INTRODUCTION

A **Benchmark** is the act of running a computer program, or a set of programs, or other operations, in order to assess the relative **performance** of an object, normally by running a number of standard tests and trials against it. The term benchmark is also mostly utilized for the purposes of elaborately-designed benchmarking programs themselves.

There are primarily two kinds of benchmarking. One is the **Functionality Benchmarking** to test all the functionalities supported by the system and the other is the **Performance Benchmarking** which test the speed of the system. Without Benchmarks, making detailed comparisons between different systems concerning their functionality or performance is generally difficult.

Database benchmarks are used to measure the throughput and response times of different databases. Well known database benchmarks are TPC-C, TPC-E, TPC-H, etc. But all of these benchmarks are for Relational Database System. There are few well known Benchmarks for Spatial Database System especially for vector data. One of them is VESPA [1].

This report aims at designing a new benchmark for the spatial database for the vector data. The databases that are being tested here are **Oracle 11g** and **Postgres 9.04/Postgis 1.5.2**.

The databases are being populated using **2010 TIGER/LINE data** [7]. And the queries that have been chosen are based on these data.

Since this project is basically a **comparison** between **Open-source** and **Proprietary products**, That is why the combination of **Oracle 11g** on **Windows 7** and **Postgres/Postgis** on **Ubuntu 10.10** have been chosen for the experiment. And also Postgres uses the underlying **GEOS**(Geometry Engine - Open Source) library functions to implement the geometric operations whereas Oracle 11g has its own, so we also need to test whose geometric algorithms are more efficiently designed.

The experiment has two parts, one is the **Cold Start** and the other one is the **Warm Start**. In the cold phase each query is being run freshly by clearing all the buffer cache pages. And in the warm phase, a bunch of related queries is run to measure the performances of the two databases. More on these will be discussed in the subsequent sections.

The rest of the report is presented as follows. Section 1 gives an overview of the TIGER data. Section 2 deals with the details of the Database explaining all the configuration parameters, data loading scripts, performance measurement tools, etc. Section 3 explains the Table sizes & indexes and commands to view them both in Postgres and in Oracle 11g. Section 4 gives the Experiment and Results. Section 5 gives the conclusion and summary of the queries and Percentage bar graphs for the two phases separately.

1. OVERVIEW OF TIGER DATA

The TIGER/Line Shapefiles are extracts of selected geographic and cartographic information from the U.S. Census Bureau's Master Address File/Topologically Integrated Geographic Encoding and Referencing (MAF/TIGER) database. The Shapefiles include information for the fifty states, the District of Columbia, and Puerto Rico, containing features such as roads, railroads, rivers, as well as legal and statistical geographic areas. The TIGER/Line Shapefiles are designed for use with geographic information system (GIS) software.

For our experiment we have used the **2010 TIGER/Line Shapefiles**. These Shapefiles are available for download by type, e.g. roads, linearwater, areawater, rails, etc. The following section will explain the details of these types. We will list only those features that we have used in our experiments. To get the details of all the features, kindly refer to its technical documentation [7].

a) Area Landmark:
Area Landmark Shapefile contains features such as airports, cemeteries, parks, educational facilities, shopping mall, etc. Following are the attributes of this table:

Field	Length	Type	Description
STATEFP	2	String	State FIPS code
COUNTYFP	3	String	County FIPS code
ANSICODE	8	String	Official code for the landmark for use by federal agencies for data transfer and dissemination
AREAID	22	String	Area landmark identifier
FULLNAME	100	String	Feature Name
MTFCC	5	String	MAF/TIGER feature class code
ALAND	14	Number	Land area
AWATER	14	Number	Water area
INTPTLAT	11	String	Latitude of the internal point
INTPTLON	12	String	Longitude of the internal point

Out of this comprehensive list, we have **used only** the **attributes** State Code, MTFCC and FULLNAME from all the tables for our queries. State & County FIPS code denotes the State & County to which this feature belongs. MTFCC denotes the type of feature. For example, MTFCC='K2361' denotes the Shopping center or Major Retail Center. The details of all the feature class code and their description is listed in the technical documentation [7]. FULLNAME denotes the feature name, e.g. the name of the shopping center.

b) Point Landmark:
Point Landmark Shapefile contains features such as schools, churches, hospitals, etc. Following are the attributes of this table:

Field	Length	Type	Description
STATEFP	2	String	State FIPS code
COUNTYFP	3	String	County FIPS code
ANSICODE	8	String	Official code for the point landmark for use by federal agencies for data transfer and dissemination
POINTID	22	String	Point landmark identifier

FULLNAME	100	String	Feature Name
MTFCC	5	String	MAF/TIGER feature class code

c) Area Water:
Area Hydrography Shapefile contains features such as ponds, lakes, oceans, swamps, etc. Following are the attributes of this table:

Field	Length	Type	Description
STATEFP	2	String	State FIPS code
COUNTYFP	3	String	County FIPS code
ANSICODE	8	String	Official code for the water body for use by federal agencies for data transfer and dissemination
HYDROID	22	String	Area Hydrography identifier
FULLNAME	100	String	Feature Name
MTFCC	5	String	MAF/TIGER feature class code
ALAND	14	Number	Land area
AWATER	14	Number	Water area
INTPTLAT	11	String	Latitude of the internal point
INTPTLON	12	String	Longitude of the internal point

d) Linear water:
Linear Hydrography Shapefile includes streams/rivers, braided streams, canals, ditches, etc. Following are the attributes of this table:

Field	Length	Type	Description
STATEFP	2	String	State FIPS code
COUNTYFP	3	String	County FIPS code
ANSICODE	8	String	Official code for the point landmark for use by federal agencies for data transfer and dissemination
LINEARID	22	String	Point landmark identifier
FULLNAME	100	String	Feature Name
MTFCC	5	String	MAF/TIGER feature class code

e) Roads:
Roads Shapefile contains all types of roads in U.S. It includes primary roads, secondary roads, local neighborhood roads, rural roads, city streets, etc. Following are the attributes of this table:

Field	Length	Type	Description
STATEFP	2	String	State FIPS code
COUNTYFP	3	String	County FIPS code
LINEARID	22	String	Point landmark identifier
FULLNAME	100	String	Feature Name
MTFCC	5	String	MAF/TIGER feature class code

f) Primary Roads:

Primary roads are generally divided, limited-access highways within the Federal interstate highway system or under state management. These highways are distinguished by the presence of interchanges and are accessible by ramps, and may include some toll highways. These are basically the inter-connector between states. Following are its attributes.

Field	Length	Type	Description
LINEARID	22	String	Linear Feature identifier
FULLNAME	100	String	Feature Name
MTFCC	5	String	MAF/TIGER feature class code

g) Rails:

Rails Shapefile includes all the Railway features found in the U.S. Following are its attributes:

Field	Length	Type	Description
LINEARID	22	String	Linear Feature identifier
FULLNAME	100	String	Feature Name
MTFCC	5	String	MAF/TIGER feature class code

h) States:

States Shapefile have been used to obtain the geometry of a particular state given a State FIPS code. This Shapefile has many attributes which can be seen in the technical documentation under **state10.shp** Shapefile.

2. DETAILS OF THE DATABASES

For this experiment to carry out effectively we have used two different machines having the same configuration, one for the Postgres and the other for the Oracle 11g. Both the machines have Intel(R) Core(TM)2 Duo CPU E450 @ 2.20GHz and 2.00 GB RAM.

2.1 POSTGRES

Postgres 9.04 and Postgis 1.5.2 have been used for this experiment. It has been installed on Ubuntu 10.10, a 32-bit operating system. The installation is being done without modifying the default values of the configuration parameters, i.e. all the parameters have the default values. Following are the **Memory** parameters with their default values.

1. Block size=8 KB
2. shared_buffers=28 MB
3. temp_buffers=8 MB
4. work_mem=1 MB
5. maintenance_work_mem=16 MB
6. max_stack_depth=2 MB

The following are the definitions of these parameters [9]:

1. **Block size** is the unit of storage and I/O.
2. **shared_buffers** is the amount of memory the database server uses for shared memory buffers. The default is typically 32 MB, but might be less if the kernel settings do not support it. This is determined during initdb. In our case it is 28 MB which is less than the default value 32 MB. However, settings significantly higher than the minimum are usually needed for good performance. This parameter can only be set at server start. If the database server has 1GB or more of RAM, a reasonable starting value for shared_buffers is 25% of the memory. There are some workloads where even large settings for shared_buffers are effective, but because PostgreSQL also relies on the operating system cache, it is unlikely that an allocation of more than 40% of RAM to shared_buffers will work better than a smaller amount.
3. **temp_buffers** is the maximum number of temporary buffers used by each database session. These are session-local buffers used only for access to temporary tables. The setting can be changed within individual sessions, but only before the first use of temporary tables within the session, subsequent attempts to change the value will have no effect on that session.
4. **work_mem** is the amount of memory to be used by internal sort operations and hash tables before writing to temporary disk files. For a complex query, several sort or hash operations might be running in parallel; each operation will be allowed to use as much memory as this value specifies before it starts to write data into temporary files. Also, several running sessions could be doing such operations concurrently. Therefore, the total memory used could be many times the value of work_mem.
5. **maintenance_work_mem** is the maximum amount of memory to be used by maintenance operations, such as VACUUM, CREATE INDEX, ALTER TABLE, ADD FOREIGN KEY, etc. Since only one of these operations can be executed at a time by a database session, and an installation normally

doesn't have many of them running concurrently, it's safe to set this value significantly larger than **work_mem**. Larger settings might improve performance for vacuuming and for restoring database dumps. VACUUM scans a table, marking tuples that are no longer needed as free space so that they can be overwritten by newly inserted or updated data. More on VACUUM can be found on [17] and [18].

6. **max_stack_depth** is the maximum safe depth of the server's execution stack. The ideal setting for this parameter is the actual stack size limit enforced by the kernel. Setting **max_stack_depth** higher than the actual kernel limit will mean that a runaway recursive function can crash an individual backend process. On platforms where PostgreSQL can determine the kernel limit, the server will not allow this variable to be set to an unsafe value.

2.1.1 DATA LOADING

The TIGER/Line data has been downloaded from their main server using the bash command **wget**. A list of urls were provided in a file and was supplied to this command for each feature, so that multiple features can be downloaded simultaneously. To load these features into the database, a script has been written called load.sh whose code is given below:

/* Script for Postgres */

```
cd “Path to the required Feature” ; ls -p | grep "/" | awk '
                    BEGIN{FS="/";i=1;}
                    {
                    if(i==1)
                    {
                    str="cd " $1 ";";
                    str=str "/usr/local/pgsql/bin/shp2pgsql -s 4326 -I " $1 " tiger.feature |";
                    str=str "grep -v \"\\[\" | ";
                    str=str "/usr/local/pgsql/bin/psql -d test;" "cd ..;";
                    }
                    else
                    {
                    str="cd " $1 ";";
                    str=str "/usr/local/pgsql/bin/shp2pgsql -a -s 4326 " $1 " tiger.feature |";
                    str=str "grep -v \"\\[\" | ";
                    str=str "/usr/local/pgsql/bin/psql -d test;" "cd ..;";
                    }
                    print str>>"Generate ld.sh";
                    i++;
```

}' ; cp “copy ld.sh to the required folder”

This program generates a new script file called ld.sh for each feature and copies it to the directory where that feature lies. The reason for not using a single script for loading is that if we run ld.sh in different terminals for each new feature, then multiple features can be loaded into the database at the same time, i.e. they will be like concurrent transactions.

The program uses the command **shp2pgsql** to convert the shape files into SQL and psql executes these SQL commands to populate the database. The following is the brief description of some of its options:

(c|a|d|p) These are mutually exclusive options:

-c Creates a new table and populates it from the shapefile. This is the default mode.

-a Appends data from the Shape file into the database table. Note that to use this option to load multiple files, the files must have the same attributes and same data types.

-d Drops the database table before creating a new table with the data in the Shape file.

-p Only produces the table creation SQL code, without adding any actual data. This can be used if you need to completely separate the table creation and data loading steps.

-s <SRID> Creates and populates the geometry tables with the specified SRID.

-I Create a GIST index on the geometry column.

There more other options details of which can be found in [12].

2.1.2 PERFORMANCE MEASUREMENT

The tool that has been used to measure the performance of each query in Postgres is its in-built command **EXPLAIN (ANALYZE,BUFFERS).**

EXPLAIN command displays the execution plan that the PostgreSQL planner generates for the supplied statement. The execution plan shows how the table(s) referenced by the statement will be scanned, e.g. by plain sequential scan, index scan, etc. and if multiple tables are referenced, what join algorithms will be used to bring together the required rows from each input table. It also displays the estimated statement execution cost, which is the planner's guess at how long it will take to run the statement [10].

