Assignment-8

G1

Xhitij Manish Choudhary
Bhoomika Mandloi
Ramireddy Lakshmi Nageswari
Shreyshi Singh

G2

Likhita Baswani
Paras Jain
Shridhar Somnath Pawar

Question 1

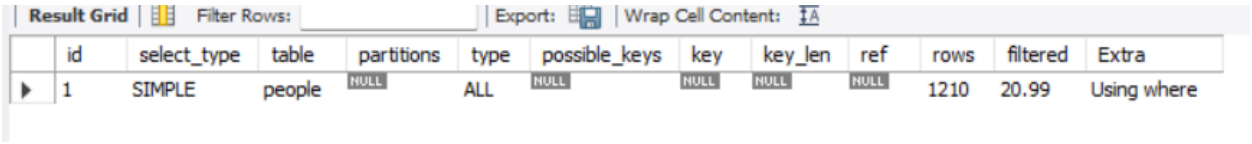
Before Optimization

-> The like operator performs a full table scan for the pattern matching. Hence, it is slow. One can use indexing to avoid full table scans and union to smartly reduce the size of tables over which we perform the scan. This is what we have done in the question.

Query:

```
select * from people where name_of_person like "x%" or name_of_person like "%o";
explain select * from people where name_of_person like "x%" or name_of_person like "%o";
show profiles;
```

Number of scans



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	people	NULL	ALL	NULL	NULL	NULL	NULL	1210	20.99	Using where

Execution Time

21	0.00485225	select * from people where name_of_person lik...

After Optimization

Query

```
(select * from people where name_of_person like "x%") union(select * from people where name_of_person like "%o");
explain (select * from people where name_of_person like "x%") union(select * from people where name_of_person like "%o");
```

Number of scans

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	PRIMARY	people	NULL	ALL	NULL	NULL	NULL	NULL	1210	11.11	Using where
2	UNION	people	NULL	ALL	NULL	NULL	NULL	NULL	1210	11.11	Using where
NULL	UNION RESULT	<union1,2>	NULL	ALL	NULL	NULL	NULL	NULL	NULL	NULL	Using temporary

<u>Execution time:</u>

<u>A better optimization</u>

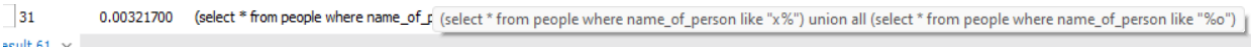
Query

```
(select * from people where name_of_person like "x%") union all (select * from people where name_of_person like "%o");
explain (select * from people where name_of_person like "x%") union all (select * from people where name_of_person like "%o");
```

Number of scans

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	PRIMARY	people	NULL	ALL	NULL	NULL	NULL	NULL	1210	11.11	Using where
2	UNION	people	NULL	ALL	NULL	NULL	NULL	NULL	1210	11.11	Using where

Execution time



->The UNION operator combines the results and performs a distinct sort whereas UNION ALL combines both queries or tables and their result set. The Union All is not concerned about sorting data to delete records. Thus it is faster than UNION also.

->Generally we use union all when we are not bothered about the duplicates but when we are concerned about duplicates, then union may not work as optimizer in most of the cases it even works for increasing the execution time. Even union all also may not optimize because we see the same number of scans. So the below method gives better performance.

<u>Optimization With the help of indexing and Union that works better than the above two:</u>

-> The following method of indexing avoids full table scans and hence, reduces the execution time considerably.

Alter table people add fulltext (name_of_person);

Execution time comparison :

```
mysql> show profiles;
+----------+------------+-----------------------------------------------------------------------------------------------------------+
| Query_ID | Duration   | Query                                                                                                     |
+----------+------------+-----------------------------------------------------------------------------------------------------------+
|        1 | 0.00347025 | select * from people where match (name_of_person) against ("x* union g*" in boolean mode)                 |
```

```
|        3 | 0.03268550 | select * from people where name_of_person like "x%" or name_of_person like "g%"                           |
+----------+------------+-----------------------------------------------------------------------------------------------------------+
```

