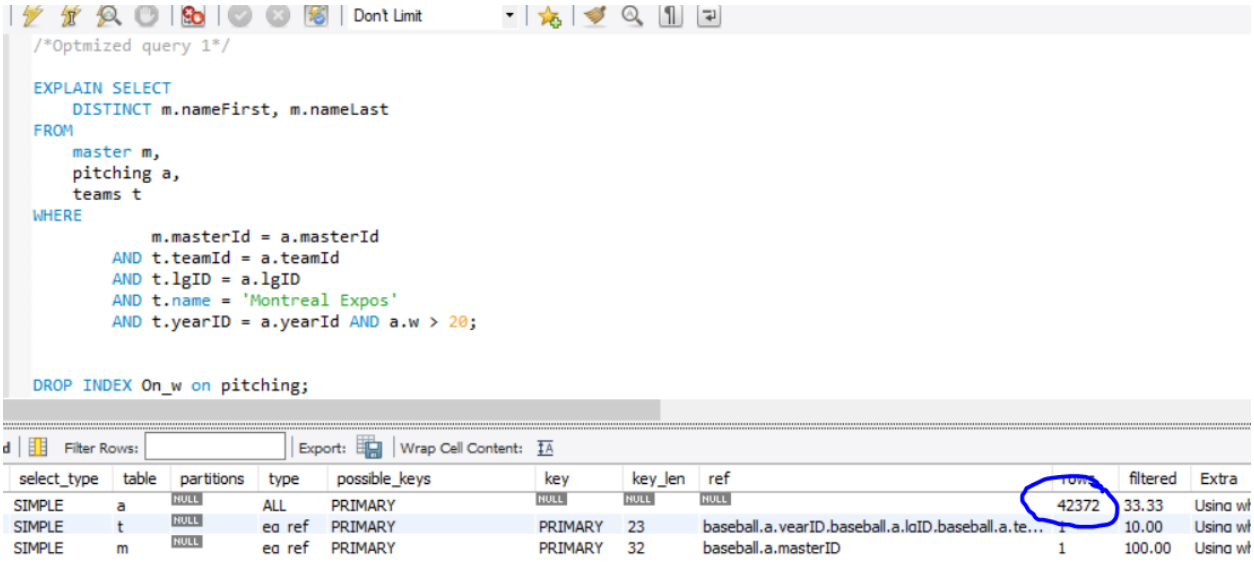
Query 1:

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.



```
/*Optmized query 1*/

EXPLAIN SELECT
    DISTINCT m.nameFirst, m.nameLast
FROM
    master m,
    pitching a,
    teams t
WHERE
            m.masterId = a.masterId
        AND t.teamId = a.teamId
        AND t.lgID = a.lgID
        AND t.name = 'Montreal Expos'
        AND t.yearID = a.yearId AND a.w > 20;


DROP INDEX On_w on pitching;
```

Filter Rows: | Export: | Wrap Cell Content:

select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
SIMPLE	a	NULL	ALL	PRIMARY	NULL	NULL	NULL	42372	33.33	Using wh
SIMPLE	t	NULL	eq_ref	PRIMARY	PRIMARY	23	baseball.a.yearID,baseball.a.lgID,baseball.a.te...	1	10.00	Using wh
SIMPLE	m	NULL	eq_ref	PRIMARY	PRIMARY	32	baseball.a.masterID	1	100.00	Using wh

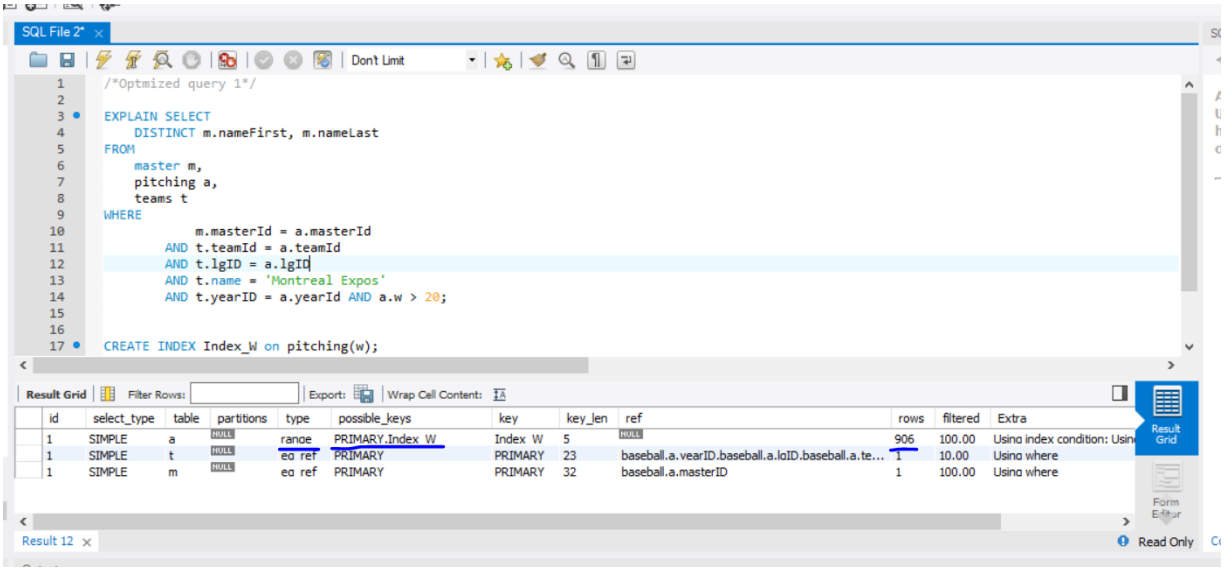
Firstly, I removed the like keyword and replaced it with =, which improves the efficiency of finding matching rows by a very slight margin.