But **EXPLAIN ANALYZE** causes the statement to be actually executed, not only planned. The total elapsed time in milliseconds within each plan node and total number of rows it actually returned are displayed. In the display two numbers are shown: the **start-up time** which is the time taken to return the first row, and the **total time** to return all the rows. Apart from time, there is another parameter **cost** which is the estimate of the time a node is expected to take. It also has two costs one is start-up cost and the other is the total-cost. By default costs are in units of “time a sequential 8 KB block read takes”. Optimizer selects plans based on overall lowest start-up and total cost. In reality, there may be a huge discrepancies between the estimated cost and the actual time. The following are the conventions used by the planner while computing the cost. These parameters are measured on an arbitrary scale.

seq_page_cost = 1.0 which is the cost of a sequentially fetched disk page.

random_page_cost = 4.0 which is the cost of a non-sequentially fetched disk page(4x slower).

cpu_tuple_cost = 0.01 cpu cost of processing each tuple or row(100x faster).

cpu_operator_cost = 0.0025 cost of processing each operator or function call.

cpu_index_tuple_cost = 0.005 cost of processing each index entry during an index scan.

ANALYZE collects statistics about the contents of tables in the database, and stores the results in the **pg_statistic** system catalog. Subsequently, the query planner uses these statistics to determine the most efficient execution plans for queries. Hence **EXPLAIN ANALYZE** not only executes the SQL statement but also collects the statistics in **pg_statistic** system catalog [11].

BUFFERS command has been added recently in the Postgres 9.0, previous versions don't support this command. It shows information on buffer usage. Specifically, include the number of shared blocks hits, reads, and writes, the number of local blocks hits, reads, and writes, and the number of temp blocks reads and writes. Shared blocks, local blocks, and temp blocks contain tables and indexes, temporary tables and temporary indexes, and disk blocks used in sort and materialized plans, respectively. The number of blocks shown for an upper-level node includes those used by all its child nodes. In text format, only non-zero values are printed. This parameter may only be used with ANALYZE parameter. It defaults to FALSE [10].

Following is an example to sum up these concepts:

```
                                                                QUERY PLAN
-------------------------------------------------------------------------------------------------------------------------------------------

 HashAggregate  (cost=53.95..73.56 rows=1961 width=12) (actual time=2879.228..2879.243 rows=13
loops=1)
   Buffers: shared hit=117295 read=67

  ->  Nested Loop  (cost=0.00..37.73 rows=6490 width=12) (actual time=282.130..2879.053 rows=40
loops=1)
        Join Filter: _st_intersects(a.the_geom, b.the_geom)
        Buffers: shared hit=117295 read=67

        ->  Seq Scan on states b  (cost=0.00..3.30 rows=2 width=1065526) (actual time=0.020..0.050 rows=2
loops=1)
              Filter: ((statefp10)::text = '36'::text)
              Buffers: shared hit=2

        ->  Index Scan using areawater_the_geom_gist on areawater a  (cost=0.00..16.43 rows=3
width=3288) (actual time=0.068..12.497 rows=2384 loops=2)
              Index Cond: (a.the_geom && b.the_geom)
              Buffers: shared hit=2111 read=9

 Total runtime: 2948.711 ms
```

Here **cost** denotes the estimated cost, **actual time** is the actual time of execution, **rows** denote the number of tuples returned as output. Two parameters **x..y** in the cost and time denotes the **start-up** time(x) and the **total** time(y) respectively, **read** denotes number of blocks read into the buffer and **shared hit** denotes the number of buffer hits. So Hit Ratio can be calculated as: Hit Ratio= shared hit/ **(**shared hit+read).

2.2 ORACLE 11g

Oracle 11g has been **installed** on **Windows 7 Professional**, 32-bit Operating System. The installation is being done without modifying the default values of the parameters, i.e. all the parameters have the default values. On executing the command **show sga,** the following output has been obtained:

SQL> show sga
Total System Global Area 535662592 bytes
Fixed Size 1375792 bytes
Variable Size 289407440 bytes
Database Buffers 239075328 bytes
Redo Buffers 5804032 bytes

The **SGA or System Global Area** is a chunk of memory that is allocated by an Oracle Instance and is shared among Oracle processes, hence the name. It contains all sorts of information about the instance and the database that is needed to operate.

Components of **SGA** consists of are **Fixed Size, Variable Size, Database Buffers, Redo Buffers.**

Fixed size is a part of the SGA containing general information about the state of the database and the instance, which the background processes need to access. This is also called the **fixed SGA**. No user data is stored here. The SGA also includes information communicated between processes, such as locking information. This is constant for a release and cannot be changed through any means such as altering the initialization parameters using alter command or in the init.ora file.

The **variable portion** is called *variable* because its size can be changed. It consists of:

- working space for **rman**. RMAN is a program that helps to manage backing up, restoring and recovering databases.
- Shared pool: The **shared pool** is used for objects that are shared among all users. For example: table definitions, PL/SQL definitions, cursors and so on.

Database Buffers is equal to is equal to db_block_size * db_block_buffers. **db_block_size** is the size the database blocks. In our case it is 8 KB.

SQL> show parameter db_block_size

NAME	TYPE	VALUE
db_block_size	integer	8192

db_block_buffers or buffer cache holds copies of data blocks so as they can be accessed quicker by oracle than by reading them off disk. The purpose of the buffer cache is to minimize physical I/O.

Redo Buffers: All changes that are covered by redo is first written into the redo buffers. The idea to first store it in the memory is to reduce disk I/O. When a transaction commits, the redo log buffer must be flushed to disk, because otherwise the recovery for that commit could not be guaranteed. Log Writer (LGWR) does that flushing.

In every system there is a limit to which **SGA** can grow, this limit is imposed by the parameter **sga_max_size**. In our system this is:

SQL> show parameter sga_max_size;

NAME	TYPE	VALUE
sga_max_size	big integer	512M

There is another parameter called **sga_target** whose value is always less than or equal to **sga_max_size**. Any attempt to raise this value beyond that will give error. There are other relevances of these two parameters, we will discuss those later in the report.

Another memory component of Oracle is **PGA or Program global area**. A PGA is a memory region that contains data and control information for a server process. It is non-shared memory created by Oracle Database when a server process is started. Access to the PGA is exclusive to the server process. There is one PGA for each server process. Background processes also allocate their own PGAs. The total memory used by all individual PGAs is known as the total instance PGA memory, and the **collection of individual PGAs is referred to as the total instance PGA, or just instance PGA**. We can use database initialization parameters to **set the size of the instance PGA, not individual PGAs**. The PGA is used to process SQL statements and to hold login and other session information. The **PGA memory** is basically divided into the following areas [6]:

- **Session Memory**

Session memory is the memory allocated to hold a session's variables (login information) and other information related to the session. For a shared server, the session memory is shared and not private.

- **Private SQL Area**

The private SQL area contains data such as bind variable values, query execution state information, and query execution work areas. Each session that issues a SQL statement has a private SQL area. Each user that submits the same SQL statement has his or her own private SQL area that uses a single shared SQL area. Thus, many private SQL areas can be associated with the same shared SQL area. The location of a private SQL area depends on the type of connection established for a session. If a session is connected through a dedicated server, private SQL areas are located in the server process's PGA. However, if a session is connected through a shared server, part of the private SQL area is kept in the SGA.

The following figure illustrates the two memory components PGA & SGA [6]:

Oracle has introduced **Automatic Shared Memory Management** in Oracle 10g and thus allows **automatic tuning** of the components of **SGA**. By default automatic tuning is disabled. The **zero value** of **sga_target** signifies that. To enable it, **sga_target** must be set a non-zero value. In our system it is zero:

SQL> show parameter sga_target;

NAME	TYPE	VALUE
sga_target	big integer	0

However, Oracle 11g has introduced a new concept called **Automatic Memory Management(AMM)** which automates SGA as well as PGA size according to the workload by dynamically transferring the

memory from SGA to PGA and vice versa. With this memory management method, the database also dynamically tunes the sizes of the individual SGA components and the sizes of the individual PGAs. To enable it, we need to set a new initialization parameter called **MEMORY_TARGET**. This parameter is dynamic and can be increased to maximum memory size specified by a new initialization parameter, called **MEMORY_MAX_TARGET**. The **memory_target** parameter specifies the amount of shared memory available for Oracle to use when dynamically controlling, reducing or enlarging the SGA and PGA as needed. In our system following are the values of these parameters:

SQL> show parameter memory_target;

NAME	TYPE	VALUE
memory_target	big integer	812M

SQL> show parameter memory_max_target;

NAME	TYPE	VALUE
memory_max_target	big integer	812M

Since these are parameters are set to non-zero values, **AMM** is enabled in our system.

2.2.1 DATA LOADING

A batch file script has been written to load TIGER data into Oracle 11g. Following is the script:

```
/* Script for Oracle 11g */

set "oracle_home=C:\app\subham\product\11.2.0\dbhome_1"
set "class_path=%oracle_home%\jdbc\lib\ojdbc5.jar;%oracle_home%\md\jlib\sdoutl.jar;%oracle_home
%\md\jlib\sdoapi.jar"

Setlocal EnableDelayedExpansion
set var=0

FOR /D %%G IN ("tl*") DO (
IF "!var!"=="0" (
java -cp %class_path% oracle.spatial.util.SampleShapefileToJGeomFeature -h localhost -p 1521 -s spatial
-u subham -d subham -t /*Feature_Name*/ -f %%G\%%G -i gid -r 4326 -g geom
set var=1
) ELSE (
java -cp %class_path% oracle.spatial.util.SampleShapefileToJGeomFeature -a -h localhost -p 1521 -s
spatial -u subham -d subham -t /*Feature_Name*/ -f %%G\%%G -i gid -r 4326 -g geom
)
)
```

To facilitate parallel data loading into the database, this script is generated for each feature and is run in different command prompts.

The script starts with the initialization of the environment variables **oracle_home** and **class_path.** Then it

uses **FOR /D** to loop through directory.

The shapefiles have been loaded using the Java Shapefile Converter which transforms an Shapefile into an Oracle database table for use with Oracle Spatial and Locator. The Shapefile Converter uses the Oracle Spatial Java-based Shapefile Adapter and **SampleShapefileToJGeomFeature** classes to load a Shapefile directly into a database table, with the Oracle-equivalent .dbf data types for the attribute columns and the SDO_GEOMETRY data type for the geometry column. The options used are:

-h: Host machine
-p: Host machine's port
-s: Host machine's SID
-u: Database user
-d: Database user's password
-t: Table name for the converted Shapefile
-f: File name of an input Shapefile (without extension)
-i: Column name for unique numeric ID; if required
-r: Valid Oracle SRID for coordinate system; 0 is used if unknown
-g: Preferred SDO_GEOMETRY column name
-a: Append Shapefile data to an existing table

2.2.2 PERFORMANCE MEASUREMENT

Performance measurement of a query in oracle has been done using **trace files** and **tkprof**. Trace files are created in the directory specified by the parameter **user_dump_dest.** In our system it is:

SQL> show parameter user_dump_dest;

NAME	TYPE	VALUE
user_dump_dest	string	c:\app\subham\diag\rdbms\spatial\spatial\trace

Before starting to measure the query performance following steps should be followed:

1. To start a SQL trace for the current session, execute:
   ALTER SESSION SET sql_trace = true;

2. To add an identifier to the trace file name for later identification:
   ALTER SESSION SET tracefile_identifier = mysqltrace;

3. By default, oracle doesn't include the timing information into the sql trace files. To include it, we have to execute this command:
   ALTER SESSION SET timed_statistics=true;

After executing the above commands, the actual query or queries are run to generate the trace file. The **TKPROF** program converts these trace files into a more readable form. Following is the syntax of **tkprof:**

TKPROF <trace-file> <output-file> EXPLAIN=username/password

If **EXPLAIN** option is given, then **TKPROF** determines the execution plan for each SQL statement in the trace file and writes these execution plans to the output file. It determines these execution plans by issuing

the **EXPLAIN PLAN** statement after connecting to Oracle with the username and password specified in this parameter. So it may take longer time to process a large trace file if the **EXPLAIN** option is used. The following is a sample output of tkprof:

call	count	cpu	elapsed	disk	query	current	rows
Parse	1	0.00	0.02	0	0	0	0
Execute	1	0.00	0.00	0	0	0	0
Fetch	2	1545.37	1548.40	17	537	4	13
total	4	1545.37	1548.42	17	537	4	13

The output is given separately for each phase:

PARSE : This phase translates the SQL statement into an execution plan, including checks for proper security authorization and checks for the existence of tables, columns, and other referenced objects.

