CS 432: Databases
Assignment 8: Query Optimization

TASK 1

Table: CUSTOMER
Columns: first_name & last_name

When we want to run queries using the comparison operator 'OR' on the columns (first_name & last_name) in a CUSTOMER Table. Here, if we use the 'OR' operator excessively in the WHERE clause, then there are chances that these query optimizers may incorrectly choose a full table scan to retrieve a record.

Hence, if we have an index that can optimize one side of the query and a different index to optimize the other side, then a UNION clause can make the query run faster.

We are running the below queries with the columns 'first_name' and 'last_name.'

```
SELECT * FROM customer_details_table
WHERE first_name LIKE 'Mar%' OR last_name LIKE 'Sta%';

SELECT * FROM customer_details_table
WHERE first_name LIKE 'Mar%'
UNION
SELECT * FROM customer_details_table WHERE last_name LIKE 'Sta%';
```

We found that the first query (having a LIKE statement) run far much slower than the second query which uses UNION clause merge the result of two separate queries that take advantage of the indexes.

RESULT

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	PRIMARY	customer_details_table	NULL	ALL	NULL	NULL	NULL	NULL	9706	11.11	Using where
2	UNION	customer_details_table	NULL	range	last_name_idx	last_name_idx	83	NULL	57	100.00	Using index condition

TASK 2

From the queries run in TASK 1, we found that the first query (having a LIKE statement) runs far much slower than the second query, which uses the UNION clause to merge the result of two separate queries that take advantage of the indexes.

RESULT

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	PRIMARY	customer_details_table	NULL	ALL	NULL	NULL	NULL	NULL	9706	11.11	Using where
2	UNION	customer_details_table	NULL	range	last_name_idx	last_name_idx	83	NULL	57	100.00	Using index condition

Total no. of rows in table	9886
No. of scan in rows, before optimization	9706
No. of scan in rows, after optimization	57

TASK 3

In this query optimization, we changed the datatype of the `manager_id` column of table `branch_name`. We can do this optimization because the range in which `manager_id` column values lies between 0 and 128. We can use `TINYINT` instead of `INT` as the data type for column `manager_id`. This modification makes sure that we use less data on disk as `TINYINT` datatype takes 1-byte storage space compared to `INT` datatype, which takes 4 bytes. Also, this gives rise to better performance.

Schema with `manager_id` datatype as `INT` (table name: **`branch_table`**)

Field	Type	Null	Key	Default	Extra
branch_name	text	YES		NULL	
branch_city	text	YES		NULL	
assets	text	YES		NULL	
manager_id	int	YES		NULL	

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	branch_table	NULL	ALL	NULL	NULL	NULL	NULL	20	100.00	NULL

Schema with `manager_id` datatype as `TINYINT` (table name: **`branch_table_tiny`**)

Field	Type	Null	Key	Default	Extra
branch_name	text	YES		NULL	
branch_city	text	YES		NULL	
assets	text	YES		NULL	
manager_id	tinyint	YES		NULL	

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	branch_table_tiny	NULL	ALL	NULL	NULL	NULL	NULL	20	100.00	NULL

TASK 4

Our database Banking System contains table `employee_details` which has a column `start_date`. This column contains the date on which the employee joined the bank.

This column was initially implemented in `text` data type. We changed it to `date` type.

Searching can be optimized for a specific date. This is achieved by first creating an index on the primary key of that table `employee_id` (also implemented in A7). This index can be of two types. We implemented a unique BTree index.

```
select * from employee_details_table_ori
where start_date = 30-11-2016 ;
```

For query optimization, we first strip column for temporary table according to specific query

YEAR() returns a year from the date (e.g., 2012)
MONTH() returns month from the date (1 - 12)
DAY() returns the day of the month (1 - 31)

And then search in it asc/dsc in that temporary table.

```sql
explain
select * from employee_details_table_by_tree
where start_date = '30-11-2016' order by month('30-11-2016')  ;

explain
select * from employee_details_table_by_tree
where start_date = '30-11-2016' order by year('30-11-2016')  ;
```

More optimization can be achieved by using `order by` on each subpart of date. For specific query,

Without optimization,

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	employee_details_table	NULL	ALL	NULL	NULL	NULL	NULL	500	10.00	Using where

With optimization,

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	employee_details_table_ori	NULL	ALL	NULL	NULL	NULL	NULL	460	10.00	Using where