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.

```
explain SELECT
    h / ab as Average, h as Hits, ab as 'At Bats', nameFirst as 'First Name', nameLast as 'Last Name', batting.yearID as Year
FROM
    batting,
    master
WHERE
    ab is not null
        and batting.masterID = master.masterID
        AND master.masterID IN (SELECT
            masterID
        FROM
            schoolsplayers
        WHERE
            schoolID in (SELECT
                    schoolID
                FROM
                    schools
                WHERE
                    schoolName like '%Utah State%'
)) order by year;
```

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

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	schoolsplayers	NULL	index	PRIMARY	PRIMARY	76	NULL	6147	94.84	Using index; Using temporary; U
1	SIMPLE	schools	NULL	eq_ref	PRIMARY	PRIMARY	47	baseball.schoolsplayers.schoolID	1	11.11	Using where; FirstMatch(schools
1	SIMPLE	master	NULL	eq_ref	PRIMARY	PRIMARY	32	baseball.schoolsplayers.masterID	1	100.00	Using where
1	SIMPLE	batting	NULL	ref	PRIMARY	PRIMARY	29	baseball.master.masterID	5	90.00	Using where

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);

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

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	<subquery2>	NULL	ALL	NULL	NULL	NULL	NULL	NULL	100.00	Using temporary; Using
1	SIMPLE	master	NULL	eq_ref	PRIMARY	PRIMARY	32	<subquery2>.masterID	1	100.00	Using where
1	SIMPLE	batting	NULL	ref	PRIMARY	PRIMARY	29	baseball.master.masterID	5	90.00	Using where
2	MATERIALIZED	schools	NULL	ALL	PRIMARY	NULL	NULL	NULL	749	11.11	Using where
2	MATERIALIZED	schoolsplayers	NULL	ref	PRIMARY,Index_On_Play...	Index_On_SchoolID	47	baseball.schools.schoolID	8	100.00	Using index

Result 16

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);

```
explain SELECT
    h / ab as Average, h as Hits, ab as 'At Bats', nameFirst as 'First Name', nameLast as 'Last Name', batting.yearID as Year
FROM
    batting,
    master
WHERE
    ab is not null
        and batting.masterID = master.masterID
        AND master.masterID IN (SELECT
            masterID
        FROM
            schoolsplayers
        WHERE
            schoolID in (SELECT
                    schoolID
                FROM
                    schools
                WHERE
                    schoolName like '%Utah State%'
)) order by year;
```

Filter Rows: | Export: | Wrap Cell Content:

select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
SIMPLE	<subquery2>	NULL	ALL	NULL	NULL	NULL	NULL	NULL	100.00	Using temporary; Using
SIMPLE	master	NULL	eq_ref	PRIMARY	PRIMARY	32	<subquery2>.masterID	1	100.00	Using where
SIMPLE	batting	NULL	ref	PRIMARY	PRIMARY	29	baseball.master.masterID	5	90.00	Using where
MATERIALIZED	schools	NULL	ref	PRIMARY,Index On Scho...	Index On SchoolName	768	const	1	100.00	Using index
MATERIALIZED	schoolsplayers	NULL	ref	PRIMARY,Index On Play...	Index On SchoolID	47	baseball.schools.schoolID	8	100.00	Using index

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-

```
1  explain SELECT
2      h / ab as Average, h as Hits, ab as 'At Bats', nameFirst as 'First Name', nameLast as 'Last Name', batting.yearID as Year
3  FROM
4      batting,
5      master
6  WHERE
7      ab is not null
8          and batting.masterID = master.masterID
9          AND master.masterID IN (SELECT
10             masterID
11         FROM
12             schoolsplayers
13         WHERE
14             schoolID in (SELECT
15                     schoolID
16                 FROM
17                     schools
18                 WHERE
19                     schoolName ='Utah State University'
20 )) order by year;
```

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

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	<subquery2>	NULL	ALL	NULL	NULL	NULL	NULL	NULL	100.00	Using temporary; Using
1	SIMPLE	master	NULL	eq_ref	PRIMARY	PRIMARY	32	<subquery2>.masterID	1	100.00	Using where
1	SIMPLE	batting	NULL	ref	PRIMARY	PRIMARY	29	baseball.master.masterID	5	90.00	Using where
2	MATERIALIZED	schools	NULL	ref	PRIMARY,Index On Scho...	Index On SchoolName	768	const	1	100.00	Using index
2	MATERIALIZED	schoolsplayers	NULL	ref	PRIMARY,Index On Play...	Index On SchoolID	47	baseball.schools.schoolID	8	100.00	Using index

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.

```
Explain SELECT distinct jeter.masterID, jeterT.masterID, jeterTY.masterID, jeterTT.masterID
FROM
    master m,
    appearances jeter,
    appearances jeterT,
    appearances jeterTY,
    appearances jeterTT
WHERE
        m.masterID = jeter.masterID
       AND m.nameLast = 'Jeter'
AND m.nameFirst ='Derek'
AND jeter.teamID = jeterT.teamID AND jeter.yearID = jeterT.yearID AND jeter.lgID = jeterT.lgID AND jeter.masterID <> jeterT.masterID
AND jeterT.masterID = jeterTY.masterID
AND jeterTY.teamID = jeterTT.teamID and jeterTY.yearID = jeterTT.yearID AND jeterTY.lgID = jeterTT.lgID AND jeterTY.masterID <> jeterTT.masterI
AND jeterTT.masterID <> jeter.masterID
AND jeter.teamID <> jeterTY.teamID
```