EXECUTE : In this phase actual execution of the statement occurs. For INSERT, UPDATE, and DELETE statements, this modifies the data. For SELECT statements, this identifies the selected rows.

FETCH : This phase retrieves rows returned by a query. Fetches are only performed for **SELECT** statements.

The following are the **definitions** of these parameters [6]:

COUNT denotes number of times a statement was parsed, executed, or fetched.

CPU denotes the total CPU time in seconds for all parse, execute, or fetch calls for the statement. This value is zero (0) if TIMED_STATISTICS is not turned on.

ELAPSED is the total elapsed time in seconds for all parse, execute, or fetch calls for the statement. This value is also zero (0) if TIMED_STATISTICS is not turned on.

DISK indicates the number of blocks read from disk.

QUERY denotes total number of buffers retrieved in consistent mode for all parse, execute, or fetch calls. Usually, buffers are retrieved in consistent mode for queries.

CURRENT denotes total number of buffers retrieved in current mode. Buffers are retrieved in current mode for statements such as INSERT, UPDATE, and DELETE.

Actually, the **sum** of **query** and **current** is the total number of buffers accessed from both memory and disk, also called **Logical I/Os**.

Hence, *Hit Ratio* can be calculated as: Hit Ratio = 1 - (DISK / Logical I/Os)

ROWS indicate total number of rows processed by the SQL statement. This total does not include rows processed by sub-queries of the SQL statement.

3. TABLES & INDEXES

Below lists all the tables along with their sizes, number of tuples, indexes for the two databases:

3.1 POSTGRES:

Firstly, a schema named **tiger** was created and then all these following tables were created inside this tiger schema. So each table is prefixed with **"tiger."**. Each table listed here has an index. To view the index name of a particular table, one can issue this command: **\d table_name** in psql.

The number of tuples have been found by firing the SQL query: **select count(*) from table_name;**
The **sizes of each table** is obtained by issuing the command:

select pg_size_pretty(pg_relation_size('table_name'));

Table Name	Index Name	Number of Tuples	Size
arealm	arealm_the_geom_gist	1,15,646	46 MB
areawater	areawater_the_geom_gist	5,06,201	245 MB
linearwater	linearwater_the_geom_gist	56,90,700	2744 MB
pointlm	pointlm_the_geom_gist	6,75,193	63 MB
rails	rails_the_geom_gist	1,85,971	57 MB
roads	roads_the_geom_gist	2,02,43,400	6184 MB
primaryroads	primaryroads_the_geom_gist	22359	30 MB
states	states_the_geom_gist	104	16 KB

3.2 ORACLE 11g:

To view the index name of a particular table in oracle 11g, following SQL command is used:
select index_name from dba_indexes where table_name='TABLENAME';

And the sizes of each table is obtained using the SQL command:
select segment_name table_name, sum(bytes)/(1024*1024) table_size_meg from user_extents where segment_type='TABLE' and segment_name = 'MYTAB' group by segment_name;

Table Name	Index Name	Number of Tuples	Size
arealm	arealm_idx	1,15,646	62 MB
areawater	areawater_idx	5,06,201	312 MB
linearwater	linearwater_idx	56,90,700	4330 MB
pointlm	pointlm_idx	6,75,193	55 MB
rails	rails_idx	1,85,971	88 MB
roads	roads_idx	2,02,43,400	8393 MB
primaryroads	primaryroads_idx	22359	22 MB
states	states_idx	104	64 KB

4. EXPERIMENTS & RESULTS

In our experiment, the readings have been taken by running each query single, i.e. there were no other memory or cpu intensive operations or program running in parallel on the system except the operating system itself. And also there was only one connection made to the database.

Here we measure not only the CPU cost of each query but also the Buffer hits and Disk I/Os. For each query, we also output its query plan chosen by the query planner.

This experiment has two parts one is the **Cold Part** and the other is the **Warm Part**. **Each reading in the Cold Phase has been taken only once.**

4.1 COLD PHASE

In this phase each query is being run freshly on the system by clearing all the buffer cache pages.

Postgres does region based memory management creating and destroying **memory contexts or regions** as and when required. So stopping and restarting the server will destroy all the memory regions and free up the buffer contents. But Postgres also heavily relies on the Linux system cache. It is designed so because portable software like PostgreSQL cannot know enough about the filesystem or disk layout to make optimal decisions about how to read and write files. So the shared buffer cache is really duplicating what the operating system is already doing: caching popular file blocks [14].

pg_buffercache module in the **contrib** folder can be used to view the contents of its buffer cache [8]. And **pgfincore** module let us know which and how many disk blocks from a relation are in the **Buffer cache of the Operating System** [15].

Due to all these reasons we need to clear the Linux buffer pages too to get the correct result. Hence the commands used to do so are:

sync to flush all the unwritten and dirty data blocks to disk to be sure that everything is safely written.

Then issuing the command **echo 3 > /proc/sys/vm/drop_caches** to free the page-cache, dentries and inodes.

So every time, before beginning to execute a new query, postgres server was stopped and restarted and the above two commands were run.

On the other hand, **Oracle 11g** provides an in-built command: **alter system flush buffer_cache** to flush out the contents of the buffer cache in the **SGA**. This command was run every time before executing a new query.

The queries have been chosen carefully to cover up most of the functionalities that are supported by a spatial database. For example operations such as intersection, touches, inside, etc. For each category we have given queries comprising of different features and sizes, e.g. Polygon/Point, Polygon/Line, Line/Line, etc.

These queries also test the efficiency of the optimizer, i.e. selecting the most efficient query plan, proper and efficient use of indexes, etc. For example suppose there are two spatial relations R1 & R2, let R2 is the smaller one which fits into the main memory. Suppose both the relations have the R-Tree indexes. Now for queries such as **Retrieve all those geom_ids from R1 which intersects with a particular object present in R2.** Now the most efficient plan for this will be a sequential scan on R2 to retrieve the geometry of that particular object since R2 fits in main memory and then use the R-Tree of R1 to retrieve all the intersecting

objects. Now if the planner chooses just the opposite plan then it will take too much time to process this query. Thats why in every query we have used one Big Relation and one small Relation which is **states** since it contains only **104 tuples and fits in the memory**.

The following are the queries and time taken by each database:

1. CONTAINS PROPERLY

This geometric operation tests whether one object is completely within the other one.

Postgres supplies **boolean ST_ContainsProperly(geometry geomA, geometry geomB)** for this purpose. This function returns true if B intersects the interior of A but not the boundary (or exterior). This function call will automatically include a bounding box comparison that will make use of any indexes that are available on the geometries [3].

Oracle 11g supplies **SDO_INSIDE(geometry1, geometry2)** and **SDO_CONTAINS(geometry1, geometry2)** for this purpose, where **geometry1 must be spatially indexed** [4].

INSIDE(A,B) is the opposite of CONTAINS(B,A). That is, A INSIDE B implies B CONTAINS A.

But there is a problem with the oracle 11g functions. If B is the smaller relation and A is the larger one and both are having R-Tree indices on them, then also CONTAINS(B,A) choses the **inefficient plan** or the opposite plan of sequential scanning the larger relation and using the R-Tree index on the smaller one. Even giving a **query hint** to the query didn't solve the problem. This can be illustrated with the following result:

```
select /*+ index(a pointlm_idx) */ a.fullname from pointlm a, states b where
a.mtfcc='K1231' and b.statefp10='11' and sdo_contains(b.geom,a.geom)='TRUE'
```

call	count	cpu	elapsed	disk	query	current	rows
Parse	1	0.00	0.00	0	0	0	0
Execute	1	0.00	0.00	0	0	0	0
Fetch	3	134.48	135.57	6939	78617	4592	24
total	5	134.48	135.57	6939	78617	4592	24

Misses in library cache during parse: 0
Optimizer mode: ALL_ROWS
Parsing user id: 92 (SUBHAM)

Rows Row Source Operation
---

24 NESTED LOOPS (cr=3775543 pr=9847 pw=0 time=0 us cost=1894 size=37746 card=233)
2296 TABLE ACCESS FULL POINTLM (cr=6928 pr=6923 pw=0 time=193162 us cost=1894 size=791588 card=11641)
24 TABLE ACCESS BY INDEX ROWID STATES (cr=3768615 pr=2924 pw=0 time=157 us cost=1894 size=94 card=1)
4592 DOMAIN INDEX STATES_IDX (cr=3764114 pr=2924 pw=0 time=37851 us cost=0 size=0 card=0)

```
Rows     Execution Plan
-------  ---------------------------------------------------
     0  SELECT STATEMENT   MODE: ALL_ROWS
    24   NESTED LOOPS
  2296    TABLE ACCESS   MODE: ANALYZED (FULL) OF 'POINTLM' (TABLE)
    24    TABLE ACCESS   MODE: ANALYZED (BY INDEX ROWID) OF 'STATES'
           (TABLE)
  4592     DOMAIN INDEX OF 'STATES_IDX' (INDEX (DOMAIN))
```

As we can see from the result, **Oracle 11g** took about **135 seconds** to execute this query since the planner couldn't choose the optimized plan in-spite of giving the query hint.

Whereas in **Postgres,** the same query took only about **0.4 seconds** to execute, since the query planner was able to produce the correct plan. Following is the Postgres result:

```
explain (analyze,buffers) select a.fullname from tiger.pointlm as a, tiger.states as b
where ST_ContainsProperly(b.the_geom,a.the_geom) and a.mtfcc='K1231' and
b.statefp10='11';
```

```
                                                                QUERY PLAN
-------------------------------------------------------------------------------------------------------------------------------------------
 Nested Loop  (cost=0.00..36.76 rows=8 width=14) (actual time=127.512..322.979 rows=24 loops=1)
   Join Filter: _st_containsproperly(b.the_geom, a.the_geom)
   Buffers: shared hit=890 read=121
   ->  Seq Scan on states b  (cost=0.00..3.30 rows=2 width=1065526) (actual time=17.851..17.923 rows=2
loops=1)
         Filter: ((statefp10)::text = '11'::text)
         Buffers: shared read=2
   ->  Index Scan using pointlm_the_geom_gist on pointlm a  (cost=0.00..16.47 rows=1 width=114) (actual
time=28.223..98.370 rows=13 loops=2)
         Index Cond: (b.the_geom && a.the_geom)
         Filter: ((a.mtfcc)::text = 'K1231'::text)
         Buffers: shared hit=383 read=81
 Total runtime: 324.812 ms
```

For this reason we **used** the function **sdo_inside()** instead of **sdo_contains()**.

The following are the queries involving different features:

We have chosen the state **District_of_Columbia** of **minimum area** to test the efficiency of the R-tree index. It is because if the R-Tree is built efficiently then there will be less overlapping nodes and the query involving the minimum area geometry will be fast. Its state code is **11**.

1.1 Polygon/Point

Here we queried **Retrieve all the Hospitals located inside the state District_of_Columbia.** Hospitals have the mtfcc code of K1231 [7].