TASK 5

Table: `CUSTOMER`
Column: `first_name`

JUSTIFICATION

Around 1/5th of the rows in the first_name column are changed to `NULL`. Imported the `CUSTOMER` table data into a new table `customer_null` and then made the values in the first name column null, having customers id's from 300000 to 600000. This can be done using the below query.

```sql
UPDATE customer_null
SET first_name = NULL
WHERE customer_id < 600000
AND customer_id > 400000;
```

In the newly updated table, we get the count values by removing the null values in the row, and this can be seen after running the query

```
select count(first_name) from customer_null;
```

The computation time is reduced after adding the null values and considering them as zero, and this can be seen from the below image. This is because supposing the data is stored in a B+ tree, the number of nodes in case of no null values is higher than null values and is intuitive. Since the number of nodes is decreased, the search/query time is reduced.

```
mysql> show profiles;
+----------+------------+---------------------------------------------+
| Query_ID | Duration   | Query                                       |
+----------+------------+---------------------------------------------+
|        1 | 0.00366250 | select count(*) from customer               |
|        2 | 0.00214400 | select count(*) from customer_null          |
|        3 | 0.00782000 | select count(first_name) from customer      |
|        4 | 0.00601025 | select count(first_name) from customer_null |
+----------+------------+---------------------------------------------+
4 rows in set, 1 warning (0.00 sec)
```

Here, we can clearly explain the queries.

```
mysql> explain select count(first_name) from customer;
+----+-------------+----------+------------+------+---------------+------+---------+------+------+----------+-------+
| id | select_type | table    | partitions | type | possible_keys | key  | key_len | ref  | rows | filtered | Extra |
+----+-------------+----------+------------+------+---------------+------+---------+------+------+----------+-------+
|  1 | SIMPLE      | customer | NULL       | ALL  | NULL          | NULL | NULL    | NULL | 9834 |   100.00 | NULL  |
+----+-------------+----------+------------+------+---------------+------+---------+------+------+----------+-------+
1 row in set, 1 warning (0.00 sec)
```

```
mysql> explain select count(first_name) from customer_null;
+----+-------------+---------------+------------+------+---------------+------+---------+------+------+----------+-------+
| id | select_type | table         | partitions | type | possible_keys | key  | key_len | ref  | rows | filtered | Extra |
+----+-------------+---------------+------------+------+---------------+------+---------+------+------+----------+-------+
|  1 | SIMPLE      | customer_null | NULL       | ALL  | NULL          | NULL | NULL    | NULL | 9959 |   100.00 | NULL  |
+----+-------------+---------------+------------+------+---------------+------+---------+------+------+----------+-------+
1 row in set, 1 warning (0.00 sec)
```

TASK 6

OBSERVATION

In the tables having large data, it takes a lot of time to retrieve the data. With this given, if a client wants to request the same query that he just requested before, it will take the same time as before, which will be frustrating. In such cases, we use query caching on the database.

JUSTIFICATION

We perform the query caching on the customer table, account table, and loan table as these tables have large data. The query caching helps us to retrieve the data faster from the database. This can be achieved by storing the part of select statements whenever the client retrieves the data the first time. If the client requests the same query again, it will take less time than before. The query_cache_size is not too large because it hampers the performance when more and more data is added to the database. So, here the idea is to increase the size in small increments.

```
SET query_cache_size = 10M
SET query_cache_type = 1
SET profiling  = 1

select * from CUSTOMER
select * from CUSTOMER

select * from ACCOUNT
select * from ACCOUNT

select * from LOAN
select * from LOAN

SHOW profiles;
```

TASK 7

```
SELECT *
FROM CUSTOMER as CUS
JOIN EMPLOYEE as EMP
WHERE CUS.employee_id = EMP.employee_id
AND CUS.first_name LIKE “A%”;
```

HOW `JOIN` HELPS BETTER IN THE OPTIMIZATION

- `JOIN` helps to retrieve the data faster as its execution is faster.
- `JOIN` is opted instead of subquery as the retrieval time in the case of `JOINs` is faster than that of subqueries.

`INNER JOIN` in the query improves the performance instead of using **`OUTER JOIN`**. As the **`INNER JOIN`** results in common rows of the two joined tables, whereas in the case of **`OUTER JOIN`**, every row of the left table will appear in the result. **`INNER JOIN`** will always be better in performance than **`OUTER JOIN`** because no matter how large no rows there are, the **`INNER JOIN`** will output lesser data faster in retrieval and requires lesser time.

DRAWBACKS OF MULTIPLE JOINS

Multiple joins in the query will take more time to retrieve data from the database. Therefore, the database server will have to do more work. Thus the multiple joins lead to a heavy load on the database server and are time-consuming.