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

select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
SIMPLE	jeter	NULL	ALL	PRIMARY	NULL	NULL	NULL	97811	100.00	Using temporary
SIMPLE	m	NULL	eq_ref	PRIMARY	PRIMARY	32	baseball.jeter.mast...	1	5.00	Using where
SIMPLE	jeterTY	NULL	ALL	PRIMARY	NULL	NULL	NULL	97811	90.00	Using where; Using join buffer (Block Nested Lo...
SIMPLE	jeterT	NULL	eq_ref	PRIMARY	PRIMARY	44	baseball.jeter.yearI...	1	10.00	Using where
SIMPLE	jeterTT	NULL	ref	PRIMARY	PRIMARY	15	baseball.jeterTY.ye...	36	8.10	Using where

Result Grid | Form Editor | Read Only

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-

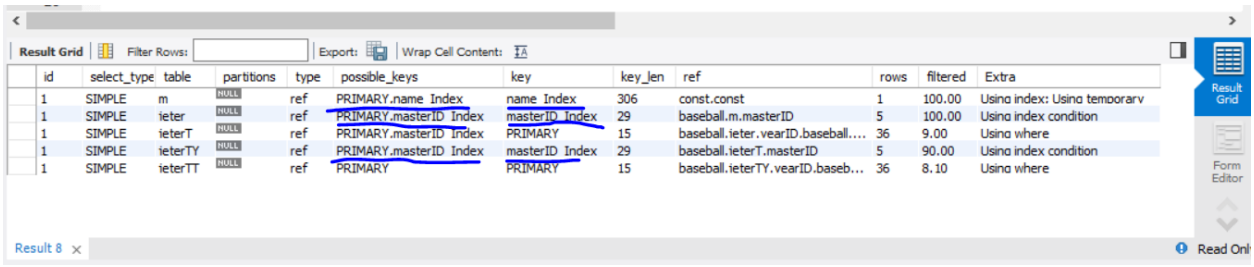
Result Grid | Filter Rows: | Export: | Wrap Cell Content:

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	m	NULL	ALL	PRIMARY	NULL	NULL	NULL	18224	1.00	Using where; Using temporary
1	SIMPLE	jeter	NULL	ref	PRIMARY,masterID_Index	masterID_Index	29	baseball.m.masterID	5	100.00	Using index condition
1	SIMPLE	jeterT	NULL	ref	PRIMARY,masterID_Index	PRIMARY	15	baseball.jeter.yearID,baseball....	36	9.00	Using where
1	SIMPLE	jeterTY	NULL	ref	PRIMARY,masterID_Index	masterID_Index	29	baseball.jeterT.masterID	5	90.00	Using index condition
1	SIMPLE	jeterTT	NULL	ref	PRIMARY	PRIMARY	15	baseball.jeterTY.yearID,baseb...	36	8.10	Using where

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);



id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	m	NULL	ref	PRIMARY,name_Index	name_Index	306	const,const	1	100.00	Using index; Using temporary
1	SIMPLE	ieter	NULL	ref	PRIMARY,masterID_Index	masterID_Index	29	baseball.m.masterID	5	100.00	Using index condition
1	SIMPLE	ieterT	NULL	ref	PRIMARY,masterID_Index	PRIMARY	15	baseball.ieter.yearID,baseball....	36	9.00	Using where
1	SIMPLE	ieterTY	NULL	ref	PRIMARY,masterID_Index	masterID_Index	29	baseball.ieterT.masterID	5	90.00	Using index condition
1	SIMPLE	ieterTT	NULL	ref	PRIMARY	PRIMARY	15	baseball.ieterTY.yearID,baseb...	36	8.10	Using where

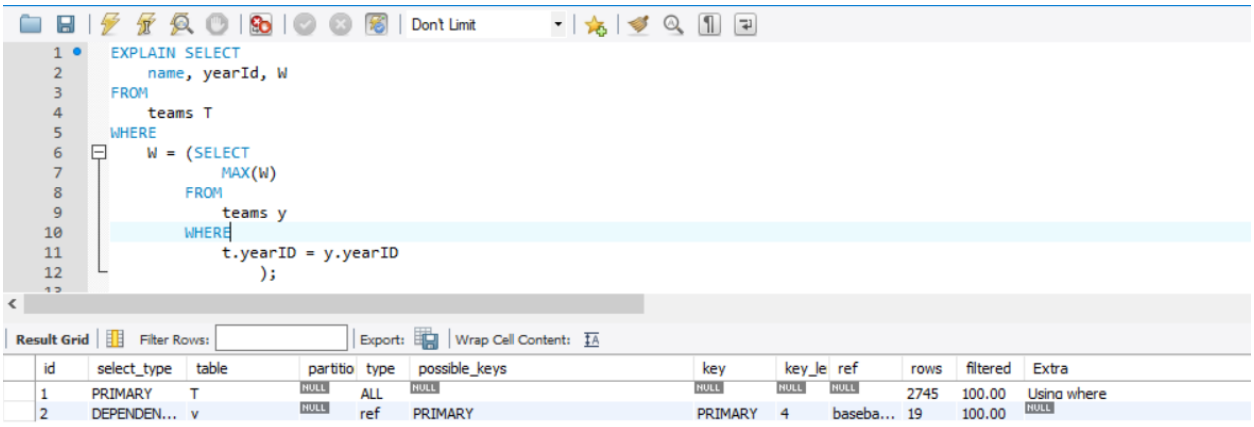
This results in significant improvement. As we can see, the number of rows it has to read now is way lesser than we had earlier.

