Writing Efficient SQL

1. Use Column Names Instead of * in a SELECT Statement

By selecting only, the columns you need, you are reducing the size of the result table, reducing the network traffic and also in turn boosting the overall performance of the query.

Inefficient and inaccurate:

```
select * from employee e;
```

Efficient and accurate:

```
select first_name ,department_id from employee e ;
```

2. Eliminate Unnecessary DISTINCT Conditions

Considering the case of the following example, the DISTINCT keyword in the original query is unnecessary because the table_name contains the primary key p.ID, which is part of the result set.

Inefficient and inaccurate:

```
explain  SELECT DISTINCT  e.department_id
FROM  employee e;
```

Efficient and accurate:

```
explain SELECT e.department_id from  employee e
group by  e.department_id ;
```

3. Use EXISTS instead of DISTINCT when using table joins that involves tables having one-to-many relationships

The DISTINCT keyword works by selecting all the columns in the table then parses out any duplicates.Instead, if you use sub query with the EXISTS keyword, you can avoid having to return an entire table

Inefficient and inaccurate:

```
explain select distinct d.department_name
   from department d join employee e
   on d.department_id =e.department_id
   and salary>2000;
```

Efficient and accurate:

```
  explain   select d.department_name
   from department d
   where exists ( select 1 from employee e
                              where e.department_id=d.department_id
                              and salary>2000);
```

4. Try to use UNION ALL in place of UNION

The UNION ALL statement is much faster than UNION, because UNION ALL statement does not look for duplicate rows, and UNION statement does look for duplicate rows, whether or not they exist.

Inefficient and inaccurate:

```
explain SELECT p.project_id , p.project_name
from project p
UNION
SELECT e.employee_id , e.first_name
FROM employee e ;
```

Efficient and accurate:

```
explain SELECT p.project_id , p.project_name
from project p
UNION ALL
SELECT e.employee_id , e.first_name
FROM employee e ;
```

5. Consider using an IN predicate when querying an indexed column

The IN-list predicate can be exploited for indexed retrieval and also, the optimizer can sort the IN-list to match the sort sequence of the index, leading to more efficient retrieval. Note that the IN-list must contain only constants, or values that are constant during one execution of the query block, such as outer references

Inefficient and inaccurate:

```
explain select e.employee_id ,d.department_name
from employee e join department d
on e.department_id =d.department_id
where e.department_id =5 or e.department_id =3
or e.department_id =2;
```

Efficient and accurate:

```
explain select e.employee_id ,d.department_name
from employee e join department d
on e.department_id =d.department_id
where  e.department_id in (5,3,2);
```

6. Avoid using OR in join conditions

Any time you place an ‘OR’ in the join condition, the query will slow down by at least a factor of two.

Inefficient and inaccurate:

```
explain  SELECT  d.department_name,
                 count(employee_id) totalemp
from employee e join department d
on e.department_id =d.department_id
or e.first_name =d.department_name
group by d.department_name;
```

Efficient and accurate:

```
explain  SELECT  d.department_name,
                 count(employee_id) totalemp
from employee e join department d
on e.department_id =d.department_id
group by d.department_name
```

```
union all
sELECT  d.department_name,
                              count(employee_id) totalemp
from employee e join department d
on e.first_name =d.department_name
group by d.department_name;
```

7. Wildcard vs Substr

Using wildcard will definitely slow down your query especially for table that are really huge. We can optimize our query with wildcard by doing a postfix wildcard instead of pre or full wildcard.

Inefficient and inaccurate:

```
select * from employees e
where e.first_name like '%430201A%';
--Prefix Wildcard
select * from employees e
where e.first_name like '%430201A’;
```

Inefficient and inaccurate:

```
explain select e.first_name,e.last_name
from employees e
where substring(e.first_name,1,7)= ‘430201A’;
```

Efficient and accurate:

```
--Postfix Wildcard
explain select e.first_name,e.last_name
from employees e
where e.first_name like ‘430201A%' ;
```

8. Create JOINs with INNER JOIN (not WHERE)

WHERE clause creates the CROSS join/ CARTESIAN product for merging tables. CARTESIAN product of two tables takes a lot of time.

Inefficient and inaccurate:

```
explain SELECT d.department_id , d.department_name ,
                                      e.first_name, p.project_name
FROM employee e, department d , project p
```

```
WHERE e.department_id = d.department_id
and d.department_id =p.department_id ;
```

Efficient and accurate:

```
explain SELECT d.department_id , d.department_name ,
e.first_name,p.project_name
FROM employee e join department d
on e.department_id = d.department_id
join project p on d.department_id =p.department_id;
```

9. Ignore linked subqueries

A linked subquery depends on the query from the parent or from an external source. It runs row by row, so the average cycle speed is greatly affected.

Inefficient and inaccurate:

```
select e.first_name , (select d.department_id from department d where
d.department_id=e.department_id)
from employee e ;
```

For each row returned by the external query, the inner query is run every time. Alternatively, **JOIN** can be used to solve these problems for SQL database optimization.