Oracle 11g:

```
select a.fullname from  pointlm a, states b where a.mtfcc='K1231' and b.statefp10='11' and  sdo_inside(a.geom,b.geom)='TRUE'
```

call	count	cpu	elapsed	disk	query	current	rows
Parse	1	0.04	0.88	78	1623	0	0
Execute	1	0.00	0.00	0	0	0	0
Fetch	3	2.70	3.35	42	990	4	24
total	5	2.75	4.23	120	2613	4	24

Misses in library cache during parse: 1
Optimizer mode: ALL_ROWS
Parsing user id: 92 (SUBHAM)

```
Rows     Row Source Operation
-------  ---------------------------------------------------
     24  NESTED LOOPS  (cr=1210 pr=106 pw=0 time=0 us cost=3 size=37746 card=233)
      2   TABLE ACCESS FULL STATES (cr=9 pr=6 pw=0 time=34 us cost=3 size=188 card=2)
     24   TABLE ACCESS BY INDEX ROWID POINTLM (cr=1201 pr=100 pw=0 time=1650 us cost=3
size=7888 card=116)
    936    DOMAIN INDEX  POINTLM_IDX (cr=483 pr=94 pw=0 time=9132 us cost=0 size=0 card=0)

Rows     Execution Plan
-------  ---------------------------------------------------
      0  SELECT STATEMENT   MODE: ALL_ROWS
     24   NESTED LOOPS
      2    TABLE ACCESS   MODE: ANALYZED (FULL) OF 'STATES' (TABLE)
     24    TABLE ACCESS   MODE: ANALYZED (BY INDEX ROWID) OF 'POINTLM'
            (TABLE)
    936     DOMAIN INDEX OF 'POINTLM_IDX' (INDEX (DOMAIN))
```

Postgres:

```
explain (analyze,buffers) select a.fullname from tiger.pointlm as a, tiger.states as b where ST_ContainsProperly(b.the_geom,a.the_geom) and a.mtfcc='K1231' and b.statefp10='11';
```

```
                              QUERY PLAN
------------------------------------------------------------------------------------------------------------------------
 Nested Loop  (cost=0.00..36.76 rows=8 width=14) (actual time=127.512..322.979 rows=24 loops=1)
   Join Filter: _st_containsproperly(b.the_geom, a.the_geom)
   Buffers: shared hit=890 read=121
   ->  Seq Scan on states b  (cost=0.00..3.30 rows=2 width=1065526) (actual time=17.851..17.923 rows=2 loops=1)
         Filter: ((statefp10)::text = '11'::text)
         Buffers: shared read=2
   ->  Index Scan using pointlm_the_geom_gist on pointlm a  (cost=0.00..16.47 rows=1 width=114) (actual time=28.223..98.370 rows=13 loops=2)
         Index Cond: (b.the_geom && a.the_geom)
         Filter: ((a.mtfcc)::text = 'K1231'::text)
         Buffers: shared hit=383 read=81
 Total runtime: 324.812 ms
```

So here **Oracle 11g** takes **2.75 secs** to process the query whereas Postgres takes only **0.324 secs**.
And the **Hit Ratio=890/(890+121)=0.88** in **Postgres** and **Hit Ratio=(1-120/(2613+4))=0.95** in **Oracle 11g**. Though hit ratio is more in Oracle 11g, in overall **Postgres performs better**. The reason is that Postgres may have found most of the buffers in the Linux system cache to be brought into its own buffer-cache and hence though the **read** parameter gets incremented, it doesn't incur the Disk I/O cost.

1.2 Polygon/Line

Here we queried **List all the linear water bodies in the state District_of_Columbia.**

Oracle 11g:

```
select a.fullname from linearwater a, states b where b.statefp10='11' and sdo_inside(a.geom,b.geom)='TRUE'
```

call	count	cpu	elapsed	disk	query	current	rows
Parse	1	0.06	0.78	78	1620	0	0
Execute	1	0.00	0.00	0	0	0	0
Fetch	19	3.21	3.52	39	490	4	264
total	21	3.28	4.31	117	2110	4	264

Misses in library cache during parse: 1
Optimizer mode: ALL_ROWS
Parsing user id: 92 (SUBHAM)

```
Rows     Row Source Operation
-------  ---------------------------------------------------
    264  NESTED LOOPS  (cr=760 pr=119 pw=0 time=1446 us cost=3 size=15478704 card=113814)
```

```
     2   TABLE ACCESS FULL STATES (cr=9 pr=6 pw=0 time=70 us cost=3 size=188 card=2)
   264   TABLE ACCESS BY INDEX ROWID LINEARWATER (cr=751 pr=113 pw=0 time=5851 us
cost=3 size=2390094 card=56907)
   264    DOMAIN INDEX  LINEARWATER_IDX (cr=515 pr=110 pw=0 time=5633 us cost=0 size=0
card=0)

Rows     Execution Plan
-------  ---------------------------------------------------
     0  SELECT STATEMENT   MODE: ALL_ROWS
   264   NESTED LOOPS
     2    TABLE ACCESS   MODE: ANALYZED (FULL) OF 'STATES' (TABLE)
   264    TABLE ACCESS   MODE: ANALYZED (BY INDEX ROWID) OF
           'LINEARWATER' (TABLE)
   264     DOMAIN INDEX OF 'LINEARWATER_IDX' (INDEX (DOMAIN))
```

Postgres:

```
explain (analyze,buffers) select a.fullname from tiger.linearwater as a,
tiger.states as b where ST_ContainsProperly(b.the_geom,a.the_geom) and
b.statefp10='11';
                                                                  QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------------------------
 Nested Loop  (cost=0.00..254.28 rows=72958 width=12) (actual time=159.376..1598.362 rows=266
loops=1)
   Join Filter: _st_containsproperly(b.the_geom, a.the_geom)
   Buffers: shared hit=18406 read=338
   ->  Seq Scan on states b  (cost=0.00..3.30 rows=2 width=1065526) (actual time=17.070..17.149 rows=2
loops=1)
         Filter: ((statefp10)::text = '11'::text)
         Buffers: shared read=2
   ->  Index Scan using linearwater_the_geom_gist on linearwater a  (cost=0.00..118.14 rows=28
width=3380) (actual time=42.121..496.375 rows=472 loops=2)
         Index Cond: (b.the_geom && a.the_geom)
         Buffers: shared hit=439 read=291
 Total runtime: 1601.035 ms
```

So here **Postgres** takes only **1.6 secs** and **Oracle 11g** takes **3.28 secs.**
And **Hit Ratio=18406/(18406+338)=0.982** in **Postgres** and **Hit Ratio=1-(117/2110+4)=0.945** in **Oracle 11g.**

1.2 Polygon/Polygon

Here we did two queries one with arealm and states and the other with areawater and states. In the query involving **arealm/states** we queried **Retrieve all Shopping Center located in the state California** having stateid of 06. Shopping Center has the mtfcc code of K2361 [7]. We deliberately have chosen the state

California since it is a famous state and will have more shopping centers than any other states. In the other case i.e. query involving **areawater/states** we queried **Retrieve all the Lakes/Ponds in the state District_of_Columbia.** Lakes/Ponds have the mtfcc code of H2030 [7].

A) AREALM/STATES

Oracle 11g:

```
select distinct a.fullname from  arealm a,states b where a.mtfcc='K2361' and
b.statefp10='06' and  sdo_inside(a.geom,b.geom)='TRUE'
```

call	count	cpu	elapsed	disk	query	current	rows
Parse	1	0.17	1.02	93	1620	0	0
Execute	1	0.00	0.00	0	0	0	0
Fetch	10	384.68	390.94	776	15657	4	133
total	12	384.85	391.96	869	17277	4	133

Misses in library cache during parse: 1
Optimizer mode: ALL_ROWS
Parsing user id: 92 (SUBHAM)

```
Rows     Row Source Operation
-------  ---------------------------------------------------
    133  HASH UNIQUE (cr=18007 pr=1095 pw=0 time=0 us cost=4 size=6253 card=37)
    292   NESTED LOOPS  (cr=18007 pr=1095 pw=0 time=6195681 us cost=3 size=6253 card=37)
      2    TABLE ACCESS FULL STATES (cr=7 pr=6 pw=0 time=51 us cost=3 size=188 card=2)
    292    TABLE ACCESS BY INDEX ROWID AREALM (cr=18000 pr=1089 pw=0 time=4633112 us
cost=3 size=1350 card=18)
  15890     DOMAIN INDEX  AREALM_IDX (cr=3220 pr=363 pw=0 time=40098 us cost=0 size=0
card=0)


Rows     Execution Plan
-------  ---------------------------------------------------
      0  SELECT STATEMENT   MODE: ALL_ROWS
    133   HASH (UNIQUE)
    292    NESTED LOOPS
      2    TABLE ACCESS   MODE: ANALYZED (FULL) OF 'STATES' (TABLE)
    292     TABLE ACCESS   MODE: ANALYZED (BY INDEX ROWID) OF 'AREALM'
             (TABLE)
  15890      DOMAIN INDEX OF 'AREALM_IDX' (INDEX (DOMAIN))
```

Postgres:

```
explain (analyze,buffers) select distinct a.fullname from tiger.arealm as
```

```
a,tiger.states as b where ST_ContainsProperly(b.the_geom,a.the_geom) and
a.mtfcc='K2361' and b.statefp10='06';
```

```
                                                                QUERY PLAN
-------------------------------------------------------------------------------------------------------------------------------------------
 Unique  (cost=20.43..20.44 rows=2 width=18) (actual time=3092.205..3092.598 rows=133 loops=1)
   Buffers: shared hit=13303 read=923
   ->  Sort  (cost=20.43..20.44 rows=2 width=18) (actual time=3092.202..3092.334 rows=292 loops=1)
         Sort Key: a.fullname
         Sort Method:  quicksort  Memory: 30kB
         Buffers: shared hit=13303 read=923
         ->  Nested Loop  (cost=0.00..20.42 rows=2 width=18) (actual time=402.202..3089.878 rows=292
loops=1)
               Join Filter: _st_containsproperly(b.the_geom, a.the_geom)
               Buffers: shared hit=13301 read=923
               ->  Seq Scan on states b  (cost=0.00..3.30 rows=2 width=1065526) (actual time=13.591..13.637
rows=2 loops=1)
                     Filter: ((statefp10)::text = '06'::text)
                     Buffers: shared read=2
               ->  Index Scan using arealm_the_geom_gist on arealm a  (cost=0.00..8.30 rows=1 width=3941)
(actual time=176.597..777.269 rows=146 loops=2)
                     Index Cond: (b.the_geom && a.the_geom)
                     Filter: ((a.mtfcc)::text = 'K2361'::text)
                     Buffers: shared hit=5842 read=870
 Total runtime: 3095.388 ms
```

So here **Postgres** takes only **3.1 secs** whereas **Oracle 11g** takes about **385 secs=6 mins! Hit ratio** in **Postgres** is 13303/(13303+923)**=0.94** and Hit Ratio in **Oracle 11g** is (1−869/(17277+4))**=0.95**

B) AREAWATER/STATES

Oracle 11g:

```
select a.fullname from areawater a,states b where a.mtfcc='H2030' and
b.statefp10='11' and  sdo_inside(a.geom,b.geom)='TRUE'
```

call	count	cpu	elapsed	disk	query	current	rows
Parse	1	0.06	0.48	78	1467	0	0
Execute	1	0.00	0.00	0	0	0	0
Fetch	19	34.00	34.27	42	510	4	262
total	21	34.06	34.76	120	1977	4	262

Misses in library cache during parse: 1
Optimizer mode: ALL_ROWS
Parsing user id: 92 (SUBHAM)

```
Rows     Row Source Operation
-------  ---------------------------------------------------
    262  NESTED LOOPS  (cr=1338 pr=146 pw=0 time=1435 us cost=3 size=150880 card=920)
      2   TABLE ACCESS FULL STATES (cr=9 pr=6 pw=0 time=67 us cost=3 size=188 card=2)
    262   TABLE ACCESS BY INDEX ROWID AREAWATER (cr=1329 pr=140 pw=0 time=1040 us
cost=3 size=32200 card=460)
    308    DOMAIN INDEX  AREAWATER_IDX (cr=1048 pr=135 pw=0 time=535 us cost=0 size=0
card=0)

Rows     Execution Plan
-------  ---------------------------------------------------
      0  SELECT STATEMENT   MODE: ALL_ROWS
    262   NESTED LOOPS
      2    TABLE ACCESS   MODE: ANALYZED (FULL) OF 'STATES' (TABLE)
    262    TABLE ACCESS   MODE: ANALYZED (BY INDEX ROWID) OF 'AREAWATER'
               (TABLE)
    308     DOMAIN INDEX OF 'AREAWATER_IDX' (INDEX (DOMAIN))
```

Postgres:

```
explain (analyze,buffers) select a.fullname from tiger.areawater as a,
tiger.states as b where ST_ContainsProperly(b.the_geom,a.the_geom) and
a.mtfcc='H2030' and b.statefp10='11';
```

```
                                   QUERY PLAN
----------------------------------------------------------------------------------------------------------------------------
 Nested Loop  (cost=0.00..37.22 rows=6226 width=12) (actual time=132.221..427.452 rows=262 loops=1)
   Join Filter: _st_containsproperly(b.the_geom, a.the_geom)
   Buffers: shared hit=5126 read=96

   ->  Seq Scan on states b  (cost=0.00..3.30 rows=2 width=1065526) (actual time=20.698..20.748 rows=2
loops=1)
         Filter: ((statefp10)::text = '11'::text)
         Buffers: shared read=2

   ->  Index Scan using areawater_the_geom_gist on areawater a  (cost=0.00..16.43 rows=2 width=3288)
(actual time=32.486..64.231 rows=132 loops=2)
         Index Cond: (b.the_geom && a.the_geom)
         Filter: ((a.mtfcc)::text = 'H2030'::text)
         Buffers: shared hit=67 read=43

 Total runtime: 430.503 ms
```

So here **Postgres** takes only **0.43 secs** whereas **Oracle 11g** takes about **34 secs**. And **Hit Ratio** in **Postgres** is 5126/(5126+96)=**0.98** and Hit Ratio in **Oracle 11g** is (1-120/(1977+4))=**0.94**

2. ADJACENT OPERATIONS

In adjacent operation we see whether two geometric objects touches or not, i.e. their boundaries intersect or not. We have given three categories of queries Polygon/Polygon, Polygon/Line, Line/Line.