Number of Scans

```
mysql> explain select * from people where match (name_of_person) against ("x* union g*" in boolean mode);
+----+-------------+--------+------------+----------+---------------+---------------+---------+-------+------+----------+-------------------------------+
| id | select_type | table  | partitions | type     | possible_keys | key           | key_len | ref   | rows | filtered | Extra                         |
+----+-------------+--------+------------+----------+---------------+---------------+---------+-------+------+----------+-------------------------------+
|  1 | SIMPLE      | people | NULL       | fulltext | name_of_person | name_of_person | 0       | const |    1 |   100.00 | Using where; Ft_hints: no_ranking |
+----+-------------+--------+------------+----------+---------------+---------------+---------+-------+------+----------+-------------------------------+
1 row in set, 1 warning (0.04 sec)

mysql> explain select * from people where name_of_person like "x%" or name_of_person like "g%";
+----+-------------+--------+------------+------+---------------+------+---------+------+------+----------+-------------+
| id | select_type | table  | partitions | type | possible_keys | key  | key_len | ref  | rows | filtered | Extra       |
+----+-------------+--------+------------+------+---------------+------+---------+------+------+----------+-------------+
|  1 | SIMPLE      | people | NULL       | ALL  | name_of_person | NULL | NULL    | NULL | 1210 |    20.99 | Using where |
+----+-------------+--------+------------+------+---------------+------+---------+------+------+----------+-------------+
1 row in set, 1 warning (0.00 sec)
```

Explain analyze for the better understanding of execution time

```
mysql> explain analyze select * from people where name_of_person like "x%" or name_of_person like "g%";
+-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------+
| EXPLAIN                                                                                                                                                                                                                                 |
+-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------+
| -> Filter: ((people.name_of_person like 'x%') or (people.name_of_person like 'g%'))  (cost=122.50 rows=254) (actual time=0.050..2.148 rows=337 loops=1)
    -> Table scan on people  (cost=122.50 rows=1210) (actual time=0.032..1.627 rows=1210 loops=1)
 |
+-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------+
1 row in set (0.00 sec)
```

```
mysql> explain analyze select * from people where match (name_of_person) against ("x* union g*" in boolean mode);
+-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------+
| EXPLAIN                                                                                                                                                                                                                                 |
+-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------+
| -> Filter: (match people.name_of_person against ('x* union g*' in boolean mode))  (cost=0.35 rows=1) (actual time=0.169..1.272 rows=337 loops=1)
    -> Full-text index search on people using name_of_person (name_of_person='x* union g*')  (cost=0.35 rows=1) (actual time=0.165..1.192 rows=337 loops=1)
 |
+-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------+
```

->Here first we add fulltext index to the name_of_person and then we use fulltext to match against “g*” or “x*” using union function in the match statement which makes the query optimized.

We observe a good change in the execution time with the fulltext index that is from 0.032 to 0.0034. That is we observe the execution time has become 1/10 of the execution time without optimization. In the above union and union all we observe a very slight decrease

<u>Question 2</u>

<u>Before Optimization:</u>

<u>Query:</u>

```
select * from soldiers where battalion_name like "east%";
explain select * from soldiers where battalion_name like "east%";
show profiles;
```

<u>Number of Scans:</u>

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	soldiers	NULL	ALL	battalion_name	NULL	NULL	NULL	1000	11.11	Using where

<u>Execution time:</u>

82	0.00154825	select * from soldiers where

select * from soldiers where battalion_name like "east%" LIMIT 0, 5000

Result 100 ×

<u>Explain analyze:</u>

```
mysql> explain analyze select * from soldiers where battalion_name like "east%";
+--------------------------------------------------------------------------------------------------------------
---------+
| EXPLAIN
         |
+--------------------------------------------------------------------------------------------------------------
---------+
| -> Filter: (soldiers.battalion_name like 'east%')  (cost=101.50 rows=111) (actual time=0.019..0.631 rows=13 loops=1)
    -> Table scan on soldiers  (cost=101.50 rows=1000) (actual time=0.017..0.532 rows=1000 loops=1)
 |
+--------------------------------------------------------------------------------------------------------------
---------+
```

<u>After Optimization:</u>
<u>Query</u>

```
alter table soldiers add fulltext (battalion_name);
select * from soldiers where match(battalion_name) against ('east*' in boolean mode);
explain select * from soldiers where match(battalion_name) against ('east*' in boolean mode);
show profiles;
```

<u>Number of Scans:</u>

```
+----+-------------+----------+------------+----------+----------------+----------------+---------+-------+------+----------+-----------------------------------+
| id | select_type | table    | partitions | type     | possible_keys  | key            | key_len | ref   | rows | filtered | Extra                             |
+----+-------------+----------+------------+----------+----------------+----------------+---------+-------+------+----------+-----------------------------------+
|  1 | SIMPLE      | soldiers | NULL       | fulltext | battalion_name | battalion_name | 0       | const |    1 |   100.00 | Using where; Ft_hints: no_ranking |
+----+-------------+----------+------------+----------+----------------+----------------+---------+-------+------+----------+-----------------------------------+
1 row in set, 1 warning (0.00 sec)
```

