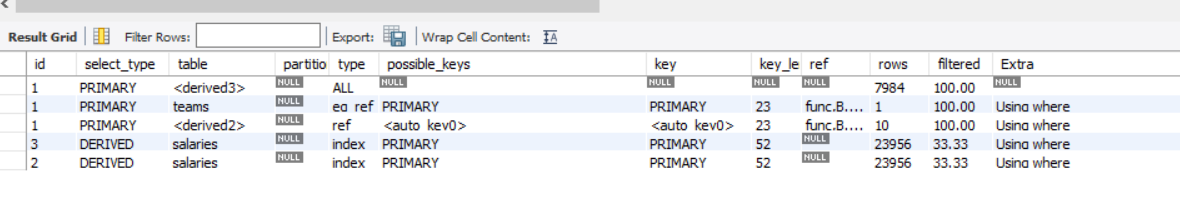
I modified the query and this is how it looks like now-



Now this is the optimization result after I amend the query with where clause mentioned in the 1st approach-

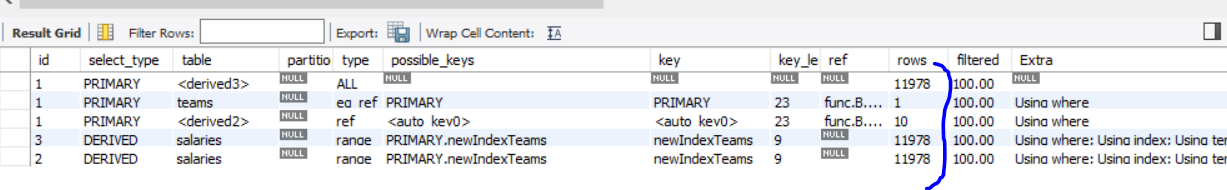


This performs better but now we add the 2nd approach of creating a composite index. These are the results when we do that-



The cost remains the same. Perhaps, because we already have an index created on the primary keys of the salaries table. So, its already using an index.

Now we try the 3rd approach of creating an index on the salary column. These are the results when we do that-



We have halved the cost, which means 50% improvement.

So, we optimized this query by two ways- Restricting the number of tuples that need to be grouped by using a where clause before group by and applying an index on the column that is being used in the where clause before the group by clause.