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.



```
EXPLAIN SELECT
    name, yearId, W
FROM
    teams T
WHERE
    W = (SELECT
            MAX(W)
        FROM
            teams y
        WHERE
            t.yearID = y.yearID
                );
```

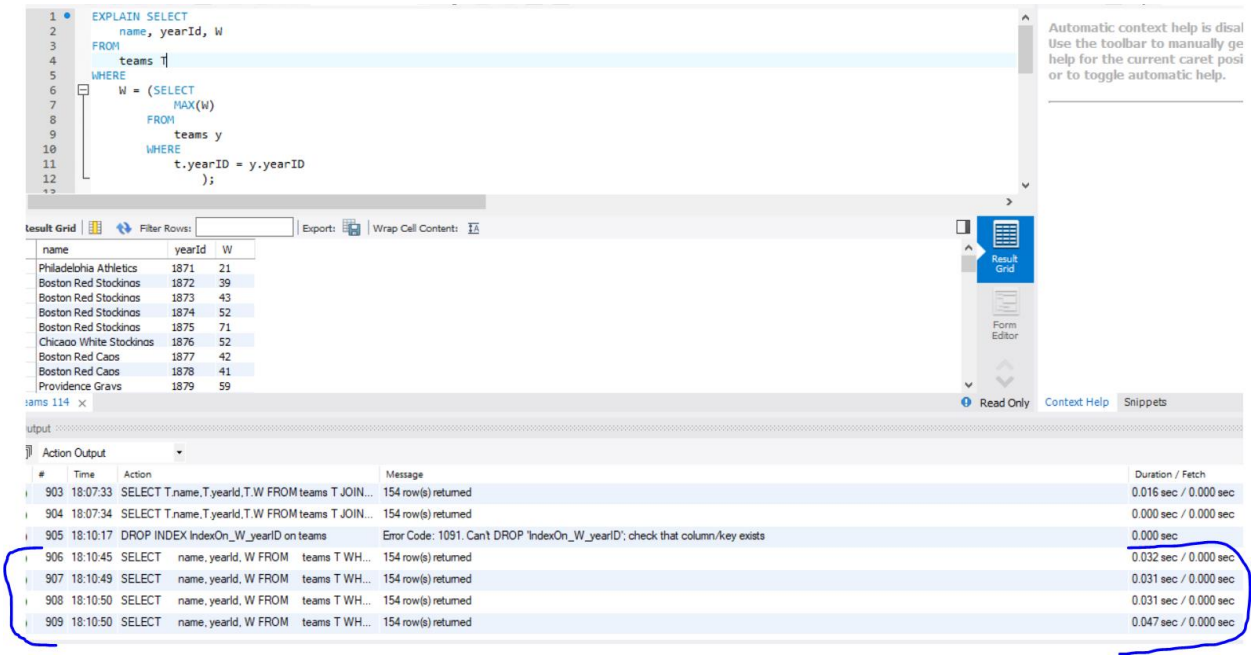
id	select_type	table	partitio	type	possible_keys	key	key_le	ref	rows	filtered	Extra
1	PRIMARY	T	NULL	ALL	NULL	NULL	NULL	NULL	2745	100.00	Using where
2	DEPENDEN...	y	NULL	ref	PRIMARY	PRIMARY	4	baseba...	19	100.00	NULL

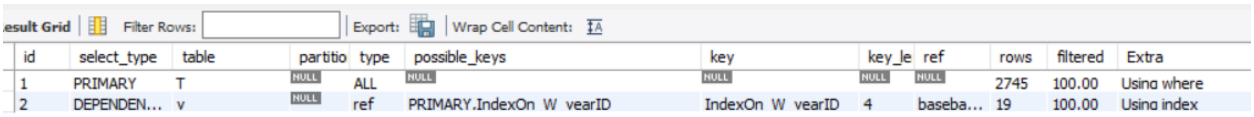
Approach 1 – This didn't work out because it still has to go through all the rows in the subquery to find out a valid yearId, and then it finds the max. The index did not improve any performance here.

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

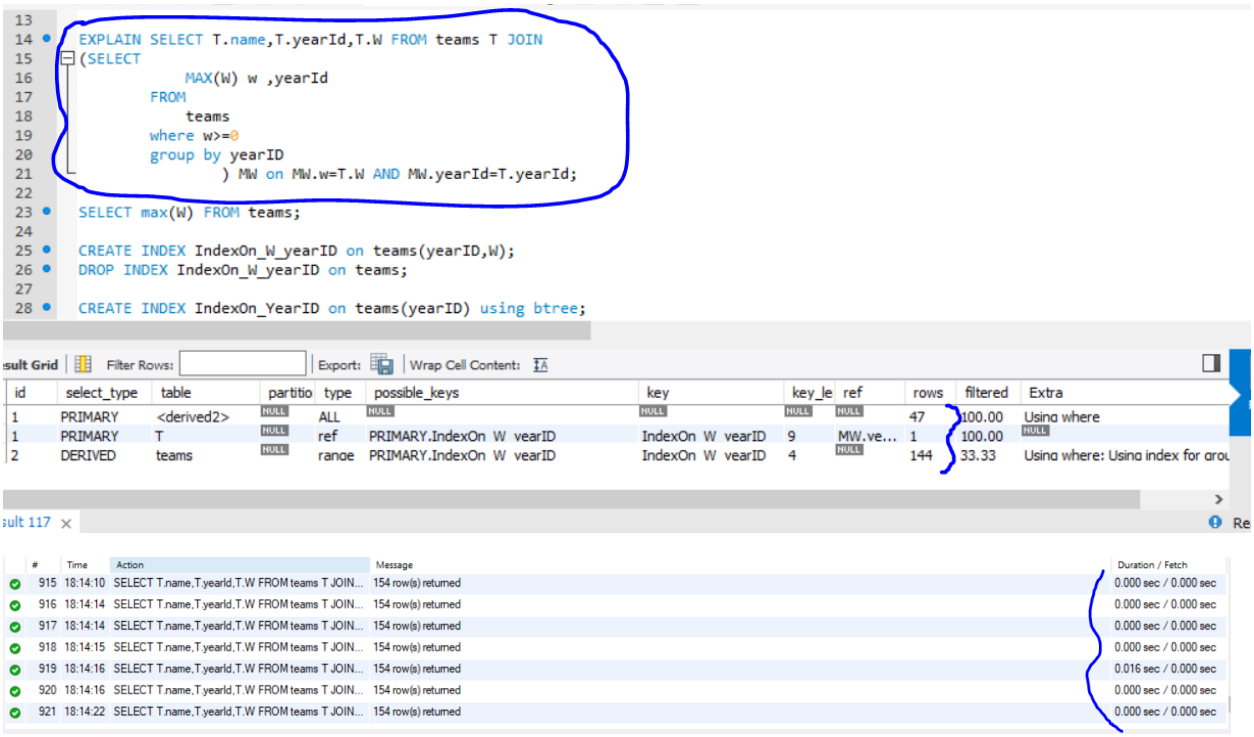
Approach 3 – This significantly improved the performance. The performance differences are as below