Execution time

```
+----------+------------+-------------------------------------------------------------------------------------------+
| Query_ID | Duration   | Query                                                                                     |
+----------+------------+-------------------------------------------------------------------------------------------+
|        1 | 0.00047300 | select * from soldiers where match(battalion_name) against ('east*' in boolean mode) |
+----------+------------+-------------------------------------------------------------------------------------------+
1 row in set, 1 warning (0.00 sec)
```

Explain analyze

```
mysql> explain analyze select * from soldiers where match(battalion_name) against ('east*' in boolean mode);
+--------------------------------------------------------------------------------------------------------------
| EXPLAIN
                                                                                                             |
+--------------------------------------------------------------------------------------------------------------
| -> Filter: (match soldiers.battalion_name against ('east*' in boolean mode))  (cost=0.35 rows=1) (actual time=1.401..1.756 rows=13 loops=1)
    -> Full-text index search on soldiers using battalion_name (battalion_name='east*')  (cost=0.35 rows=1) (actual time=1.228..1.579 rows=13 loops=1)
 |
+--------------------------------------------------------------------------------------------------------------
```

Similar to the above to avoid full scan over the entire table and reduce the execution time we used full text index on battalion_name. By using fulltext, we have reduced the scan set from 1000 to 1. This implies that the set only contains the values of our search condition. Thus it reduces the execution time significantly. The execution has been decreased from 0.001548 to 0.00047300 and this helps at a large extent when we are having a large number of tuples.

Question 3

On table Battalions
Query:

ALTER TABLE battalions MODIFY battalion_id tinyint;

The storage space of int is 4 bytes whereas that of tinyint is 1 byte. The range to which we can use it is surely reduced, however the aspects like data length and storage size are reduced by a factor of four. However, with this data type, we cannot use values beyond 0 to 255. Thus it was an ideal change for battalion_id which would not exceed from the given value.

Question 4

Before Optimization

Query:

```
select * from people where date_of_birth="2002/6/28";
explain select * from people where date_of_birth="2002/6/28";
show profiles;
```

Number-of-scans:

```
mysql> explain select * from people where date_of_birth="2002/6/28";
+----+-------------+--------+------------+------+---------------+------+---------+------+------+----------+-------------+
| id | select_type | table  | partitions | type | possible_keys | key  | key_len | ref  | rows | filtered | Extra       |
+----+-------------+--------+------------+------+---------------+------+---------+------+------+----------+-------------+
|  1 | SIMPLE      | people | NULL       | ALL  | NULL          | NULL | NULL    | NULL | 1210 |    10.00 | Using where |
+----+-------------+--------+------------+------+---------------+------+---------+------+------+----------+-------------+
1 row in set, 1 warning (0.00 sec)
```

Execution time

```
|        2 | 0.00646600 | select * from people where date_of_birth="2002/6/28"
```

After Optimization:

Number-of-scans:

```
mysql> explain select * from people where date_of_birth="2002/6/28";
+----+-------------+--------+------------+------+---------------+------+---------+------+------+----------+-------------+
| id | select_type | table  | partitions | type | possible_keys | key  | key_len | ref  | rows | filtered | Extra       |
+----+-------------+--------+------------+------+---------------+------+---------+------+------+----------+-------------+
|  1 | SIMPLE      | people | NULL       | ALL  | NULL          | NULL | NULL    | NULL | 1210 |    10.00 | Using where |
+----+-------------+--------+------------+------+---------------+------+---------+------+------+----------+-------------+
1 row in set, 1 warning (0.00 sec)

mysql>
```

Execution time:

```
|        4 | 0.00377525 | select * from people where date_of_birth="2002/6/28"
+----------+------------+-----------------------------------------------------------------
```

In this question, we simply changed the datatype of date_of_birth from varchar to date. Thus the query statement is not affected. By doing this, we have reduced the size of the data because we are no longer obligated to store length information in varchar. Storing the date_of_birth in date

datatype helps in avoiding the complex functions later on which are needed for changing the columns into date format which increases the execution time. So to avoid these complexities it's easier to check the date format and perform the comparisons. In the above example we observe there is a decrease of execution time from 0.006 to 0.003 due to the change of data type into date.

