Steps to Optimize SQL Query Performance – Best Practices

We pay lots of attention to improve the performance of the web application, but ignore back-end SQL performance tuning. Even experts like application architects and the developer does not have an idea on how databases process the SQL queries internally. This could be because of lack of SQL and database knowledge. In this post, we will check best practices to **optimize SQL query performance**.

Steps to Optimize SQL Query Performance

Here are simple tips and steps that will *improve your SQL query performance* on databases. You can incorporate these best practices to tune SQL query performance.

Avoid SELECT * in Your Queries

Yes, you can improve query performance simply by replacing your SELECT * with actual column names. *Do not use * in your SQL queries, instead, use actual column names that you want to return*. The database may scan column names and replace * with actual columns of the table. Providing column names avoid search and replace, this may not be noticeable but definitely improves performance.

Original Query:

```
Select * from table_name;
```

Optimized Query;

```
select col1, col2, col3 from table_name;
```

Keep Table Statistics up-to-date

Databases like Netezza, Redshift *uses table statistics to generate an optimal execution plan*. Tables stats play important role in improving performance of SQL query execution.

Related reading:

- Best Practices to Optimize Hive Query Performance
- Netezza Generate statistics command and Examples

Avoid DISTINCT in SQL Queries

SELECT DISTINCT is most commonly used SQL clause to get unique rows from tables. But you should avoid using SELECT DISTINCT whenever possible. *SELECT DISTINCT works by GROUPing all fields in the query to create distinct results*. You can rewrite such queries by adding more unique columns in your column list.

Original Query:

```
SELECT DISTINCT FirstName, LastName, State
FROM Customers;
```

Optimized Query:

```
SELECT FirstName, LastName, State, Address, zip, city,
state
FROM Customers;
```

Another possible option to improve performance is to use GROUP BY to remove duplicate values instead of DISTINCT.

Avoid using UNION on Large Tables

Avoid using UNION clause whenever possible. *UNION clause causes sorting data in the table* and that slows down SQL execution. Instead, use UNION ALL and remove duplicates by adding more columns before applying UNION all.

Create Joins with INNER JOIN Rather than WHERE

Almost all databases support joining tables using JOIN clause and WHERE clause. In some databases, these types of queries are inefficient as it first creates temp data with all possible options (most probably CROSS JOIN) and then it applies WHERE conditions.

Original Query:

```
SELECT C.CustomerID, C.Name, S.LastSaleDate
FROM Customers as C, Sales as S
WHERE C.CustomerID = S.CustomerID;
```

Optimized Query:

```
SELECT C.CustomerID, C.Name, S.LastSaleDate
FROM Customers
   INNER JOIN Sales
   ON C.CustomerID = S.CustomerID;
```

Use WHERE instead of HAVING to Define Filters on non-aggregate Columns

Avoid using the HAVING clause on non-aggregate columns. Pull only records that are required to perform given task. *HAVING clause will filter out records only after grouping* them, instead, we can filter out these in initial step using WHERE clause.

Original Query:

```
SELECT C.CustomerID, C.Name, Count(S.SalesID)
FROM Customers as C
   INNER JOIN Sales as S
   ON C.CustomerID = S.CustomerID
GROUP BY C.CustomerID, C.Name
HAVING S.LastSaleDate BETWEEN '1/1/2019' AND
'12/31/2019';
```

Optimized Query:

```
SELECT C.CustomerID, C.Name, Count(S.SalesID)
FROM Customers as C
   INNER JOIN Sales as S
   ON C.CustomerID = S.CustomerID
WHERE S.LastSaleDate BETWEEN '1/1/2019' AND '12/31/2019'
GROUP BY C.CustomerID, C.Name;
```

Avoid Wildcard at Beginning of Predicate

You should avoid wild card at the beginning of the predicate. Try to write query with a wildcard at the end of the predicate. The predicates like '%abc' causes full table scan to get matching results set. Avoid using these kinds of statements. This is true for all databases including Hive framework.

Original Query:

```
Select col1, col2
From table_A
```

```
Where col2 like '%abc%';
```
Optimized Query:

```
Select col1, col2
From table_A
Where col2 like 'abc%';
```

Avoid using Functions in Predicates

Avoid using functions in WHERE clause as it may be costly for cost-based optimizer. Optimizer will not utilize index or bucket if there are any function on that columns.

For example, avoid queries something like below:

```
Select name from employee
Where to_char(join_date,'YYYYMMDD')  = '20190101';
```
In case if functions are required, use function at right side of the operator.

```
Select name from employee
Where join_date = cast('20190101' as date);
```

Avoid using IN Operator

Avoid using IN operator whenever possible. Use IN operator only if necessary. IN operator has slowest performance. Use EXISTS in place of IN. The database will check each result of the column to see if it is null.

Original Query:

```
Select * from products p
where product_id IN
(select product_id from order_items);
```
Optimized Query:

```
Select * from products p
where EXISTS (select * from order_items o
where o.product_id = p.product_id);
```

Concatenation in WHERE clause

Avoid concatenation in WHERE clause whenever possible. You should Avoid concatenating multiple columns in WHERE clause. If there is concatenation, break the query into multiple conditions.

Original Query:

```
SELECT id, name, salary
FROM employee
WHERE dept || location= 'ElectronicsINDIA';
```

Optimized Query:

```
SELECT id, name, salary
FROM employee
WHERE dept = 'Electronics'
AND location = 'INDIA';
```

Avoid Calculated Fields in JOIN and WHERE clause

You should not include any calculations on columns which are participating in JOIN and WHERE clause. Calculated fields in JOIN consumes resources and slows down the execution. For example:

```
SELECT id, name, salary
FROM employee
WHERE (salary *2) = 10000;
```

Use OLAP functionality (OVER and RANK)

OLAP functions or analytic function in SQL are fast and easy to use. You can make use of these functions whenever possible.

Original Query:

```
SELECT clicks.* FROM clicks_table inner join (select
sessionID, max(timestamp) as max_ts from clicks_table
group by sessionID) latest ON clicks.sessionID =
latest.sessionID
and clicks.timestamp = latest.max_ts;
```

Optimal Query:

```
SELECT * FROM (SELECT *, RANK() over (partition by
sessionID,order by timestamp desc) as rank FROM
clicks_table) ranked_clicks WHERE ranked_clicks.rank=1;
```

Considering the Cardinality within GROUP BY

GROUP BY can be little bit faster if you consider cardinality carefully. Consider unique column first in GROUP BY column list.

Original Query:

```
SELECT GROUP BY gender, uid;
```

Optimal Query:

```
SELECT GROUP BY uid, gender;
```