TASK 8

1) Find the Name of the employee who is assigned to the customer

Using optimized nested subqueries
Using nested subquery

```
select employee_name from banking_system.employee
where exists
(select * from banking_system.customer
where customer.employee_id = employee.employee_id
and customer.customer_id = "137")
```

5	0.00952925	select employee_name from banking_system.employee_details_table where exists (select * from banking_system.customer_details_tabl...
6	0.00086650	select employee_name from employee_details_table,t1 where t1.employee_id = employee_details_table.employee_id

id	select_type	table	partitions	type	possibl	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	employee_details...	NULL	ALL	hast...	NULL	NULL	NULL	500	100.00	Using where
1	SIMPLE	<subquery2>	NULL	eq_...	<au...	<auto_...	5	banking_system.employee_details_table.emplo...	1	100.00	NULL
2	MATERIALIZED	customer_details...	NULL	ALL	NULL	NULL	NULL	NULL	10098	10.00	Using where

Using optimized nested subquery
Option 1: using table

→ create table t4 as select employee_id
from banking_system.customer_details_table
where customer_details_table.customer_id between '111000' and '311000';
explain select employee_name from employee_details_table,t4
where t4.employee_id = employee_details_table.employee_id
and t4.employee_id = '111800';

id	select_type	table	partitions	type	possibl	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	employee_details...	NULL	ref	hast...	hasttrial1	5	const	1	100.00	NULL
1	SIMPLE	t4	NULL	ALL	NULL	NULL	NULL	NULL	2322	1.00	Using where; Using join buffer (hash join)

Option 2: inner join

→ select employee_name from banking_system.employee
inner join banking_system.customer
on employee.manager_id = customer.employee_id
and customer.customer_id = '211800';

id	select_type	table	partitions	type	possibl	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	customer_details...	NULL	ALL	NULL	NULL	NULL	NULL	10098	10.00	Using where
1	SIMPLE	employee_details...	NULL	ref	hast...	hasttrial1	5	banking_system.customer_details_table.employ...	1	100.00	NULL

2) Find contact information of the assigned employee for the customer

→ The execution is similar to that of finding name of employee from the employee table

Optimized nested subquery

Select contact_number from banking_system.employee
inner join banking_system.customer
on employee.manager_id = customer.employee_id
and customer.customer_id = ‘111800’;

3) Find account number of the customer

Normal nested subquery

→ select account_number from banking_system.depositor
where exists (select * from banking_system.customer
where customer.customer_id =depositor.customer_id
and customer.customer_id = "137");

id	select_type	table	partitions	type	possibl	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	depositor	NULL	ref	PRI...	PRIMARY	4	const	1	100.00	Using index

Execution time

123	0.00096700	select account_number from banking_system.depositor where exists (select * from banking_system.customer where customer.customer_id =depositor.customer_id and customer.customer_id = "137" and customer.customer_id between '100' and '498')
124	0.00043975	select account_number from banking_system.depositor inner join banking_system.customer on depositor.customer_id = customer.customer_id and customer.customer_id = '137'

Optimized nested subquery

Option 1: create table

→ create table t3 as select customer_id
from banking_system.customer
where customer.customer_id = '137';
select account_number from depositor,t3
where t3.customer_id = depositor.customer_id;

id	select_type	table	partitions	type	possibl	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	t3	NULL	ALL	NULL	NULL	NULL	NULL	1	100.00	NULL
1	SIMPLE	depositor	NULL	ref	PRI...	PRIMARY	4	banking_system.t3.customer_id	1	100.00	Using index

Option 2: inner join

→ select account_number from banking_system.depositor
inner join banking_system.customer
on depositor.customer_id = customer.customer_id
and customer.customer_id = '137';

id	select_type	table	partitions	type	possibl	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	customer	NULL	const	PRI...	PRIMARY	4	const	1	100.00	Using index
1	SIMPLE	depositor	NULL	ref	PRI...	PRIMARY	4	const	1	100.00	Using where; Using index

JUSTIFICATION

The nested subquery is useful to get the information across multiple tables. Here I have used a nested subquery to find the name and contact information of the employee from the employee table, which is assigned to the customer from the customer table. I have also used a nested subquery to gather the customer's account number from the depositor table. Here I have optimized the nested subquery using two approaches. The first is using create a table where we can store some of the required information in a table t1 and then directly call it. The other approach is using the inner join, where we can join the employee and customer table in parts 1 and 2, and join the depositor and customer table in part 3. Looking at the number of scan and execution times, we can find the difference between the normal nested subquery and the optimized nested subquery. Hence optimization decreases the time for execution to a great extent. All the queries are mentioned.