Question 5

Queries

```
select count(ifnull(qty_remaining, 0)) from weapons;
select count(qty_remaining) from weapons;


explain select count(ifnull(qty_remaining, 0)) from weapons;
explain select count(qty_remaining) from weapons;
```

Number of scans

```
mysql> explain select count(ifnull(qty_remaining, 0)) from weapons;explain select count(qty_remaining) from weapons;
+----+-------------+---------+------------+------+---------------+------+---------+------+------+----------+-------+
| id | select_type | table   | partitions | type | possible_keys | key  | key_len | ref  | rows | filtered | Extra |
+----+-------------+---------+------------+------+---------------+------+---------+------+------+----------+-------+
|  1 | SIMPLE      | weapons | NULL       | ALL  | NULL          | NULL | NULL    | NULL |  104 |   100.00 | NULL  |
+----+-------------+---------+------------+------+---------------+------+---------+------+------+----------+-------+
1 row in set, 1 warning (0.00 sec)

+----+-------------+---------+------------+------+---------------+------+---------+------+------+----------+-------+
| id | select_type | table   | partitions | type | possible_keys | key  | key_len | ref  | rows | filtered | Extra |
+----+-------------+---------+------------+------+---------------+------+---------+------+------+----------+-------+
|  1 | SIMPLE      | weapons | NULL       | ALL  | NULL          | NULL | NULL    | NULL |  104 |   100.00 | NULL  |
+----+-------------+---------+------------+------+---------------+------+---------+------+------+----------+-------+
1 row in set, 1 warning (0.00 sec)
```

Execution times

```
+----------+-------------+-----------------------------------------------------------+
| Query_ID | Duration    | Query                                                     |
+----------+-------------+-----------------------------------------------------------+
|        1 | 0.00357900 | select count(ifnull(qty_remaining, 0)) from weapons |
|        2 | 0.00019225 | select count(qty_remaining) from weapons            |
+----------+-------------+-----------------------------------------------------------+
2 rows in set, 1 warning (0.00 sec)
```

Explain Analyze before and after optimization

```
mysql> explain analyze select count(qty_remaining) from weapons;
+-----------------------------------------------------------------------------------------------------------------------------------------------------------+
| EXPLAIN                                                                                                                                                   |
+-----------------------------------------------------------------------------------------------------------------------------------------------------------+
| -> Aggregate: count(weapons.qty_remaining)  (cost=21.05 rows=104) (actual time=1.234..1.234 rows=1 loops=1)
    -> Table scan on weapons  (cost=10.65 rows=104) (actual time=1.125..1.218 rows=104 loops=1)
 |
+-----------------------------------------------------------------------------------------------------------------------------------------------------------+
1 row in set (0.01 sec)

mysql> explain analyze select count(ifnull(qty_remaining, 0)) from weapons;
+-------------------------------------------------------------------------+
| EXPLAIN                                                                 |
+-------------------------------------------------------------------------+
| -> Count rows in weapons  (actual time=4.100..4.100 rows=1 loops=1)
 |
+-------------------------------------------------------------------------+
```

Null explains the absence of values in the columns. When we perform operations then there are chances where the result might give erroneous results in those cases. To treat such cases we generally use ifnull() which says what to return from that entry when performing the query. In the above example we used ifnull() to return 0 incase of Null value. So that we can count the not null values in the column. From the explain analyze query we can observe that complexity has been decreased.

Queries

Before optimization

```
select sum(qty_remaining) from weapons;
```

After optimization

```
select sum(ifnull(qty_remaining, 0)) from weapons;
```

Execution Time

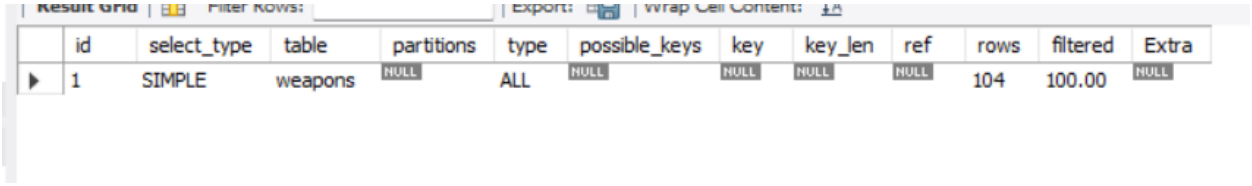
224	0.00041150	select sum(qty_remaining) from weapons LIMIT ...
225	0.00037425	select sum(ifnull(qty_remaining, 0)) from weapo...

Number of scans

Before optimization

	id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
▶	1	SIMPLE	weapons	NULL	ALL	NULL	NULL	NULL	NULL	104	100.00	NULL

After optimization



	id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
▶	1	SIMPLE	weapons	NULL	ALL	NULL	NULL	NULL	NULL	104	100.00	NULL

One more example for the ifnull() case is while performing the sum operation. If there are null values then we need to treat them separately and perform the query. Using ifnull() helps in performing the query by replacing the null value with any value we wish without changing the actual table. This helps in reducing the if else clauses that we write to take for null values. From above example we observe a decrease in the execution time from 0.0004 to 0.0003

Question 6

MySQL query cache is a powerful method to speed up the performance of the database. It is most useful when there are several **select** queries in the operation of our database. What this query cache does, is that it stores the text of the SQL query along with the results that were sent back. So, the next time the server receives a similar query, the server retrieves the results from the cache instead of again executing the query.
Another benefit of caches is that, query cache is shared among sessions. So, a query cache generated by one client can be used to serve the request of another client.