Efficient and accurate:

```
select e.first_name , d.department_id
from employee e join department d
on d.department_id=e.department_id;
```

similarly,

Inefficient and inaccurate:

```
select e.first_name , (select d.department_id from department d where
d.department_id=e.department_id
limit 1)
```

```
from employee e ;
```

Efficient and accurate:

```
select tt.first_name , tt.department_id
from (select e.first_name , d.department_id , row_number() over (partition by
d.department_id ) as rn
from employee e join department d
on d.department_id=e.department_id ) tt
where tt.rn=1;
```

10. IN versus EXISTS

IN operator is more costly than EXISTS in terms of scans especially when the result of the subquery is a large dataset. So we should try to use EXISTS rather than using IN for fetching results with a subquery.

Note: If the sub-query result is larger, then EXISTS works faster than the IN Operator. Usually IN has the slowest performance. IN works faster than the EXISTS Operator when If the sub-query result is small.

Inefficient and inaccurate:

```
explain select d.department_id ,d.department_name from department d
where d.department_id in (select p.department_id from project p);
```

Efficient and accurate:

```
explain select d.department_id ,d.department_name from department d
where exists (select p.department_id from project p
                                    where d.department_id=p.department_id);
```

11. Avoid operator

Inefficient and inaccurate:

```
explain SELECT e.employee_id , d.department_name
        FROM employee e join department d
        on e.department_id =d.department_id
        where salary >= 3000 and salary<= 5000 ;
```

Efficient and accurate:

```
explain SELECT e.employee_id , e.first_name
FROM employee e join department d
        on e.department_id =d.department_id
        WHERE salary BETWEEN 3000 and 5000;
```

12. Combine multiples scans with case statements

When we need to calculate multiple aggregates from the same table, avoid writing a multiple select query for each aggregate. Use CASE statement instead.

Inefficient and inaccurate:

```
explain
select e.department_id ,
                (select count(e.employee_id) from employee e where salary<1100),
         (select count(e.employee_id) from employee e where salary     between
1100 and 3000),
                (select count(e.employee_id) from employee e where salary>3000)
                from employee e
group by e.department_id ;
```

Efficient and accurate:

```
explain select e.department_id ,
                count( case when salary<1100 then 1 else null end ) count1,
 count(case when salary between 1100 and 3000 then 1 else null end) count2,
                count( case when salary>3000 then 1 else null end) count3
 from employee e
group by e.department_id ;
```

13. Making joins less complicated

it's better to reduce table sizes before joining them.

Inefficient and inaccurate:

```
explain
SELECT  d.department_name,
                count(employee_id) totalemp,
                SUM(salary) AS salary ,
                count(p.project_id) totalproject
from employee e join department d
on e.department_id =d.department_id
join project p on p.department_id =d.department_id
group by d.department_name;
```

So dropping that in a subquery and then joining to it in the outer query will reduce the cost of the join substantially:

Efficient and accurate:

```
explain
SELECT d.department_name ,a.totalemp,a.salary,p.totalproject
FROM department d
INNER JOIN (
                SELECT    department_id,
                                count(employee_id) totalemp,
                                SUM(salary) AS salary
                        FROM employee GROUP BY department_id
                        ) a
        ON d.department_id = a.department_id
join (select department_id,
                        count(p.project_id) totalproject
                        from project p
    group by department_id ) p on
    p.department_id=d.department_id ;
```

if you were talking about hundreds of thousands of rows or more, you'd see a noticeable improvement by aggregating before joining.

14. Wrong Join Case gives wrong results

```
explain select d.department_name , count(e.employee_id), count(p.project_id)
from department d join employee e
on e.department_id = d.department_id
join project p on p.department_id =d.department_id
group by d.department_name;
--1 min 11 sec
```

```
Administration          15004560    15004560
Finance                 61323010    61323010
Human Resource          66103190    66103190
IT                      15538936    15538936
Sales                   61476384    61476384
```

those are wrong numbers, where is the mistake?
You are aggregating along two independent dimension. The recommended alternative is usually to do aggregation *before* the join

Efficient and accurate:

```
select d.department_name , ee.ecount, pp.pcount
from department d
            left join (select e.department_id,count(e.employee_id) ecount
                        from employee e
                        group by e.department_id ) ee on d.department_id =
ee.department_id
            left join (select p2.department_id,count(p2.project_id) pcount
                        from project p2
                        group by p2.department_id ) pp
on pp.department_id=ee.department_id;
```

```
IT                      125314      124
```

Sales	249904	246
Finance	250298	245
Human Resource	249446	265
Administration	125038	120
Marketing		
Business		

15. **Use the select statements with TOP keyword or the SET ROWCOUNT statement, if you need to return only the first n rows**

    Queries that use the limit clause with ORDER BY or analytic function ROW_NUMBER() return a specific subset of rows in the query result. Vertica processes these queries efficiently using *Top-K Optimization*, which is a database query ranking process. Top-K optimization avoids sorting (and potentially writing to disk) an entire data set to find a small number of rows. This can significantly improve query performance.

Inefficient and inaccurate:

```
explain SELECT *
 FROM public.raw_serial_number_data e
 limit 200000;
```

Efficient and accurate:

```
explain select *
from (SELECT *,row_number () over(partition by id)as rn
 FROM public.raw_serial_number_data e)tt
 where tt.rn<=200000;
```