<u>Original Query-</u>





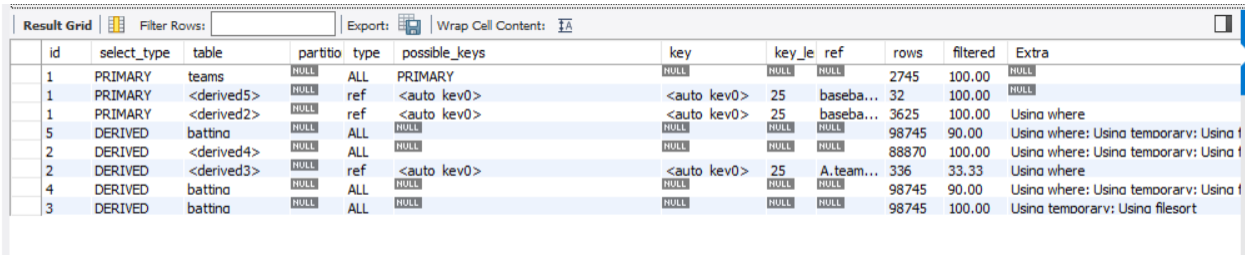
Modified Query



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-



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

id	select_type	table	partitio	type	possible_keys	key	key_le	ref	rows	filtered	Extra
1	PRIMARY	teams	NULL	ALL	PRIMARY	NULL	NULL	NULL	2745	100.00	NULL
1	PRIMARY	<derived5>	NULL	ref	<auto_key0>	<auto_key0>	25	baseba...	32	100.00	NULL
1	PRIMARY	<derived2>	NULL	ref	<auto_key0>	<auto_key0>	25	baseba...	3625	100.00	Using where
5	DERIVED	batting	NULL	ALL	NULL	NULL	NULL	NULL	98745	90.00	Using where; Using temporary; Using f
2	DERIVED	<derived4>	NULL	ALL	NULL	NULL	NULL	NULL	88870	100.00	Using where; Using temporary; Using f
2	DERIVED	<derived3>	NULL	ref	<auto_key0>	<auto_key0>	25	A.team...	336	33.33	Using where
4	DERIVED	batting	NULL	ALL	NULL	NULL	NULL	NULL	98745	90.00	Using where; Using temporary; Using f
3	DERIVED	batting	NULL	ALL	NULL	NULL	NULL	NULL	98745	100.00	Using temporary; Using filesort

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.

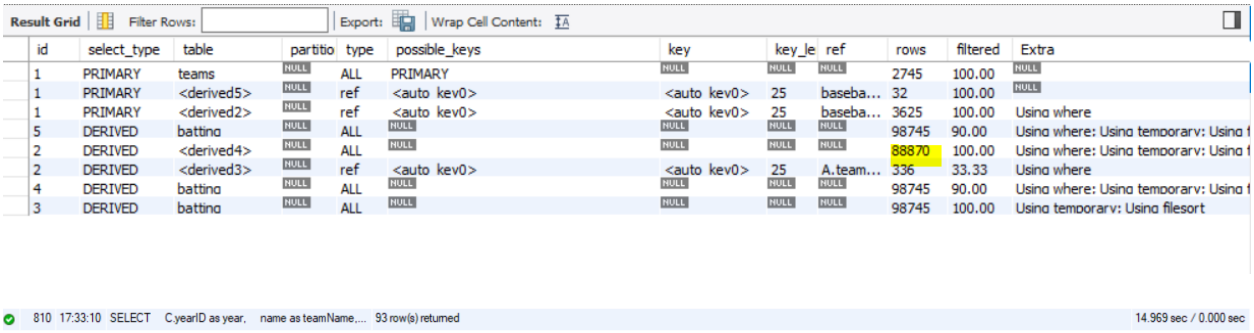
Here is the re-written query-

```
FROM
    (SELECT
        count(masterID) as cnt, A.yearID, A.teamID, A.lgID
        FROM
        (select
            masterID,
            teamID,
            yearID,
            lgID,
            sum(AB),
            sum(H),
            sum(H) / sum(AB) as avg
    FROM
        batting
    WHERE AB>=0 and H>=0
    GROUP BY teamID , yearID , lgID , masterID) B
    INNER JOIN
    (select
            teamID,
            yearID,
            lgID,
            sum(AB),
            sum(H),
            sum(H) / sum(AB) as avg
    FROM
        batting
    WHERE AB>=0 and H>=0
    GROUP BY teamID , yearID , lgID) A
    ON
        A.avg >= B.avg AND A.teamID = B.teamID
            AND A.yearID = B.yearID
            AND A.lgID = B.lgID
    GROUP BY teamID , yearID , lgID) C
    INNER JOIN
    (SELECT
        count(masterID) as cnt, teamID,yearID, lgID
    FROM
        batting
    WHERE ab>=0
    GROUP BY  teamID,yearID,lgID) D
    INNER JOIN
    teams
    ON
        C.cnt / D.cnt >= 0.75
        AND C.yearID = D.yearID
        AND C.teamID = D.teamID
        AND C.lgID = D.lgID
        AND teams.yearID = C.yearID
        AND teams.lgID = C.lgID
        AND teams.teamID = C.teamID;
```