As mentioned earlier, query caches are useful when:

- The tables are not altered much
- There are many select operations
- The queries are identical
- Queries must match byte-for-byte
- There are no non-deterministic features. Non-deterministic features will not let the query get cached (including temporary tables, user variables, RAND(), NOW() and UDFs.)
- Underlying tables are not modified. When they are modified, it results in the stored cache getting deleted for that table.

The best use of query caches is when the operations are extremely large and are read-only, examining millions of rows.

The statement SHOW VARIABLES LIKE "%query_cache%" is used to know whether the query cache is enabled or no, and to see what parameters are set.
From the command SHOW VARIABLES LIKE 'have_query_cache'; we can know whether the query cache is available or not.

The output comes to be something like this (taken from here)

```
+------------------+-------+
| Variable_name    | Value |
+------------------+-------+
| have_query_cache | YES   |
+------------------+-------+
```

When the “set_query_type” variable equals 0 or OFF, it means query caching is disabled. To enable it, we can either

- Set set_query_type to 1 - to cache non-select operations
- Set set_query_type to 2 - to cache select operations

However, one of the major drawbacks of query caching is that it is known to not scale with high-throughput workloads on multi-core machines. MySQL query cache was meant to improve performance. But since it had severe scalability issues, it became a bottleneck.
Even if the scalability was to be improved, the disadvantage of using query caches is that only the queries that access the caches will improve. It is improbable that the predictability of the performance improves.

Also the research conducted by Jiamin Huang, Barzan Mozafari, Grant Schoenebeck, Thomas F. Wenisch at the University of Michigan, Ann Arbor came to a similar conclusion. Hence, it was decided that support for query caches would be discontinued from MySQL version 8 onwards.

Question 7

Query:

We performed cartesian product operation on the tables of soldiers and people. Let's say, we need the count of people who are soldiers. So, a basic, unoptimized query is:

Before optimization

```
> select count(people.id), count(s_id) from people, soldiers where people.id=soldiers.s_id;
--------------+--------------+
```

After optimization

An optimized query using joins is:

```
select count(people.id),count(s_id) from people join soldiers where people.id=soldiers.s_id;
--------------+--------------+
```

Number of scans

```
mysql> explain select count(people.id), count(s_id) from people, soldiers where people.id=soldiers.s_id;
+----+-------------+----------+------------+------+---------------+------+---------+------+------+----------+----------------------------------------------+
| id | select_type | table    | partitions | type | possible_keys | key  | key_len | ref  | rows | filtered | Extra                                        |
+----+-------------+----------+------------+------+---------------+------+---------+------+------+----------+----------------------------------------------+
|  1 | SIMPLE      | soldiers | NULL       | ALL  | NULL          | NULL | NULL    | NULL | 1000 |   100.00 | NULL                                         |
|  1 | SIMPLE      | people   | NULL       | ALL  | NULL          | NULL | NULL    | NULL | 1210 |    10.00 | Using where; Using join buffer (hash join) |
+----+-------------+----------+------------+------+---------------+------+---------+------+------+----------+----------------------------------------------+
2 rows in set, 1 warning (0.03 sec)

mysql> explain select count(people.id),count(s_id) from people join soldiers where people.id=soldiers.s_id;
+----+-------------+----------+------------+------+---------------+------+---------+------+------+----------+----------------------------------------------+
| id | select_type | table    | partitions | type | possible_keys | key  | key_len | ref  | rows | filtered | Extra                                        |
+----+-------------+----------+------------+------+---------------+------+---------+------+------+----------+----------------------------------------------+
|  1 | SIMPLE      | soldiers | NULL       | ALL  | NULL          | NULL | NULL    | NULL | 1000 |   100.00 | NULL                                         |
|  1 | SIMPLE      | people   | NULL       | ALL  | NULL          | NULL | NULL    | NULL | 1210 |    10.00 | Using where; Using join buffer (hash join) |
+----+-------------+----------+------------+------+---------------+------+---------+------+------+----------+----------------------------------------------+
2 rows in set, 1 warning (0.00 sec)
```