2.1 Polygon/Polygon

Here we queried **Retrieve all the states which are adjacent to a particular state**. We have chosen **Nebraska** having stateid of 31, since it is located in the middle of the US map and will have more states surrounding it.

Oracle 11g:

```
select a.name10 from states a,states b where b.statefp10='31' and sdo_touch(a.geom,b.geom)='TRUE'
```

```
call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        1      0.04       1.41         89       1620          0           0
Execute      1      0.00       0.00          0          0          0           0
Fetch        3     62.35      62.87         32        716          4          24
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total        5     62.40      64.28        121       2336          4          24

Misses in library cache during parse: 1
Optimizer mode: ALL_ROWS
Parsing user id: 92  (SUBHAM)

Rows     Row Source Operation
-------  ---------------------------------------------------
     24  NESTED LOOPS  (cr=11488 pr=400 pw=0 time=0 us cost=3 size=376 card=2)
      2   TABLE ACCESS FULL STATES (cr=9 pr=6 pw=0 time=44 us cost=3 size=188 card=2)
     24   TABLE ACCESS BY INDEX ROWID STATES (cr=11479 pr=394 pw=0 time=1474 us cost=3
size=94 card=1)
     24    DOMAIN INDEX  STATES_IDX (cr=11455 pr=394 pw=0 time=1144 us cost=0 size=0 card=0)

Rows     Execution Plan
-------  ---------------------------------------------------
      0  SELECT STATEMENT   MODE: ALL_ROWS
     24   NESTED LOOPS
      2    TABLE ACCESS   MODE: ANALYZED (FULL) OF 'STATES' (TABLE)
     24    TABLE ACCESS   MODE: ANALYZED (BY INDEX ROWID) OF 'STATES'
             (TABLE)
     24     DOMAIN INDEX OF 'STATES_IDX' (INDEX (DOMAIN))
```

Postgres:

```
explain (analyze,buffers) select a.name10 from tiger.states a,tiger.states b
where ST_Touches(a.the_geom,b.the_geom) and b.statefp10='31';
```

QUERY PLAN

Nested Loop (cost=0.00..16.36 rows=3 width=9) (actual time=160.961..3383.322 rows=24 loops=1)
Join Filter: _st_touches(a.the_geom, b.the_geom)
Buffers: shared hit=1447 read=470

-> Seq Scan on states b (cost=0.00..3.30 rows=2 width=1065526) (actual time=14.253..14.298 rows=2 loops=1)
Filter: ((statefp10)::text = '31'::text)
Buffers: shared hit=1 read=1

-> Index Scan using states_the_geom_gist on states a (cost=0.00..6.27 rows=1 width=1065535) (actual time=6.271..6.405 rows=14 loops=2)
Index Cond: (a.the_geom && b.the_geom)
Buffers: shared hit=4 read=2

Total runtime: 3383.647 ms

So here **Postgres** takes **3.4 secs** whereas **Oracle 11g** takes **62.4 secs**. **Hit Ratio** in **Postgre**s is 1447/(1447+470)=**0.75** and Hit Ratio in **Oracle 11g** is (1-121/(2336+4))=**0.95**

2.2 Polygon/Line

Here we queried **Retrieve all polygons from arealm which are adjacent to Yukon River.**

Oracle 11g:

```
select a.fullname from arealm a,linearwater b where b.fullname='Yukon Riv' and
sdo_touch(a.geom, b.geom)='TRUE'
```

call	count	cpu	elapsed	disk	query	current	rows
Parse	1	0.03	1.07	77	1620	0	0
Execute	1	0.00	0.00	0	0	0	0
Fetch	1	3.84	64.16	547858	547941	10	0
total	3	3.87	65.24	547935	549561	10	0

Misses in library cache during parse: 1
Optimizer mode: ALL_ROWS
Parsing user id: 92 (SUBHAM)

```
Rows     Row Source Operation
-------  ---------------------------------------------------
     0  NESTED LOOPS  (cr=548274 pr=547906 pw=0 time=0 us cost=150166 size=827190 card=7070)
     5   TABLE ACCESS FULL LINEARWATER (cr=547862 pr=547856 pw=0 time=283484 us
cost=150166 size=252 card=6)
     0   TABLE ACCESS BY INDEX ROWID AREALM (cr=412 pr=50 pw=0 time=0 us cost=150166
size=86700 card=1156)
     0    DOMAIN INDEX  AREALM_IDX (cr=412 pr=50 pw=0 time=0 us cost=0 size=0 card=0)

Rows     Execution Plan
-------  ---------------------------------------------------
     0  SELECT STATEMENT   MODE: ALL_ROWS
     0   NESTED LOOPS
     5    TABLE ACCESS   MODE: ANALYZED (FULL) OF 'LINEARWATER' (TABLE)
     0    TABLE ACCESS   MODE: ANALYZED (BY INDEX ROWID) OF 'AREALM'
               (TABLE)
     0     DOMAIN INDEX OF 'AREALM_IDX' (INDEX (DOMAIN))
```

Postgres:

```
explain (analyze,buffers) select a.fullname from tiger.arealm a, tiger.linearwater b where ST_Touches(a.the_geom,b.the_geom) and b.fullname='Yukon Riv';
```

```
                              QUERY PLAN
------------------------------------------------------------------------------------------------------------------------
 Nested Loop  (cost=0.00..621112.57 rows=144 width=18) (actual time=156571.787..156571.787 rows=0
loops=1)
   Join Filter: _st_touches(a.the_geom, b.the_geom)
   Buffers: shared hit=39893 read=557578

   ->  Seq Scan on linearwater b  (cost=0.00..619321.72 rows=213 width=3368) (actual
time=6717.710..156324.597 rows=5 loops=1)
         Filter: ((fullname)::text = 'Yukon Riv'::text)
         Buffers: shared hit=39848 read=557547

   ->  Index Scan using arealm_the_geom_gist on arealm a  (cost=0.00..8.15 rows=1 width=3941) (actual
time=27.353..31.559 rows=2 loops=5)
         Index Cond: (a.the_geom && b.the_geom)
         Buffers: shared hit=15 read=9

 Total runtime: 156635.123 ms
```

Though we got zero results here, we wanted to note the time needed to complete the query. Surprisingly, here **Oracle 11g** performs exceptionally well, may be they are doing any kind of optimisation internally. It took only **3.87 secs** to process whereas **Postgres** took about **157 secs** to process the same query. **Hit ratio** in **Oracle 11g**=(1−547935/(549561+10))=**0.003** and in **Postgres** it is 39893/(39893+557578)=**0.067**

2.3 Line/Line

Here we queried **Retrieve all the tributaries of Yukon River**.

Oracle 11g:

```
select a.fullname from linearwater a,linearwater b where b.fullname='Yukon
Riv' and sdo_touch(a.geom,b.geom)='TRUE'
```

```
call     count       cpu    elapsed       disk      query      current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        1      0.03       1.05         89       1623          0           0
Execute      1      0.00       0.00          0          0          0           0
Fetch        9      6.59      68.38     547858     548156         10         109
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total       11      6.62      69.43     547947     549779         10         109

Misses in library cache during parse: 1
Optimizer mode: ALL_ROWS
Parsing user id: 92  (SUBHAM)

Rows     Row Source Operation
-------  ---------------------------------------------------
   109  NESTED LOOPS  (cr=549118 pr=548223 pw=0 time=0 us cost=150166 size=29222004
card=347881)
     5   TABLE ACCESS FULL LINEARWATER (cr=547865 pr=547856 pw=0 time=239472 us
cost=150166 size=252 card=6)
   109   TABLE ACCESS BY INDEX ROWID LINEARWATER (cr=1253 pr=367 pw=0 time=624 us
cost=150166 size=2390094 card=56907)
   109    DOMAIN INDEX  LINEARWATER_IDX (cr=1145 pr=367 pw=0 time=260 us cost=0 size=0
card=0)

Rows     Execution Plan
-------  ---------------------------------------------------
     0  SELECT STATEMENT   MODE: ALL_ROWS
   109   NESTED LOOPS
     5    TABLE ACCESS   MODE: ANALYZED (FULL) OF 'LINEARWATER' (TABLE)
   109    TABLE ACCESS   MODE: ANALYZED (BY INDEX ROWID) OF
           'LINEARWATER' (TABLE)
   109     DOMAIN INDEX OF 'LINEARWATER_IDX' (INDEX (DOMAIN))
```

Postgres:

```
explain (analyze,buffers) select a.fullname from tiger.linearwater
a,tiger.linearwater b where ST_Touches(a.the_geom,b.the_geom) and
b.fullname='Yukon Riv';
```

QUERY PLAN

Nested Loop (cost=0.00..645923.01 rows=284 width=12) (actual time=6901.882..152981.262 rows=109 loops=1)
Join Filter: _st_touches(a.the_geom, b.the_geom)
Buffers: shared hit=40451 read=557683

-> Seq Scan on linearwater b (cost=0.00..619321.72 rows=213 width=3368) (actual time=6801.689..151094.379 rows=5 loops=1)
Filter: ((fullname)::text = 'Yukon Riv'::text)
Buffers: shared hit=40137 read=557258

-> Index Scan using linearwater_the_geom_gist on linearwater a (cost=0.00..117.54 rows=28 width=3380) (actual time=19.395..165.870 rows=160 loops=5)
Index Cond: (a.the_geom && b.the_geom)
Buffers: shared hit=286 read=391

Total runtime: 153009.282 ms

Here also Oracle 11g takes less time than Postgres. **Oracle 11g** takes only **6.62 secs** whereas **Postgres** takes about **153 secs** to process the same query. **Hit Ratio** for **Postgres**=40451/(40451+557683)=**0.067** and Hit Ratio for **Oracle 11g**=(1-547947/(549779+10))=**0.003**

3. INTERSECT OPERATION

Intersect Operation is basically complement of the Disjoint Operation. It can also be termed as **any kind of interaction**. Here we have given queries involving Polygon and Line only but one involving less number of tuples than the other one. In the first query, **states and primaryroads** relations are used, both of which are small relations, whereas in the 2nd query we used **roads and arealm** relations which are very big.

3.1 Polygon/Line Small

Here we queried **Retrieve all the interstate connector roads in the state California**. That is, the query retrieves all those roads which connect California to other states. Interstate connector roads are stored in the relation **primaryroads**.

Oracle 11g:

```
select a.fullname from primaryroads a,states b where b.statefp10='06' and
sdo_anyinteract(a.geom,  b.geom)='TRUE'
```

call	count	cpu	elapsed	disk	query	current	rows
Parse	2	0.06	0.78	80	1466	0	0
Execute	2	0.00	0.00	0	0	0	0
Fetch	574	449.20	460.00	537	8411	8	8552
total	578	449.26	460.79	617	9877	8	8552

```
Misses in library cache during parse: 1
Optimizer mode: ALL_ROWS
Parsing user id: 92  (SUBHAM)

Rows     Row Source Operation
-------  ---------------------------------------------------
   4276  NESTED LOOPS  (cr=6328 pr=526 pw=0 time=45141152 us cost=3 size=62133 card=447)
      2   TABLE ACCESS FULL STATES (cr=9 pr=6 pw=0 time=50 us cost=3 size=188 card=2)
   4276   TABLE ACCESS BY INDEX ROWID PRIMARYROADS (cr=6319 pr=520 pw=0
time=43805232 us cost=3 size=10080 card=224)
   4276    DOMAIN INDEX  PRIMARYROADS_IDX (cr=2773 pr=309 pw=0 time=25559008 us cost=0
size=0 card=0)

Rows     Execution Plan
-------  ---------------------------------------------------
      0  SELECT STATEMENT   MODE: ALL_ROWS
   4276   NESTED LOOPS
      2    TABLE ACCESS   MODE: ANALYZED (FULL) OF 'STATES' (TABLE)
   4276    TABLE ACCESS   MODE: ANALYZED (BY INDEX ROWID) OF
           'PRIMARYROADS' (TABLE)
   4276     DOMAIN INDEX OF 'PRIMARYROADS_IDX' (INDEX (DOMAIN))
```

Postgres:

```
explain (analyze,buffers) select a.fullname from tiger.primaryroads a,
tiger.states b where ST_Intersects(a.the_geom,b.the_geom) and
b.statefp10='06';
                                                                  QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------------------------
 Nested Loop  (cost=0.00..20.37 rows=287 width=10) (actual time=104.224..22269.064 rows=4276
loops=1)
   Join Filter: _st_intersects(a.the_geom, b.the_geom)
   Buffers: shared hit=122802 read=1102
   ->  Seq Scan on states b  (cost=0.00..3.30 rows=2 width=1065526) (actual time=17.316..17.375 rows=2
loops=1)
         Filter: ((statefp10)::text = '06'::text)
         Buffers: shared read=2
   ->  Index Scan using primaryroads_the_geom_gist on primaryroads a  (cost=0.00..8.27 rows=1
width=7568) (actual time=15.451..581.963 rows=2359 loops=2)
         Index Cond: (a.the_geom && b.the_geom)
         Buffers: shared hit=1504 read=740
 Total runtime: 22278.777 ms
```

Here **Postgres** takes only **22 secs** to process whereas **Oracle 11g** takes about **449 secs=7 mins** to complete. **Hit Ratio** in **Postgres** is 122802/(122802+1102)=**0.99** and in **Oracle 11g** it is (1-617/(9877+8))=**0.94**

3.2 Polygon/Line Large

Here we queried **Retrieve all roads which go through forest**. Forest has the mtfcc code of K2182 [7].

