	Hope Foundation's, **Finolex Academy of Management and Technology, Ratnagiri**			
	Department of Information Technology			
Subject name: OLAP LAB				Subject Code: ITL503
Class	TE IT	Semester – V (CBCGS)		Academic year: 2018-19
Name of Student		**QUIZ Score :**		
Roll No		Assignment/Experiment No.		03
Title: **Implementation of query optimization Postgre SQL.**				

1. Course objectives applicable:

LOB1- Understand the fundamentals of query- based optimization techniques.

2. Course outcomes applicable:

LO1- Apply the knowledge of query processing and query optimization of database management system

3. Learning Objectives:

- To be able to apply the knowledge of query optimization techniques.
- To understand concept of query processing.

4. Practical applications of the assignment/experiment:

- Wherever the database management system and data retrieval is required E.g. Company database, banking system, Airline reservation system.

5. Prerequisites: SQL commands, Indexing

6. Hardware Requirements:

1. PC with 4GB RAM, 500GB HDD,

7. Software Requirements:

1. Postgre sql

8. Quiz Questions (if any): (Online Exam will be taken separately batch wise, attach the certificate/ Marks obtained)

1. https://goo.gl/BqcH8Q

9. Experiment/Assignment Evaluation:				
Sr. No.	**Parameters**		**Marks obtained**	**Out of**
1	Technical Understanding (Assessment may be done based on Q & A **or** any other relevant method.) Teacher should mention the other method used -			6
2	Neatness/presentation			2
3	Punctuality			2
Date of performance (DOP)		**Total marks obtained**		**10**
Date of checking (DOC)		**Signature of teacher**		

Theory: <HANDWRITTEN>

Query: A query is a request for information from a database.

Query Plans: A query plan (or query execution plan) is an ordered set of steps used to access data in a SQL relational database management system.

Query Optimization: A single query can be executed through different algorithms or re-written in different forms and structures. Hence, the question of query optimization comes into the picture – Which of these forms or pathways is the most optimal? The query optimizer attempts to determine the most efficient way to execute a given query by considering the possible query plans.

The goal of query optimization is to reduce the system resources required to fulfill a query, and ultimately provide the user with the correct result set faster. First, it provides the user with faster results, which makes the application seem faster to the user. Secondly, it allows the system to service more queries in the same amount of time, because each request takes less time than un- optimized queries.

Thirdly, query optimization ultimately reduces the amount of wear on the hardware (e.g. disk drives), and allows the server to run more efficiently (e.g. lower power consumption, less memory usage).

There are broadly two ways a query can be optimized:

1. Analyze and transform equivalent relational expressions: Try to minimize the tuple and column counts of the intermediate and final query processes
2. Using different algorithms for each operation: These underlying algorithms determine how tuples are accessed from the data structures they are stored in, indexing, hashing, data retrieval and hence influence the number of disk and block

PostgreSQL basic index terminology

Before describing the types of indexes in PostgreSQL and their use, let's take a look at some terminology that any DBA will come across sooner or later when reading the docs.

- **Index Access Method** (also called as **Access Method**): The index type (B-tree, GiST, GIN, etc)
- **Type:** the data type of the indexed column
- **Operator:** a function between two data types
- **Operator Family:** cross data type operator, by grouping operators of types with similar behavior
- **Operator Class** (also mentioned as **index strategy**): defines the operators to be used by the index for a column.

Types of Indexes in PostgreSQL

PostgreSQL provides the following Index types:

- **B-tree:** the default index, applicable for types that can be sorted.
- **Hash:** handles equality only.
- **GiST:** suitable for non-scalar data types (e.g. geometrical shapes, fts, arrays).
- **SP-GiST:** space partitioned GIST, an evolution of GiST for handling non-balanced structures (quadtrees, k-d trees, radix trees).
- **GIN:** suitable for complex types (e.g. jsonb, fts, arrays).
- **BRIN:** a relatively new type of index which supports data that can be sorted by storing min/max values in each block.

Observations: <HANDWRITTEN>

1.
2.
3.

Results:

B-tree Indexes:

When we define this rather common table, PostgreSQL creates two unique B-tree indexes behind the scenes: part_pkey and part_partno_key. So every unique constraint in PostgreSQL is implemented with a unique INDEX. Now let's try to do some queries on our table.

```
postgres=# select * from part where id=100000;
   id   | partno | partname | partdescr | machine_id
--------+--------+----------+-----------+------------
 100000 | 101    | abc      | aaa       |          1
(1 row)
```

```
postgres=# select * from part where partno='102';
   id    | partno | partname | partdescr | machine_id
---------+--------+----------+-----------+------------
 100001 | 102    | def      | bbb       |          2
(1 row)
```

We observe that it takes only fractions of the millisecond to get our results. We expected this since for both columns used in the above queries, we have already defined the appropriate indexes. Now let's try to query on column partname, for which no index exists.

```
postgres=# select * from part where partname='mno';
   id    | partno | partname | partdescr | machine_id
---------+--------+----------+-----------+------------
 100005 | 105    | mno      | eee       |          5
(1 row)
```

Here we see clearly that for the non indexed column, the performance drops significantly.