Execution time comparison

As we can see, there is a decrease in the execution time where we use join operation.

```
|        3 | 0.00371600 | select count(people.id), count(s_id) from people, soldiers where people.id=soldiers.s_id    |
|        4 | 0.00369800 | select count(people.id),count(s_id) from people join soldiers where people.id=soldiers.s_id |
+----------+------------+-----------------------------------------------------------------------------------------------+
4 rows in set, 1 warning (0.03 sec)
```

Explain analyze before optimization

```
-> Aggregate: count(people.id), count(soldiers.s_id)  (cost=133203.03 rows=121000) (actual time=3.759..3.759 rows=1 loops=1)
    -> Inner hash join (people.id = soldiers.s_id)  (cost=121103.03 rows=121000) (actual time=1.823..3.666 rows=500 loops=1)
        -> Table scan on people  (cost=0.01 rows=1210) (actual time=0.019..1.503 rows=1210 loops=1)
        -> Hash
            -> Table scan on soldiers  (cost=101.50 rows=1000) (actual time=0.070..1.323 rows=1000 loops=1)
```

Explain analyze after optimization

```
-> Aggregate: count(people.id), count(soldiers.s_id)  (cost=133203.03 rows=121000) (actual time=3.511..3.511 rows=1 loops=1)
    -> Inner hash join (people.id = soldiers.s_id)  (cost=121103.03 rows=121000) (actual time=1.651..3.433 rows=500 loops=1)
        -> Table scan on people  (cost=0.01 rows=1210) (actual time=0.008..1.452 rows=1210 loops=1)
        -> Hash
            -> Table scan on soldiers  (cost=101.50 rows=1000) (actual time=0.042..1.257 rows=1000 loops=1)
```

The reason why join is better, because in the case of cartesian product, we are performing the condition (where) operation on MxN tuples, whereas in the case of theta join, we are actually dealing with the joined result. So, the search condition is applied on a smaller table.

We can further optimize the join operations, by using the same logic.
It is beneficial to perform an operation on a reduced table, rather than performing an operation and then reducing the table. Hence, before applying the join operation we need to first do project over the condition and then perform the join operation. This reduces the tuple size while joining the table and hence helps in decreasing the execution time.

This is what we have done in the below :

Queries:

```
-- question 7
select * from people join soldiers where people.id=soldiers.s_id and name_of_person like "a%";
select * from soldiers join (select * from people where name_of_person like "a%") as t where t.id=soldiers.s_id;
show profiles;
```

```
explain select * from people join soldiers where people.id=soldiers.s_id and name_of_person like "a%";
explain select * from soldiers join (select * from people where name_of_person like "a%") as t where t.id=soldiers.s_id;
```

Number of scans before optimization

	id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
▶	1	SIMPLE	people	NULL	ALL	NULL	NULL	NULL	NULL	1210	11.11	Using where
	1	SIMPLE	soldiers	NULL	ALL	NULL	NULL	NULL	NULL	1000	10.00	Using where; Using join buffer (hash join)

Number of scans after optimization

	id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
▸	1	SIMPLE	people	NULL	ALL	NULL	NULL	NULL	NULL	1210	11.11	Using where
	1	SIMPLE	soldiers	NULL	ALL	NULL	NULL	NULL	NULL	1000	10.00	Using where; Using join buffer (hash join)

Execution time comparisons

```
| Query_ID | Duration   | Query
|        1 | 0.01089700 | select * from people join soldiers where people.id=soldiers.s_id and name_of_person like "a%"
|        2 | 0.00322500 | select * from soldiers join (select * from people where name_of_person like "a%") as t where t.id=soldiers.s_id
```

Explain Analyze before optimization

```
mysql> explain analyze select * from people join soldiers where people.id=soldiers.s_id and name_of_person like "a%";
| EXPLAIN
| -> Inner hash join (soldiers.s_id = people.id)  (cost=13566.98 rows=13443) (actual time=2.509..4.343 rows=334 loops=1)
    -> Table scan on soldiers  (cost=0.09 rows=1000) (actual time=0.872..2.526 rows=1000 loops=1)
    -> Hash
        -> Filter: (people.name_of_person like 'a%')  (cost=122.25 rows=134) (actual time=0.100..1.308 rows=734 loops=1)
            -> Table scan on people  (cost=122.25 rows=1210) (actual time=0.095..1.076 rows=1210 loops=1)
```

Explain analyze after optimization

```
mysql> explain analyze select * from soldiers join (select * from people where name_of_person like "a%") as t where t.id=soldiers.s_id;
| EXPLAIN
| -> Inner hash join (soldiers.s_id = people.id)  (cost=13566.98 rows=13443) (actual time=1.578..2.432 rows=334 loops=1)
    -> Table scan on soldiers  (cost=0.09 rows=1000) (actual time=0.008..0.698 rows=1000 loops=1)
    -> Hash
        -> Filter: (people.name_of_person like 'a%')  (cost=122.25 rows=134) (actual time=0.025..1.107 rows=734 loops=1)
            -> Table scan on people  (cost=122.25 rows=1210) (actual time=0.023..0.882 rows=1210 loops=1)
```