Oracle 11g:

```
select distinct a.fullname from roads a, arealm b where b.mtfcc='K2182' and
sdo_anyinteract(a.geom,b.geom)='TRUE'
```

call	count	cpu	elapsed	disk	query	current	rows
Parse	1	0.09	1.20	80	1620	0	0
Execute	1	0.00	0.01	0	0	0	0
Fetch	322	687.42	758.95	13002	82831	2676	4801
total	324	687.51	760.17	13082	84451	2676	4801

Misses in library cache during parse: 1
Optimizer mode: ALL_ROWS
Parsing user id: 92 (SUBHAM)

```
Rows     Row Source Operation
-------  ---------------------------------------------------
   4801  HASH UNIQUE (cr=169879 pr=22771 pw=0 time=2464 us cost=7221833 size=286147584
card=2345472)
  23658   NESTED LOOPS  (cr=169879 pr=22771 pw=0 time=619509952 us cost=2121
size=45334994552 card=371598316)
   1338    TABLE ACCESS FULL AREALM (cr=7808 pr=7805 pw=0 time=2883301 us cost=2121
size=137700 card=1836)
  23658    TABLE ACCESS BY INDEX ROWID ROADS (cr=162071 pr=14966 pw=0 time=25129834 us
cost=2121 size=9514398 card=202434)
  23658     DOMAIN INDEX  ROADS_IDX (cr=139541 pr=11862 pw=0 time=170320 us cost=0 size=0
card=0)
```

```
Rows     Execution Plan
-------  ---------------------------------------------------
      0  SELECT STATEMENT   MODE: ALL_ROWS
   4801   HASH (UNIQUE)
  23658    NESTED LOOPS
   1338     TABLE ACCESS   MODE: ANALYZED (FULL) OF 'AREALM' (TABLE)
  23658     TABLE ACCESS   MODE: ANALYZED (BY INDEX ROWID) OF 'ROADS'
            (TABLE)
  23658      DOMAIN INDEX OF 'ROADS_IDX' (INDEX (DOMAIN))
```

Postgres:

```
explain (analyze,buffers) select distinct a.fullname from tiger.roads a,tiger.arealm b where ST_Intersects(a.the_geom,b.the_geom) and b.mtfcc='K2182';
```

QUERY PLAN

```
 HashAggregate  (cost=631284.12..631518.73 rows=23461 width=12) (actual time=156297.692..156302.465 rows=4635 loops=1)
   Buffers: shared hit=1628315 read=33932
   ->  Nested Loop  (cost=0.00..630865.74 rows=167350 width=12) (actual time=376.758..156078.764 rows=22968 loops=1)
         Join Filter: _st_intersects(a.the_geom, b.the_geom)
         Buffers: shared hit=1627277 read=33880
         ->  Seq Scan on arealm b  (cost=0.00..9292.58 rows=1426 width=3923) (actual time=62.112..2538.040 rows=1338 loops=1)
               Filter: ((mtfcc)::text = 'K2182'::text)
               Buffers: shared read=7847
         ->  Index Scan using roads_the_geom_gist on roads a  (cost=0.00..409.37 rows=101 width=1294) (actual time=6.135..37.893 rows=71 loops=1338)
               Index Cond: (a.the_geom && b.the_geom)
               Buffers: shared hit=45557 read=21993
 Total runtime: 156306.299 ms
```

There is a few discrepancy in the number of rows returned between 4801 & 4635. This may be due to SRID. Though both the database have been loaded using the same SRID 4326.
Here **Postgres** takes only **156 secs** and **Oracle 11g** takes **687 secs=11 mins** to execute the entire query. **Hit Ratio** in **Postgres** is 1628315/(1628315+33932)=**0.98** and Hit Ratio in **Oracle 11g** is (1 - (13082/(84451+2676)))=**0.85**

4.2 WARM PHASE

In the warm phase, we basically compared the memory management capabilities and page replacement policies followed by the two databases. To make that happen, a set related queries are executed in succession. We designed a C program called the query generator which generated 80 such related queries both for the Postgres and Oracle 11g and in the same order. Following is the C code:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <string.h>
```

```
int main()
{
FILE *fp,*fo;
char stateid[][10]={"'06'","'12'","'13'","'17'","'25'","'36'","'53'","'48'"};
char tables[][50]={"arealm","areawater","linearwater","pointlm"};
char p1[]="explain (analyze,buffers) select distinct a.fullname from tiger.";
char p2[]=" a,tiger.states b where b.statefp10=" ;
char p3[]=" and ST_Intersects(a.the_geom,b.the_geom);\n";
char o1[]="select distinct a.fullname from ";
char o2[]=" a,states b where b.statefp10=";
char o3[]=" and sdo_anyinteract(a.geom,b.geom)='TRUE';\n";
int bs[]={0,0,0,0,0,0,0,0};
int bt[]={0,0,0,0,0};
int globc=0,cs=0,ct=0,rt,rs;
int flag1=0,flag2=0;
int s,t,i;
char tp[4000],to[4000];

fp=fopen("postgres.sql","w");
fo=fopen("oracle.sql","w");
srand(time(NULL));
while(1)
        {
        t=rand()%4;
        if(bt[t])
                {
                t=(t+1)%4;
                while(bt[t]==1)
                        t=(t+1)%4;
                }
        bt[t]=1;
        strcpy(tp,p1);
        strcat(tp,tables[t]);
```

```
strcat(tp,p2);
strcpy(to,o1);
strcat(to,tables[t]);
strcat(to,o2);
ct++;
if(ct==4)
        {
        for(i=0;i<4;i++)
                bt[i]=0;
        ct=0;
        }
s=rand()%8;
if(bs[s])
        {
        s=(s+1)%8;
        while(bs[s]==1)
                s=(s+1)%8;
        }
bs[s]=1;
strcat(tp,stateid[s]);
strcat(tp,p3);
strcat(to,stateid[s]);
strcat(to,o3);
cs++;
if(cs==8)
        {
        for(i=0;i<8;i++)
                bs[i]=0;
        cs=0;
        }
fputs(tp,fp);
fputs(to,fo);
globc++;
```

```
        if(globc==80)
                break;
        }

fclose(fp);
fclose(fo);
return 0;
}
```

The above code ensures that no two same queries are generated one after the another. This is required because if two same queries are executed one after the another then all the index blocks will be present in the memory and we will not get the required performance result. The idea here is to wipe out some of the previous index blocks from the memory while executing the current one. The following is the output of the program. Since the result is huge, we will list here some of them in the order they got generated and will give the performance results for the last four queries only since four features arealm, areawater, linearwater, pointlm have been used for query generation.

Postgres:

explain (analyze,buffers) select distinct a.fullname from tiger.arealm a, tiger.states b where b.statefp10='17' and ST_Intersects(a.the_geom,b.the_geom);

explain (analyze,buffers) select distinct a.fullname from tiger.linearwater a, tiger.states b where b.statefp10='12' and ST_Intersects(a.the_geom,b.the_geom);

explain (analyze,buffers) select distinct a.fullname from tiger.areawater a, tiger.states b where b.statefp10='25' and ST_Intersects(a.the_geom,b.the_geom);

explain (analyze,buffers) select distinct a.fullname from tiger.pointlm a, tiger.states b where b.statefp10='53' and ST_Intersects(a.the_geom,b.the_geom);

explain (analyze,buffers) select distinct a.fullname from tiger.linearwater a, tiger.states b where b.statefp10='48' and ST_Intersects(a.the_geom,b.the_geom);

explain (analyze,buffers) select distinct a.fullname from tiger.pointlm a, tiger.states b where b.statefp10='13' and ST_Intersects(a.the_geom,b.the_geom);

explain (analyze,buffers) select distinct a.fullname from tiger.areawater a, tiger.states b where b.statefp10='06' and ST_Intersects(a.the_geom,b.the_geom);

explain (analyze,buffers) select distinct a.fullname from tiger.arealm a, tiger.states b where b.statefp10='36' and ST_Intersects(a.the_geom,b.the_geom);

.
.
.

explain (analyze,buffers) select distinct a.fullname from tiger.linearwater a, tiger.states b where

b.statefp10='13' and ST_Intersects(a.the_geom,b.the_geom);

explain (analyze,buffers) select distinct a.fullname from tiger.arealm a, tiger.states b where b.statefp10='12' and ST_Intersects(a.the_geom,b.the_geom);

explain (analyze,buffers) select distinct a.fullname from tiger.areawater a, tiger.states b where b.statefp10='25' and ST_Intersects(a.the_geom,b.the_geom);

explain (analyze,buffers) select distinct a.fullname from tiger.pointlm a, tiger.states b where b.statefp10='17' and ST_Intersects(a.the_geom,b.the_geom);

explain (analyze,buffers) select distinct a.fullname from tiger.areawater a, tiger.states b where b.statefp10='36' and ST_Intersects(a.the_geom,b.the_geom);

explain (analyze,buffers) select distinct a.fullname from tiger.linearwater a, tiger.states b where b.statefp10='53' and ST_Intersects(a.the_geom,b.the_geom);

explain (analyze,buffers) select distinct a.fullname from tiger.pointlm a, tiger.states b where b.statefp10='48' and ST_Intersects(a.the_geom,b.the_geom);

explain (analyze,buffers) select distinct a.fullname from tiger.arealm a, tiger.states b where b.statefp10='06' and ST_Intersects(a.the_geom,b.the_geom);

Oracle 11g:

Before running the oracle.sql file we **enabled** the **tracing**.

select distinct a.fullname from arealm a,states b where b.statefp10='17' and sdo_anyinteract(a.geom,b.geom)='TRUE';

select distinct a.fullname from linearwater a,states b where b.statefp10='12' and sdo_anyinteract(a.geom,b.geom)='TRUE';

select distinct a.fullname from areawater a,states b where b.statefp10='25' and sdo_anyinteract(a.geom,b.geom)='TRUE';

select distinct a.fullname from pointlm a,states b where b.statefp10='53' and sdo_anyinteract(a.geom,b.geom)='TRUE';

select distinct a.fullname from linearwater a,states b where b.statefp10='48' and sdo_anyinteract(a.geom,b.geom)='TRUE';

select distinct a.fullname from pointlm a,states b where b.statefp10='13' and sdo_anyinteract(a.geom,b.geom)='TRUE';

select distinct a.fullname from areawater a,states b where b.statefp10='06' and sdo_anyinteract(a.geom,b.geom)='TRUE';

select distinct a.fullname from arealm a,states b where b.statefp10='36' and sdo_anyinteract(a.geom,b.geom)='TRUE';

.
.
.

select distinct a.fullname from linearwater a,states b where b.statefp10='13' and

sdo_anyinteract(a.geom,b.geom)='TRUE';

select distinct a.fullname from arealm a,states b where b.statefp10='12' and sdo_anyinteract(a.geom,b.geom)='TRUE';

select distinct a.fullname from areawater a,states b where b.statefp10='25' and sdo_anyinteract(a.geom,b.geom)='TRUE';

select distinct a.fullname from pointlm a,states b where b.statefp10='17' and sdo_anyinteract(a.geom,b.geom)='TRUE';

alter session set tracefile_identifier=spa2; **/* Added a Trace File Identifier */**

select distinct a.fullname from areawater a,states b where b.statefp10='36' and sdo_anyinteract(a.geom,b.geom)='TRUE';

select distinct a.fullname from linearwater a,states b where b.statefp10='53' and sdo_anyinteract(a.geom,b.geom)='TRUE';

select distinct a.fullname from pointlm a,states b where b.statefp10='48' and sdo_anyinteract(a.geom,b.geom)='TRUE';

select distinct a.fullname from arealm a,states b where b.statefp10='06' and sdo_anyinteract(a.geom,b.geom)='TRUE';

N.B.: To measure the performance of the four queries we have **added** a **new trace file identifier** spa2.

QUERY 1

Oracle 11g:

```
select distinct a.fullname from areawater a,states b where b.statefp10='36'
and sdo_anyinteract(a.geom, b.geom)='TRUE'
```

call	count	cpu	elapsed	disk	query	current	rows
Parse	1	0.00	0.02	0	0	0	0
Execute	1	0.00	0.00	0	0	0	0
Fetch	2	1545.37	1548.40	17	537	4	13
total	4	1545.37	1548.42	17	537	4	13

Misses in library cache during parse: 0
Optimizer mode: ALL_ROWS
Parsing user id: 92 (SUBHAM)

Rows	Row Source Operation
13	HASH UNIQUE (cr=7819 pr=550 pw=0 time=0 us cost=372 size=1660336 card=10124)

```
    40   NESTED LOOPS  (cr=7819 pr=550 pw=0 time=1443 us cost=3 size=1660336 card=10124)
     2   TABLE ACCESS FULL STATES (cr=7 pr=0 pw=0 time=38 us cost=3 size=188 card=2)
    40    TABLE ACCESS BY INDEX ROWID AREAWATER (cr=7812 pr=550 pw=0 time=1368 us
cost=3 size=354340 card=5062)
    40     DOMAIN INDEX  AREAWATER_IDX (cr=7772 pr=550 pw=0 time=1140 us cost=0 size=0
card=0)

Rows     Execution Plan
-------  ---------------------------------------------------
      0  SELECT STATEMENT   MODE: ALL_ROWS
     13   HASH (UNIQUE)
     40    NESTED LOOPS
      2    TABLE ACCESS   MODE: ANALYZED (FULL) OF 'STATES' (TABLE)
     40     TABLE ACCESS   MODE: ANALYZED (BY INDEX ROWID) OF
              'AREAWATER' (TABLE)
     40      DOMAIN INDEX OF 'AREAWATER_IDX' (INDEX (DOMAIN))
```

Postgres:

explain (analyze,buffers) select distinct a.fullname from tiger.areawater a, tiger.states b where b.statefp10='36' and ST_Intersects(a.the_geom,b.the_geom);

```
                                   QUERY PLAN
------------------------------------------------------------------------------------------------------------------------

 HashAggregate  (cost=53.95..73.56 rows=1961 width=12) (actual time=2879.228..2879.243 rows=13
loops=1)
   Buffers: shared hit=117295 read=67

   ->  Nested Loop  (cost=0.00..37.73 rows=6490 width=12) (actual time=282.130..2879.053 rows=40
loops=1)
         Join Filter: _st_intersects(a.the_geom, b.the_geom)
         Buffers: shared hit=117295 read=67

         ->  Seq Scan on states b  (cost=0.00..3.30 rows=2 width=1065526) (actual time=0.020..0.050 rows=2
loops=1)
               Filter: ((statefp10)::text = '36'::text)
               Buffers: shared hit=2

         ->  Index Scan using areawater_the_geom_gist on areawater a  (cost=0.00..16.43 rows=3
width=3288) (actual time=0.068..12.497 rows=2384 loops=2)
               Index Cond: (a.the_geom && b.the_geom)
               Buffers: shared hit=2111 read=9

 Total runtime: 2948.711 ms
```

Here **Postgres** takes only **3 secs** whereas **Oracle 11g** takes about 1545/60=**26 mins!** to execute the same query. **Hit Ratio** for **Postgres** is 117295/(117295+67)=**0.99** and in **Oracle 11g** it is (1-(17/(537+4)))=**0.97**

QUERY 2

Oracle 11g:

```
select distinct a.fullname from linearwater a,states b where b.statefp10='53'
and sdo_anyinteract(a.geom, b.geom)='TRUE'
```

```
call     count       cpu    elapsed       disk      query          current        rows
------- ------  -------- ---------- ---------- ----------  ----------  ----------
Parse        1      0.00       0.00          0          0           0           0
Execute      1      0.00       0.00          0          0           0           0
Fetch      284    246.15     283.32       9315     194261           4        4245
------- ------  -------- ---------- ---------- ----------  ----------  ----------
total      286    246.15     283.32       9315     194261           4        4245

Misses in library cache during parse: 0
Optimizer mode: ALL_ROWS
Parsing user id: 92  (SUBHAM)

Rows     Row Source Operation
-------  ---------------------------------------------------
   4245  HASH UNIQUE (cr=202941 pr=11311 pw=0 time=1929 us cost=3385 size=15018752
card=110432)
 196428   NESTED LOOPS  (cr=202941 pr=11311 pw=0 time=47318560 us cost=3 size=15478704
card=113814)
      2    TABLE ACCESS FULL STATES (cr=7 pr=0 pw=0 time=35 us cost=3 size=188 card=2)
 196428    TABLE ACCESS BY INDEX ROWID LINEARWATER (cr=202934 pr=11311 pw=0
time=47212960 us cost=3 size=2390094 card=56907)
 196428     DOMAIN INDEX  LINEARWATER_IDX (cr=13730 pr=2022 pw=0 time=160619 us cost=0
size=0 card=0)

Rows     Execution Plan
-------  ---------------------------------------------------
      0  SELECT STATEMENT   MODE: ALL_ROWS
   4245   HASH (UNIQUE)
 196428    NESTED LOOPS
      2     TABLE ACCESS   MODE: ANALYZED (FULL) OF 'STATES' (TABLE)
 196428      TABLE ACCESS   MODE: ANALYZED (BY INDEX ROWID) OF
              'LINEARWATER' (TABLE)
 196428       DOMAIN INDEX OF 'LINEARWATER_IDX' (INDEX (DOMAIN))
```

Postgres:

```
explain (analyze,buffers) select distinct a.fullname from tiger.linearwater a,
tiger.states b where b.statefp10='53' and ST_Intersects(a.the_geom,b.the_geom);
```

QUERY PLAN

HashAggregate (cost=436.67..467.87 rows=3120 width=12) (actual time=97567.370..97570.036 rows=4243 loops=1)
Buffers: shared hit=3796624 read=46038

-> Nested Loop (cost=0.00..254.28 rows=72958 width=12) (actual time=49.342..97008.239 rows=196368 loops=1)
Join Filter: _st_intersects(a.the_geom, b.the_geom)
Buffers: shared hit=3786530 read=45712

-> Seq Scan on states b (cost=0.00..3.30 rows=2 width=1065526) (actual time=0.036..0.072 rows=2 loops=1)
Filter: ((statefp10)::text = '53'::text)
Buffers: shared hit=1 read=1

-> Index Scan using linearwater_the_geom_gist on linearwater a (cost=0.00..118.14 rows=28 width=3380) (actual time=19.515..7159.395 rows=105391 loo
ps=2)
Index Cond: (a.the_geom && b.the_geom)
Buffers: shared hit=92110 read=42174

Total runtime: 97632.240 ms

Here **Postgres** took about **98 secs** whereas **Oracle 11g** took **246 secs** to process the same query. **Hit Ratio** in **Postgres** is 3796624/(3796624+46038)=**0.98** whereas in **Oracle 11g** it is (1-(9315/(194261+ 4)))=**0.95**

QUERY 3

Oracle 11g:

```
select distinct a.fullname from pointlm a,states b where b.statefp10='48' and sdo_anyinteract(a.geom,b.geom)='TRUE'
```

call	count	cpu	elapsed	disk	query	current	rows
Parse	1	0.00	0.00	0	0	0	0
Execute	1	0.00	0.00	0	0	0	0
Fetch	751	245.60	246.47	482	33731	4	11236
total	753	245.60	246.47	482	33731	4	11236

Misses in library cache during parse: 0
Optimizer mode: ALL_ROWS
Parsing user id: 92 (SUBHAM)

```
Rows     Row Source Operation
-------  ---------------------------------------------------
  11236  HASH UNIQUE (cr=35283 pr=889 pw=0 time=3874 us cost=490 size=2187648 card=13504)
  59168   NESTED LOOPS  (cr=35283 pr=889 pw=0 time=203300368 us cost=3 size=2187648
card=13504)
      2    TABLE ACCESS FULL STATES (cr=7 pr=0 pw=0 time=25 us cost=3 size=188 card=2)
  59168    TABLE ACCESS BY INDEX ROWID POINTLM (cr=35276 pr=889 pw=0 time=202847184 us
cost=3 size=459136 card=6752)
  59168     DOMAIN INDEX  POINTLM_IDX (cr=7290 pr=603 pw=0 time=200291952 us cost=0 size=0
card=0)

Rows     Execution Plan
-------  ---------------------------------------------------
      0  SELECT STATEMENT   MODE: ALL_ROWS
  11236   HASH (UNIQUE)
  59168    NESTED LOOPS
      2     TABLE ACCESS   MODE: ANALYZED (FULL) OF 'STATES' (TABLE)
  59168     TABLE ACCESS   MODE: ANALYZED (BY INDEX ROWID) OF 'POINTLM'
                (TABLE)
  59168      DOMAIN INDEX OF 'POINTLM_IDX' (INDEX (DOMAIN))
```

Postgres:

```
explain (analyze,buffers) select distinct a.fullname from tiger.pointlm a,tiger.states
b where b.statefp10='48' and ST_Intersects(a.the_geom,b.the_geom);
```

```
                                        QUERY PLAN
------------------------------------------------------------------------------------------------------------------------

 HashAggregate  (cost=59.44..146.01 rows=8657 width=14) (actual time=370445.935..370452.941
rows=11236 loops=1)
   Buffers: shared hit=12062959 read=1309

   ->  Nested Loop  (cost=0.00..37.80 rows=8657 width=14) (actual time=535.327..370094.733 rows=59168
loops=1)
         Join Filter: _st_intersects(a.the_geom, b.the_geom)
         Buffers: shared hit=12062959 read=1309

         ->  Seq Scan on states b  (cost=0.00..3.30 rows=2 width=1065526) (actual time=0.042..0.091 rows=2
loops=1)
               Filter: ((statefp10)::text = '48'::text)
               Buffers: shared read=2

         ->  Index Scan using pointlm_the_geom_gist on pointlm a  (cost=0.00..16.46 rows=3 width=114)
(actual time=14.479..857.048 rows=49394 loops=2)
               Index Cond: (a.the_geom && b.the_geom)
               Buffers: shared hit=10820 read=1066

 Total runtime: 370459.456 ms
```

Here **Postgres** takes 370/60=**6 mins** whereas **surprisingly** here **Oracle 11g** takes only 246/60=**4 mins** to execute the same query. **Hit Ratio** in **Postgres** is 12062959 /(12062959+1309)=**0.99** and in **Oracle 11g** it is (1 – 482/(33731+4))=**0.99**

QUERY 4

Oracle 11g:

```
select distinct a.fullname from arealm a,states b where b.statefp10='06' and sdo_anyinteract(a.geom,b.geom)='TRUE'
```

call	count	cpu	elapsed	disk	query	current	rows
Parse	1	0.00	0.00	0	0	0	0
Execute	1	0.00	0.00	0	0	0	0
Fetch	368	126.60	127.35	766	15705	4	5501
total	370	126.60	127.35	766	15705	4	5501

Misses in library cache during parse: 0
Optimizer mode: ALL_ROWS
Parsing user id: 92 (SUBHAM)

```
Rows     Row Source Operation
-------  ---------------------------------------------------
   5501  HASH UNIQUE (cr=18013 pr=1056 pw=0 time=1571 us cost=92 size=390897 card=2313)
  15910   NESTED LOOPS  (cr=18013 pr=1056 pw=0 time=3170854 us cost=3 size=390897 card=2313)
      2    TABLE ACCESS FULL STATES (cr=7 pr=0 pw=0 time=43 us cost=3 size=188 card=2)
  15910    TABLE ACCESS BY INDEX ROWID AREALM (cr=18006 pr=1056 pw=0 time=3140946 us
cost=3 size=86700 card=1156)
  15910     DOMAIN INDEX  AREALM_IDX (cr=3208 pr=330 pw=0 time=16413 us cost=0 size=0
card=0)