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

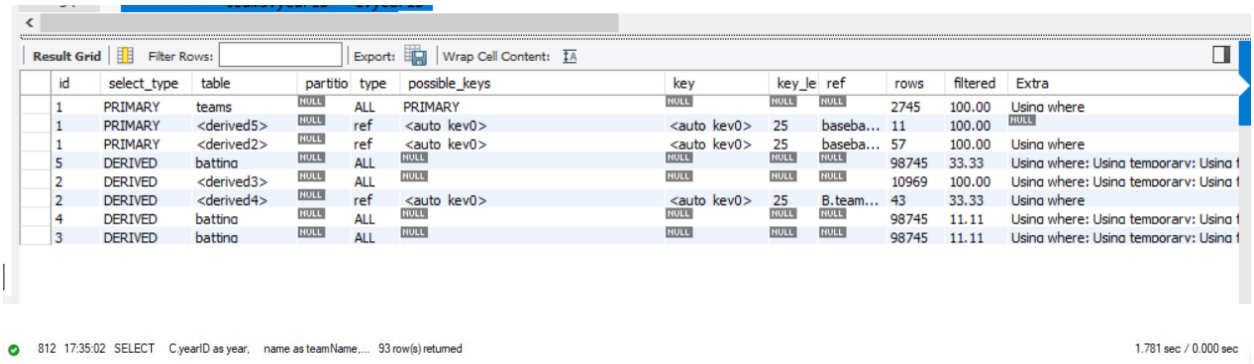


Result Grid | Filter Rows: | Export: | Wrap Cell Content:

id	select_type	table	partitio	type	possible_keys	key	key_le	ref	rows	filtered	Extra
1	PRIMARY	teams	NULL	ALL	PRIMARY	NULL	NULL	NULL	2745	100.00	NULL
1	PRIMARY	<derived5>	NULL	ref	<auto_key0>	<auto_key0>	25	baseba...	32	100.00	NULL
1	PRIMARY	<derived2>	NULL	ref	<auto_key0>	<auto_key0>	25	baseba...	3625	100.00	Using where
5	DERIVED	batting	NULL	ALL	NULL	NULL	NULL	NULL	98745	90.00	Using where; Using temporary; Using f
2	DERIVED	<derived4>	NULL	ALL	NULL	NULL	NULL	NULL	88870	100.00	Using where; Using temporary; Using f
2	DERIVED	<derived3>	NULL	ref	<auto_key0>	<auto_key0>	25	A.team...	336	33.33	Using where
4	DERIVED	batting	NULL	ALL	NULL	NULL	NULL	NULL	98745	90.00	Using where; Using temporary; Using f
3	DERIVED	batting	NULL	ALL	NULL	NULL	NULL	NULL	98745	100.00	Using temporary; Using filesort

810 17:33:10 SELECT C.yearID as year, name as teamName,... 93 row(s) returned 14.969 sec / 0.000 sec

Modified Query



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

id	select_type	table	partitio	type	possible_keys	key	key_le	ref	rows	filtered	Extra
1	PRIMARY	teams	NULL	ALL	PRIMARY	NULL	NULL	NULL	2745	100.00	Using where
1	PRIMARY	<derived5>	NULL	ref	<auto_key0>	<auto_key0>	25	baseba...	11	100.00	NULL
1	PRIMARY	<derived2>	NULL	ref	<auto_key0>	<auto_key0>	25	baseba...	57	100.00	Using where
5	DERIVED	batting	NULL	ALL	NULL	NULL	NULL	NULL	98745	33.33	Using where; Using temporary; Using f
2	DERIVED	<derived3>	NULL	ALL	NULL	NULL	NULL	NULL	10969	100.00	Using where; Using temporary; Using f
2	DERIVED	<derived4>	NULL	ref	<auto_key0>	<auto_key0>	25	B.team...	43	33.33	Using where
4	DERIVED	batting	NULL	ALL	NULL	NULL	NULL	NULL	98745	11.11	Using where; Using temporary; Using f
3	DERIVED	batting	NULL	ALL	NULL	NULL	NULL	NULL	98745	11.11	Using where; Using temporary; Using f

812 17:35:02 SELECT C.yearID as year, name as teamName,... 93 row(s) returned 1.781 sec / 0.000 sec

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.