Now let's create an index on that column, and repeat the query:

```
postgres=#    select * from part where partname='abc';
   id    | partno | partname | partdescr | machine_id
---------+--------+----------+-----------+------------
 100000 | 101    | abc      | aaa       |          1
(1 row)
```

Our new index part_partname_idx is also a B-tree index (the default). First we note that the index creation on the million rows table took a significant amount of time, about 16 seconds. Then we observe that our query speed was boosted from 89 ms down to 0.525 ms. B-tree indexes, besides checking for equality, can also help with queries involving other operators on ordered types, such as <,<=,>=,>. Lets try with <= and >=

```
postgres=# select count(*) from part where partname>='jkl';
 count
-------
     2
(1 row)
```

```
postgres=# select count(*) from part where partname<='ghi';
 count
-------
     3
(1 row)
```

The first query is much faster than the second, by using the EXPLAIN (or EXPLAIN ANALYZE) keywords we can see if the actual index is used or not:

```
postgres=# explain select count(*) from part where partname>='def';
                        QUERY PLAN
-----------------------------------------------------------
 Aggregate  (cost=1.07..1.08 rows=1 width=8)
   ->  Seq Scan on part  (cost=0.00..1.06 rows=2 width=0)
         Filter: ((partname)::text >= 'def'::text)
(3 rows)
```

```
postgres=# explain select count(*) from part where partname<='def';
                        QUERY PLAN
-----------------------------------------------------------
 Aggregate  (cost=1.07..1.08 rows=1 width=8)
   ->  Seq Scan on part  (cost=0.00..1.06 rows=2 width=0)
         Filter: ((partname)::text <= 'def'::text)
(3 rows)
```

In the first case, the query planner chooses to use the part_partname_idx index. We also observe that this will result in an index-only scan which means no access to the data tables at all. In the second case the planner determines that there is no point in using the index as the returned results are a big portion of the table, in which case a sequential scan is thought to be faster.

Hash Indexes:

Use of hash indexes up to and including PgSQL 9.6 was discouraged due to reasons having to do with lack of WAL writing. As of PgSQL 10.0 those issues were fixed, but still hash indexes made little sense to use. There are efforts in PgSQL 11 to make hash indexes a first class index method along with its bigger brothers (B-tree, GiST, GIN). So, with this in mind, let's actually try a hash index in action.

We will enrich our part table with a new column parttype and populate it with values of equal distribution, and then run a query that tests for parttype equal to 'Steering':

```
postgres=# select count(*) from part where id % 100001 = 0 AND parttype = 'Steering';
 count
-------
     1
(1 row)
```

Now we create a Hash index for this new column, and retry the previous query:

```
postgres=# select count(*) from part where id % 100001 = 0 AND parttype = 'Steering';
 count
-------
     1
(1 row)
```

We note the improvement after using the hash index. Now we will compare the performance of a hash index on integers against the equivalent b-tree index.

```
postgres=# select * from part where id=100001;
   id   | partno | partname | partdescr | machine_id | parttype
--------+--------+----------+-----------+------------+----------
 100001 | 102    | def      | bbb       |     100001 | Steering
(1 row)
```

```
postgres=#    select * from part where machine_id=100001;
   id   | partno | partname | partdescr | machine_id | parttype
--------+--------+----------+-----------+------------+----------
 100001 | 102    | def      | bbb       |     100001 | Steering
(1 row)
```

```
postgres=# select * from part where machine_id=100001;
   id   | partno | partname | partdescr | machine_id | parttype
--------+--------+----------+-----------+------------+----------
 100001 | 102    | def      | bbb       |     100001 | Steering
(1 row)
```

As we see, with the use of hash indexes, the speed of queries that check for equality is very close to the speed of B-tree indexes. Hash indexes are said to be marginally faster for equality than B-trees, in fact we had to try each query two or three times until hash index gave a better result than the b-tree equivalent.

Learning Outcomes Achieved <HANDWRITTEN>

1. This program has achieved the objective of
2. The programs was coded in
3. It was proved that

Conclusion: <HANDWRITTEN>

References :

1. https://dzone.com/articles/simple-tips-for-postgresql-query-optimization

2. https://momjian.us/main/writings/pgsql/optimizer.pdf