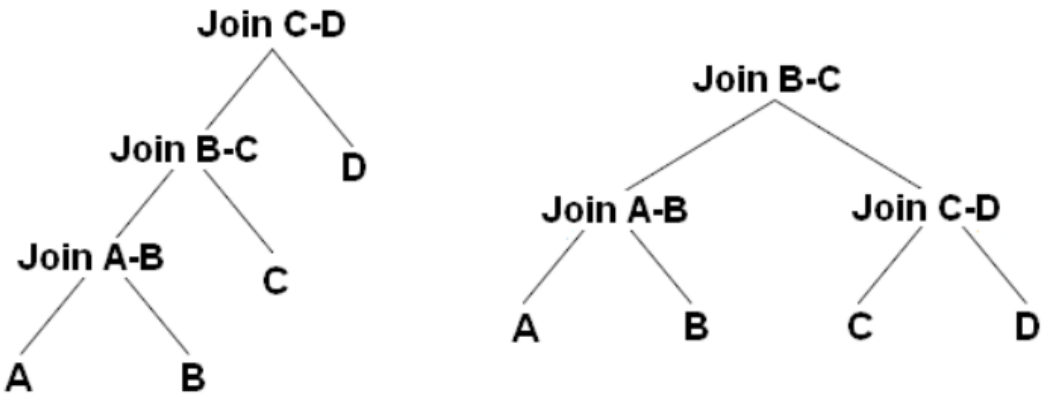
Drawbacks of multiple joins

On investigating the SQL queries when multiple joins are performed, one finds here are certain things the server has to deal with:

- The names of the objects get changed a lot
- A large number of unnecessary parentheses are used
- The most problematic thing is that the same table is used multiple times
- SQL spends too many resources while planning to execute a query with many joins. For example, the query may run for 50-150 ms, and plan creation may take up to 2.5 ms.

All of this occurs, because in any database management system, the main optimization problem is the method of joining tables. Also, the sequence of joining is also of importance.

One noteworthy point is that, possible number of joins grows exponentially, not linearly. Example, there are only 2 ways to join 2 tables, but there are 12 ways to join 3 tables. Different sequences result in different costs, and the server needs to decide the best method. When number of tables increase (multi table joins), this becomes a very resource intensive tasks.



Different ways to join tables [1]

Tables	Left-deep trees	Bushy trees
1	1	1
2	2	2
3	6	12
4	24	120
5	120	1,680
6	720	30,240
7	5,040	665,280
8	40,320	17,297,280
9	362,880	518,918,400
10	3,628,800	17,643,225,600
11	39,916,800	670,442,572,800
12	479,001,600	28,158,588,057,600

Increase of number of ways as number of tables increases [2]

These are the drawbacks of using multiple joins.

Question 8

Before optimization

Query

```
select * from weapons as r join
(select id,s_id from battalions as p
join (select * from people join soldiers where people.id=soldiers.s_id) as t where p.battalion_name like "%2") as temp
where r.weapon_id>10;
```

Number of scans

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	p	NULL	ALL	NULL	NULL	NULL	NULL	200	11.11	Using where
1	SIMPLE	r	NULL	ALL	NULL	NULL	NULL	NULL	104	33.33	Using where; Using join buffer (hash join)
1	SIMPLE	soldiers	NULL	ALL	NULL	NULL	NULL	NULL	1000	100.00	Using join buffer (hash join)
1	SIMPLE	people	NULL	ALL	NULL	NULL	NULL	NULL	1210	10.00	Using where; Using join buffer (hash join)

Execution time

270 0.00557675 select * from weapons as r join (select id,s_id fr...