Rows     Execution Plan
-------  ---------------------------------------------------
      0  SELECT STATEMENT   MODE: ALL_ROWS
   5501   HASH (UNIQUE)
  15910    NESTED LOOPS
      2     TABLE ACCESS   MODE: ANALYZED (FULL) OF 'STATES' (TABLE)
  15910      TABLE ACCESS   MODE: ANALYZED (BY INDEX ROWID) OF 'AREALM'
             (TABLE)
  15910       DOMAIN INDEX OF 'AREALM_IDX' (INDEX (DOMAIN))
```

Postgres:

```
explain (analyze,buffers) select distinct a.fullname from tiger.arealm a,tiger.states b
where b.statefp10='06' and ST_Intersects(a.the_geom,b.the_geom);
```

QUERY PLAN

```
-----------------------------------------------------------------------------------------------------------------------------
 HashAggregate  (cost=24.12..38.95 rows=1483 width=18) (actual time=64477.008..64480.448 rows=5501 loops=1)
   Buffers: shared hit=432812 read=1398
   ->  Nested Loop  (cost=0.00..20.42 rows=1483 width=18) (actual time=52.056..64350.055 rows=15910 loops=1)
         Join Filter: _st_intersects(a.the_geom, b.the_geom)
         Buffers: shared hit=427131 read=1349
         ->  Seq Scan on states b  (cost=0.00..3.30 rows=2 width=1065526) (actual time=0.013..0.053 rows=2 loops=1)
               Filter: ((statefp10)::text = '06'::text)
               Buffers: shared hit=2
         ->  Index Scan using arealm_the_geom_gist on arealm a  (cost=0.00..8.30 rows=1 width=3941) (actual time=9.024..74.826 rows=8237 loops=2)
               Index Cond: (a.the_geom && b.the_geom)
               Buffers: shared hit=5842 read=870
 Total runtime: 64495.677 ms
```

Here **Postgres** takes only **65 secs** whereas **Oracle 11g** takes about **127 secs** to process the same query, **Hit Ratio** in **Postgres** is 432812/(432812+1398)=**0.99** and in **Oracle 11g** it is (1-766/(15705+4))=**0.95**

5. CONCLUSION & SUMMARY

From the experimental results that we saw, we can conclude that Postgres performs better than Oracle 11g both in the Cold Phase and Warm Phase. Though in few queries Oracle 11g performed better but on the whole Postgres overpowered Oracle 11g. In the warm phase in 3 out of 4 queries Postgres performed significantly well, from this we can conclude that Postgres has better automatic memory management capabilities and page replacement policies. May be Oracle 11g needs more tuning to perform better. And also in the Cold Phase, Postgres performs significantly well except in few cases such as in the Adjacency Operation (TOUCH). Since Postgres uses the underlying GEOS(Geometry Engine - Open Source) library functions for implementing the geometric operations whereas Oracle 11g implements them on its own, and since in majority Postgres performs well, we can conclude that GEOS geometric algorithms are more efficiently designed than Oracle 11g. And also Postgres planner is more efficiently designed to take advantage of any available indexes to use in queries for achieving better performance whereas in Oracle 11g we saw that we have to specify them explicitly through functions. On the whole it is the open-source that wins the game! The next section briefly summarizes the queries and gives the bar graphs for the two phases separately.

5.1 COLD PHASE QUERIES

5.1.1 Contains Properly:

a) Polygon/Point

Query 1: Retrieve all the Hospitals located inside the state District_of_Columbia

b) Polygon/Line

Query 2: List all the linear water bodies in the state District_of_Columbia

c) Polygon/Polygon

Arealm/States

Query 3: Retrieve all Shopping Center located in the state California

Areawater/States

Query 4: Retrieve all the Lakes/Ponds in the state District_of_Columbia

5.1.2 Adjacent Operation:

a) Polygon/Polygon

Query 5: Retrieve all the states which are adjacent to Nebraska

b) Polygon/Line

Query 6: Retrieve all polygons from arealm which are adjacent to Yukon River

c) Line/Line

Query 7: Retrieve all the tributaries of Yukon River

5.1.3 Intersect Operation:

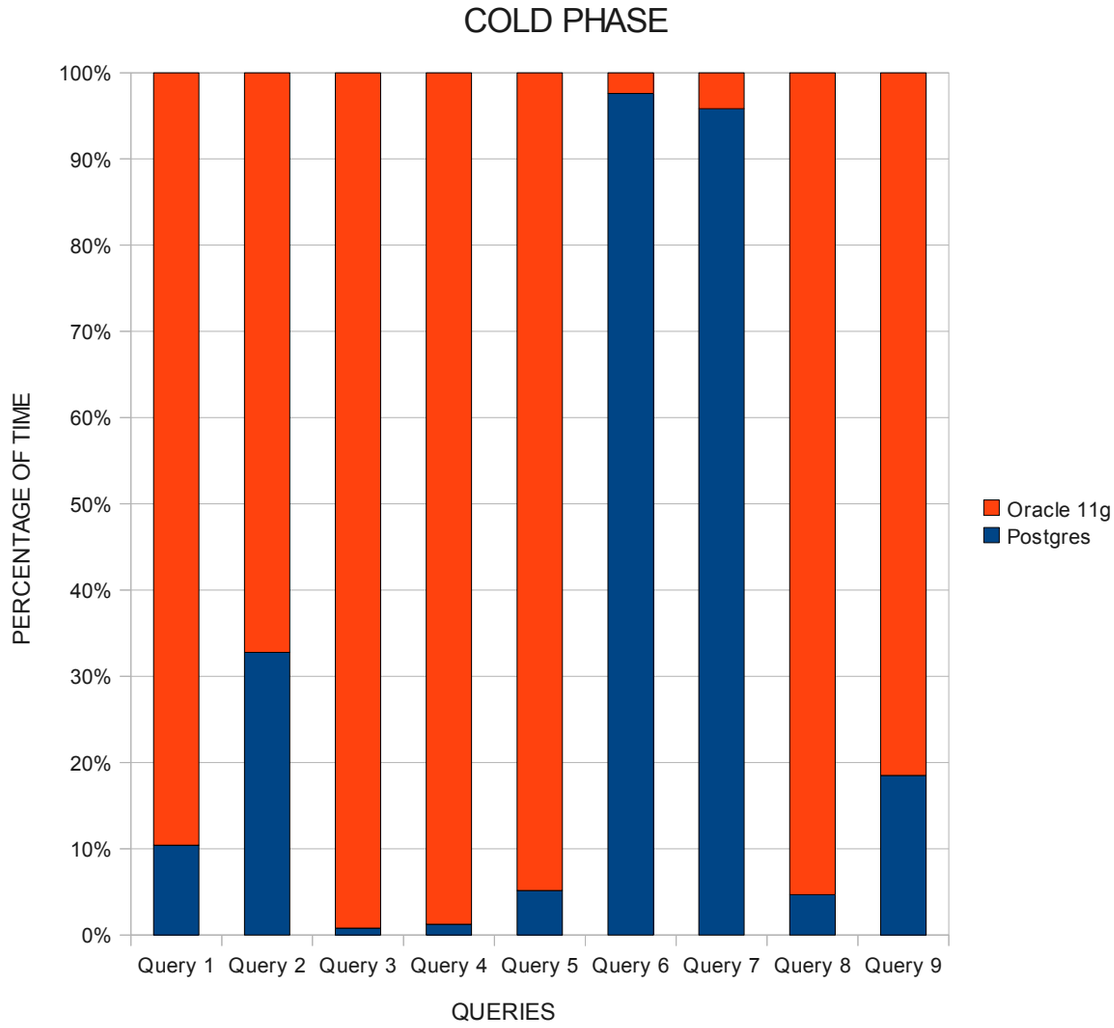
a) Polygon/Line Small

Query 8: Retrieve all the interstate connector roads in the state California

b) Polygon/Line Large

Query 9: Retrieve all roads which go through forest

Since the values range from 0.32 secs to 687 secs, so instead of bar graphs we have given Percent Bar Graph showing the percentage of time or the ratio of time taken by the two databases:



5.2 WARM PHASE QUERIES

a) Polygon/Polygon Large

Query 1: Retrieve all the areawater bodies which intersect with the state New York having stateid 36

b) Polygon/Line

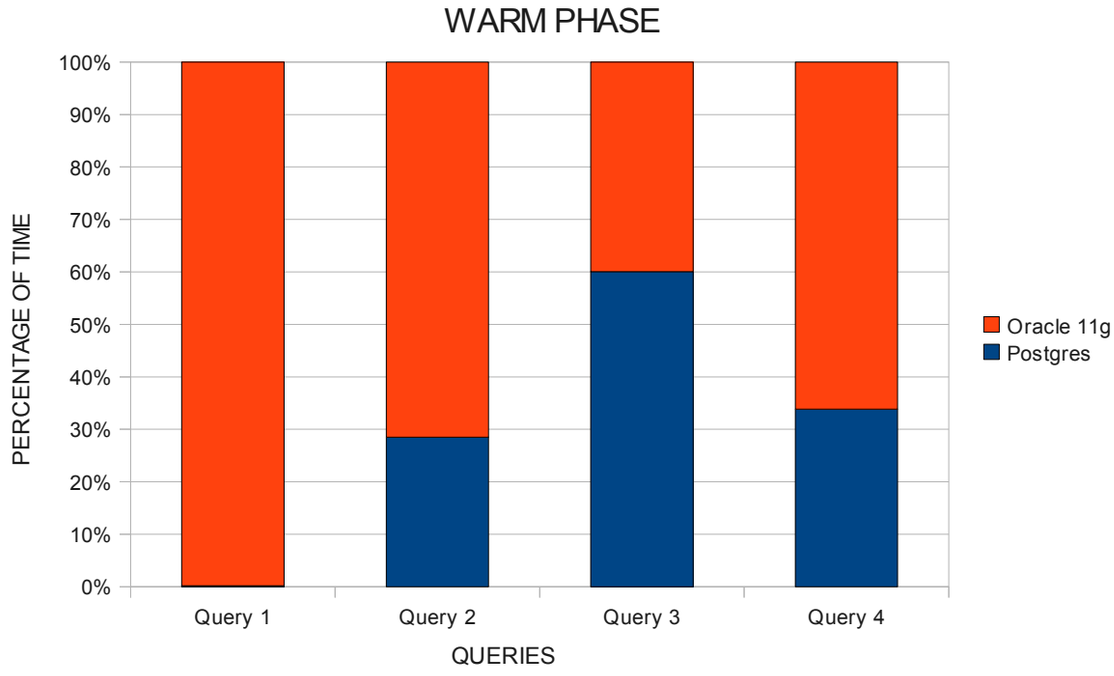
Query 2: Retrieve all the linearwater features which intersect with the state Washington having stateid 53

c) Polygon/Point

Query 3: Retrieve all the point features which intersect with the state Texas having stateid 48

a) Polygon/Polygon

Query 4: Retrieve all the area landmarks which intersect with the state California having stateid 06



REFERENCES

[1] Norman W. Paton, M. Howard Williams, Kosmas Dietrich, Olive Liew, Andrew Dinn, Alan Patrick, “VESPA: A benchmark for vector spatial databases,” Lecture Notes in Computer Science, Vol. 1832/2000, pp.81-101, 2001.

[2] Stonebraker M., Frew J., Gardels K., and Meredith J.1993. The SEQUOIA 2000 Storage Benchmark, Proc. ACM SIGMOD Int. Conf. on Management of Data, Washington, DC, 1993, pp. 2-11.

[3] PostGIS 1.5.2SVN Manual

[4] Oracle® Spatial Developer's Guide 11g Release 1 (11.1)

[5] Oracle® Database Concepts 11*g* Release 1 (11.1)

[6] Oracle® Database Performance Tuning Guide 11*g* Release 2 (11.2)

[7] 2010 TIGER/Line Shapefiles Technical Documentation.

[8] PostgreSQL 9.0.4 Documentation: Appendix F. Additional Supplied Modules: F.24. pg_buffercache

[9] PostgreSQL 9.0.4 Documentation: Chapter 18. Server Configuration

[10] PostgreSQL 9.0.4 Documentation: EXPLAIN

[11] PostgreSQL 9.0.4 Documentation: ANALYZE

[12] Chapter 4. Using PostGIS: Data Management and Queries

[13] EXPLAINING EXPLAIN EnterpriseDB

[14] Inside the PostgreSQL Shared Buffer Cache by Greg Smith

[15] http://pgfoundry.org/projects/pgfincore

[16] http://www.adp-gmbh.ch/ora/concepts/sga.html

[17] http://wiki.postgresql.org/wiki/VACUUM_FULL

[18] http://wiki.postgresql.org/wiki/Introduction_to_VACUUM,_ANALYZE,_EXPLAIN,_and_COUNT