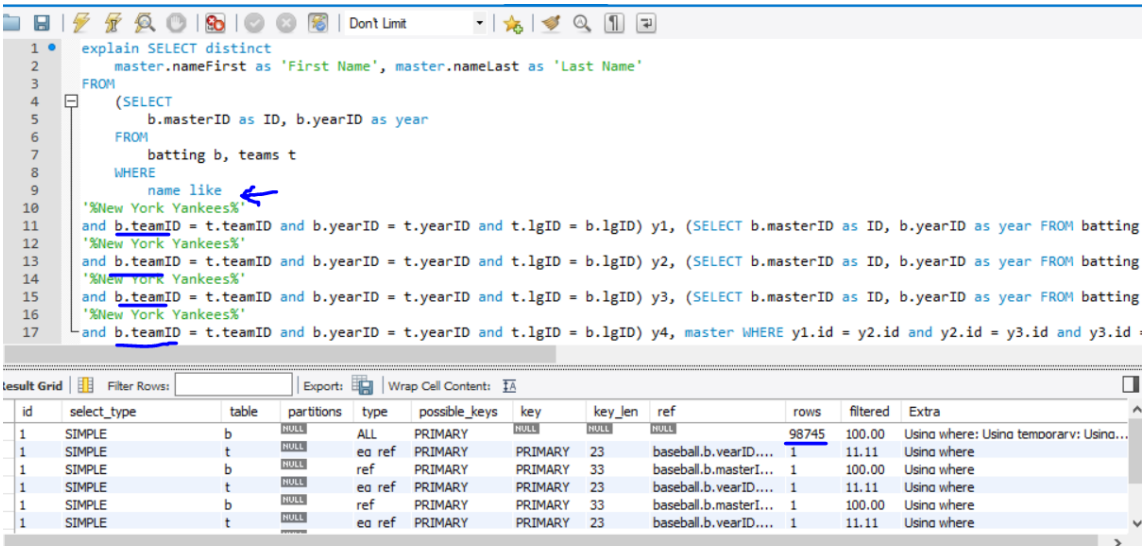
Query 6

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.



```
explain SELECT distinct
    master.nameFirst as 'First Name', master.nameLast as 'Last Name'
FROM
    (SELECT
        b.masterID as ID, b.yearID as year
    FROM
        batting b, teams t
    WHERE
        name like
'%New York Yankees%'
and b.teamID = t.teamID and b.yearID = t.yearID and t.lgID = b.lgID) y1, (SELECT b.masterID as ID, b.yearID as year FROM batting
'%New York Yankees%'
and b.teamID = t.teamID and b.yearID = t.yearID and t.lgID = b.lgID) y2, (SELECT b.masterID as ID, b.yearID as year FROM batting
'%New York Yankees%'
and b.teamID = t.teamID and b.yearID = t.yearID and t.lgID = b.lgID) y3, (SELECT b.masterID as ID, b.yearID as year FROM batting
'%New York Yankees%'
and b.teamID = t.teamID and b.yearID = t.yearID and t.lgID = b.lgID) y4, master WHERE y1.id = y2.id and y2.id = y3.id and y3.id
```

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	b	NULL	ALL	PRIMARY	NULL	NULL	NULL	98745	100.00	Using where; Using temporary; Using...
1	SIMPLE	t	NULL	eq_ref	PRIMARY	PRIMARY	23	baseball.b.yearID,...	1	11.11	Using where
1	SIMPLE	b	NULL	ref	PRIMARY	PRIMARY	33	baseball.t.masterI...	1	100.00	Using where
1	SIMPLE	t	NULL	eq_ref	PRIMARY	PRIMARY	23	baseball.b.yearID,...	1	11.11	Using where
1	SIMPLE	b	NULL	ref	PRIMARY	PRIMARY	33	baseball.t.masterI...	1	100.00	Using where
1	SIMPLE	t	NULL	eq_ref	PRIMARY	PRIMARY	23	baseball.b.yearID,...	1	11.11	Using where

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.

```sql
explain SELECT distinct
    master.nameFirst as 'First Name', master.nameLast as 'Last Name'
FROM
    (SELECT
        b.masterID as ID, b.yearID as year
    FROM
        batting b, teams t
    WHERE
        name=
'New York Yankees'
and b.teamID = t.teamID and b.yearID = t.yearID and t.lgID = b.lgID) y1, (SELECT b.masterID as ID, b.yearID as year FROM batting b, tea
'New York Yankees'
and b.teamID = t.teamID and b.yearID = t.yearID and t.lgID = b.lgID) y2, (SELECT b.masterID as ID, b.yearID as year FROM batting b, tea
'New York Yankees'
and b.teamID = t.teamID and b.yearID = t.yearID and t.lgID = b.lgID) y3, (SELECT b.masterID as ID, b.yearID as year FROM batting b, tea
'New York Yankees'
and b.teamID = t.teamID and b.yearID = t.yearID and t.lgID = b.lgID) y4, master WHERE y1.id = y2.id and y2.id = y3.id and y3.id = y4.ic

CREATE INDEX teamName_Index ON teams(name) using hash;
```

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

select_type	table	partitio	type	possible_keys	key	key_le	ref	rows	filtered	Extra
SIMPLE	t	NULL	ref	PRIMARY.newIndexBYL.teamName Index	teamName Index	153	const	101	100.00	Using index: Using temporary: Using filesort
SIMPLE	b	NULL	ref	PRIMARY.newIndexBYL	newIndexBYL	25	baseba...	34	100.00	Using index
SIMPLE	b	NULL	ref	PRIMARY.newIndexBYL	PRIMARY	33	baseba...	1	100.00	Using where
SIMPLE	t	NULL	eq ref	PRIMARY.newIndexBYL.teamName Index	PRIMARY	23	baseba...	1	5.00	Using where
SIMPLE	b	NULL	ref	PRIMARY.newIndexBYL	PRIMARY	33	baseba...	1	100.00	Using where
SIMPLE	t	NULL	eq ref	PRIMARY.newIndexBYL.teamName Index	PRIMARY	23	baseba...	1	5.00	Using where
SIMPLE	b	NULL	ref	PRIMARY.newIndexBYL	PRIMARY	33	baseba...	1	100.00	Using where
SIMPLE	t	NULL	eq ref	PRIMARY.newIndexBYL.teamName Index	PRIMARY	23	baseba...	1	5.00	Using where
SIMPLE	m...	NULL	eq ref	PRIMARY	PRIMARY	32	func	1	100.00	Using where

104 × Read Only

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.

```
explain SELECT
    name,
    A.lgID,
    A.S as TotalSalary,
    A.yearID as Year,
    B.S as PreviousYearSalary,
    B.yearID as PreviousYear
FROM
    (SELECT
        sum(salary) as S, yearID, teamID, lgID
    FROM
        salaries
    group by yearID , teamID , lgID) A,
    (SELECT
        sum(salary) as S, yearID, teamID, lgID
    FROM
        salaries
    group by yearID , teamID , lgID) B,
    teams
WHERE
```