:sult 213 ×

select * from weapons as r join
(select id,s_id from battalions as p

Explain Analyze

```
-----------------------------------------------+
| EXPLAIN

+----------------------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------------------------
-----------------------------------------------+
| -> Inner hash join (people.id = soldiers.s_id)  (cost=31088860.70 rows=31062285) (actual time=562.504..1117.203 rows=930000 loops=1)
    -> Table scan on people  (cost=0.00 rows=1210) (actual time=0.017..0.560 rows=1210 loops=1)
    -> Hash
        -> Inner hash join (no condition)  (cost=25778.25 rows=256713) (actual time=2.169..153.410 rows=1860000 loops=1)
            -> Table scan on soldiers  (cost=0.13 rows=1000) (actual time=0.006..2.336 rows=1000 loops=1)
            -> Hash
                -> Inner hash join (no condition)  (cost=104.51 rows=257) (actual time=1.803..1.969 rows=1860 loops=1)
                    -> Filter: (r.weapon_id > 10)  (cost=0.48 rows=35) (actual time=0.016..0.067 rows=93 loops=1)
                        -> Table scan on r  (cost=0.48 rows=104) (actual time=0.008..0.057 rows=104 loops=1)
                    -> Hash
                        -> Filter: (p.battalion_name like '%2')  (cost=20.25 rows=22) (actual time=1.627..1.768 rows=20 loops=1)
                            -> Table scan on p  (cost=20.25 rows=200) (actual time=1.616..1.710 rows=200 loops=1)
 |
+----------------------------------------------------------------------------------------------------------------------------
```

After Optimization

Query

```
select * from (select * from weapons where weapon_id>10) as r join
(select id,s_id from (select * from battalions  where battalion_name like "%2")as p
join (select * from people join soldiers where people.id=soldiers.s_id) as t) as temp ;
```

Number of scans

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

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	battalions	NULL	ALL	NULL	NULL	NULL	NULL	200	11.11	Using where
1	SIMPLE	weapons	NULL	ALL	NULL	NULL	NULL	NULL	104	33.33	Using where; Using join buffer (hash join)
1	SIMPLE	soldiers	NULL	ALL	NULL	NULL	NULL	NULL	1000	100.00	Using join buffer (hash join)
1	SIMPLE	people	NULL	ALL	NULL	NULL	NULL	NULL	1210	10.00	Using where; Using join buffer (hash join)

Execution time

274	0.00484550	select * from (select * from weapons wl	select * from (select * from weapons where weapon_id>10) as r join (select id,s_id from (select * from battalions where battalion_name like "%2")a join (select * from people join soldiers where people.id=soldiers.s_id) as t) as te

ult 218 ×

Explain analyze

```
-> Inner hash join (people.id = soldiers.s_id)  (cost=31088860.70 rows=31062285) (actual time=549.004..1070.320 rows=930000 loops=1)
    -> Table scan on people  (cost=0.00 rows=1210) (actual time=0.014..0.539 rows=1210 loops=1)
    -> Hash
        -> Inner hash join (no condition)  (cost=25778.25 rows=256713) (actual time=0.545..152.036 rows=1860000 loops=1)
            -> Table scan on soldiers  (cost=0.13 rows=1000) (actual time=0.005..2.241 rows=1000 loops=1)
            -> Hash
                -> Inner hash join (no condition)  (cost=104.51 rows=257) (actual time=0.184..0.348 rows=1860 loops=1)
                    -> Filter: (weapons.weapon_id > 10)  (cost=0.48 rows=35) (actual time=0.010..0.064 rows=93 loops=1)
                        -> Table scan on weapons  (cost=0.48 rows=104) (actual time=0.003..0.053 rows=104 loops=1)
                    -> Hash
                        -> Filter: (battalions.battalion_name like '%2')  (cost=20.25 rows=22) (actual time=0.038..0.160 rows=20 loops=1)
                            -> Table scan on battalions  (cost=20.25 rows=200) (actual time=0.029..0.120 rows=200 loops=1)
```

We observe after the query optimization execution time has been decreased from 0.0055 to 0.0048. Number of scans are the same in both the queries because here we are dealing with the joins. The optimization here is instead of performing the join operation on the entire table we join the shorter table by first selecting the required tuples. We can observe the reduction of the execution time from the explain analyze from the nested sub queries.

Contributions

Group G1

Lakshmi = 30%
Xhitij = 30%
Bhoomika = 30%
Shreyshi = 10%

Group G2

Paras = 33.33%
Shridhar = 33.33%
Likhita = 33.33%

Group G1 and G2

Group G1 = 50%
Group G2 = 50%

References

1. sql - Problems with multiple joins and a sum - Stack Overflow
2. (106) What are the disadvantages of using joins in SQL? - Quora
3. Why Multiple JOINs are bad for Query or Do Not Get in the Way of Optimizer - {coding}Sight (codingsight.com)
4. The MySQL Query Cache: How it works and workload impacts (percona.com)
5. Performance Tuning in MySQL with Query Caching (logicalread.com)
6. Understanding MySQL Query Cache – The Geek Diary
7. MySQL Explain - javatpoint
8. MySQL :: MySQL 8.0 Reference Manual :: 8.2.1.16 ORDER BY Optimization
9. Optimize MySQL Query Performance With Joins (logicalread.com)