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

d	select_type	table	partitio	type	possible_keys	key	key_le	ref	rows	filtered	Extra
	PRIMARY	<derived3>	NULL	ALL	NULL	NULL	NULL	NULL	23956	100.00	NULL
	PRIMARY	teams	NULL	eq_ref	PRIMARY	PRIMARY	23	func.B....	1	100.00	Using where
	PRIMARY	<derived2>	NULL	ref	<auto_key0>	<auto_key0>	23	func.B....	10	100.00	Using where
	DERIVED	salaries	NULL	index	PRIMARY	PRIMARY	52	NULL	23956	100.00	NULL
	DERIVED	salaries	NULL	index	PRIMARY	PRIMARY	52	NULL	23956	100.00	NULL

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

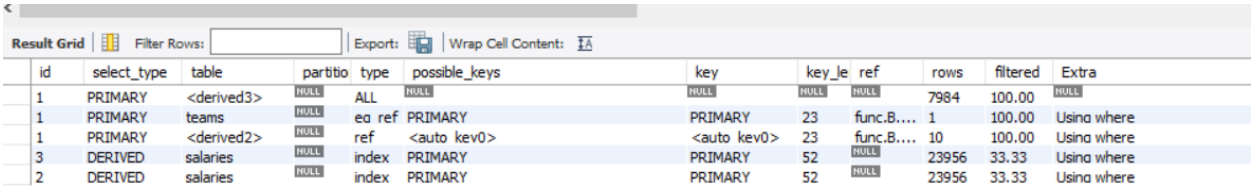
```
Explain SELECT
    name,
    A.lgID,
    A.S as TotalSalary,
    A.yearID as Year,
    B.S as PreviousYearSalary,
    B.yearID as PreviousYear
FROM
    (SELECT
        sum(salary) as S, yearID, teamID, lgID
    FROM
        salaries
    where salary>=0
    group by yearID , teamID , lgID) A,
    (SELECT
        sum(salary) as S, yearID, teamID, lgID
    FROM
        salaries
    where salary>=0
    group by yearID , teamID , lgID) B,
    teams
WHERE
        A.yearID = B.yearID + 1
    AND (A.S * 2) <= (B.S)
    AND A.teamID = B.teamID
    AND A.lgID = B.lgID
    AND teams.yearID = A.yearID
    AND teams.lgID = A.lgID
    AND teams.teamID = A.teamID;
```

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

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

id	select_type	table	partitio	type	possible_keys	key	key_le	ref	rows	filtered	Extra
1	PRIMARY	<derived3>	NULL	ALL	NULL	NULL	NULL	NULL	7984	100.00	NULL
1	PRIMARY	teams	NULL	eq_ref	PRIMARY	PRIMARY	23	func.B....	1	100.00	Using where
1	PRIMARY	<derived2>	NULL	ref	<auto_key0>	<auto_key0>	23	func.B....	10	100.00	Using where
3	DERIVED	salaries	NULL	index	PRIMARY	PRIMARY	52	NULL	23956	33.33	Using where
2	DERIVED	salaries	NULL	index	PRIMARY	PRIMARY	52	NULL	23956	33.33	Using where

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

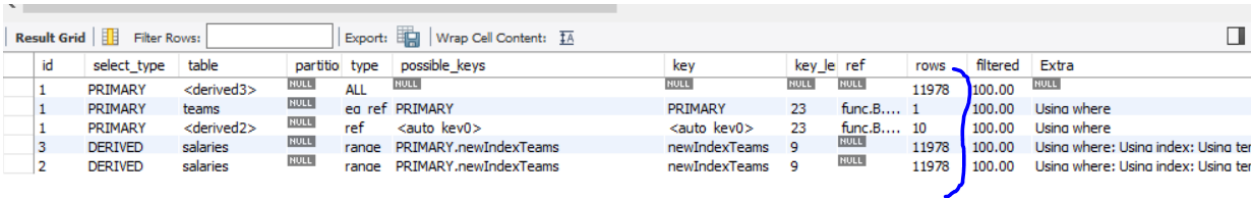


Result Grid | Filter Rows: | Export: | Wrap Cell Content:

id	select_type	table	partitio	type	possible_keys	key	key_le	ref	rows	filtered	Extra
1	PRIMARY	<derived3>	NULL	ALL	NULL	NULL	NULL	NULL	7984	100.00	NULL
1	PRIMARY	teams	NULL	eq_ref	PRIMARY	PRIMARY	23	func.B....	1	100.00	Using where
1	PRIMARY	<derived2>	NULL	ref	<auto_key0>	<auto_key0>	23	func.B....	10	100.00	Using where
3	DERIVED	salaries	NULL	index	PRIMARY	PRIMARY	52	NULL	23956	33.33	Using where
2	DERIVED	salaries	NULL	index	PRIMARY	PRIMARY	52	NULL	23956	33.33	Using where

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-



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

id	select_type	table	partitio	type	possible_keys	key	key_le	ref	rows	filtered	Extra
1	PRIMARY	<derived3>	NULL	ALL	NULL	NULL	NULL	NULL	11978	100.00	NULL
1	PRIMARY	teams	NULL	eq_ref	PRIMARY	PRIMARY	23	func.B....	1	100.00	Using where
1	PRIMARY	<derived2>	NULL	ref	<auto_key0>	<auto_key0>	23	func.B....	10	100.00	Using where
3	DERIVED	salaries	NULL	range	PRIMARY,newIndexTeams	newIndexTeams	9	NULL	11978	100.00	Using where; Using index; Using te
2	DERIVED	salaries	NULL	range	PRIMARY,newIndexTeams	newIndexTeams	9	NULL	11978	100.00	Using where; Using index; Using te

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.