UNIT 3 POSTGRESQL

Structure **Page Nos.**

3.0 Introduction 38
3.1 Objectives 38
3.2 Important Features 38
3.3 PostgreSQL Architectural Concepts 39
3.4 User Interfaces 41
3.5 SQL Variation and Extensions 43
3.6 Transaction Management 44
3.7 Storage and Indexing 46
3.8 Query Processing and Evaluation 47
3.9 Summary 49
3.10 Solutions/Answers 49

3.0 INTRODUCTION

PostgreSQL is an open-source object relational DBMS (ORDBMS). This DBMS was developed by the academic world, thus it has roots in Academia. It was first developed as a database called Postgres (developed at UC Berkley in the early 80s). It was officially called PostgreSQL around 1996 mostly, to reflect the added ANSI SQL compliant translator. It is one of the most feature-rich robust open-source database. In this unit we will discuss the features of this DBMS. Some of the topics that are covered in this unit include its architecture, user interface, SQL variation, transactions, indexes etc.

3.1 OBJECTIVES

After going through this unit, you should be able to:

- define the basic features of PostgreSQL;
- discuss the user interfaces in this RDBMS;
- use the SQL version of PostgreSQL, and
- identify the indexes and query processing used in PostgreSQL.

3.2 IMPORTANT FEATURES

PostgreSQL is an object relational database management system. It supports basic object oriented features including inheritance and complex data types along with special functions to deal with these data types. But, basically most of it is relational. In fact, most users of PostgreSQL do not take advantage of its extensive object oriented functionality. Following are the features of PostgreSQL that make it a very good DBMS:

- Full ANSI-SQL 92 compliance: It supports:
  - o most of ANSI 99 compliance as well,
  - o extensive support for the transactions,
  - o BEFORE and AFTER triggers, and
  - o implementation of stored procedures, constraints, referential integrity with cascade update/delete.

- Many high level languages and native interfaces can be used for creating user-defined database functions.
- You can use native SQL, PgSQL (postgres counterpart to Oracle PL/SQL or MS SQL Server's/Sybase TransactSQL), Java, C, C++, and Perl.
- Inheritance of table structures - this is probably one of the rarely used useful features.
- Built-in complex data types such as IP Address, Geometries (Points, lines, circles), arrays as a database field type and ability to define your own data types with properties, operators and functions on these user-defined data types.
- Ability to define Aggregate functions.
- Concept of collections and sequences.
- Support for multiple operating systems like Linux, Windows, Unix, Mac.
- It may be considered to be one of the important databases for implementing Web based applications. This is because of the fact that it is fast and feature rich. PostgreSQL is a reasonably fast database with proper support for web languages such as PHP, Perl. It also supports the ODBC and JDBC drivers making it easily usable in other languages such as ASP, ASP.Net and Java. It is often compared with MySQL - one of the fastest databases on the web (open source or non). Its querying speed is in line with MySQL. In terms of features though PostgreSQL is definitely a database to take a second look.

3.3 POSTGRESQL ARCHITECTURAL CONCEPTS

Before we dig deeper into the features of PostgreSQL, let's take a brief look at some of the basic concepts of Postgres system architecture. In this context, let us define how the parts of Postgres interact. This will make the understanding of the concepts simpler. *Figure 1* shows the basic architecture of the PostGreSQL on the Unix operating system.

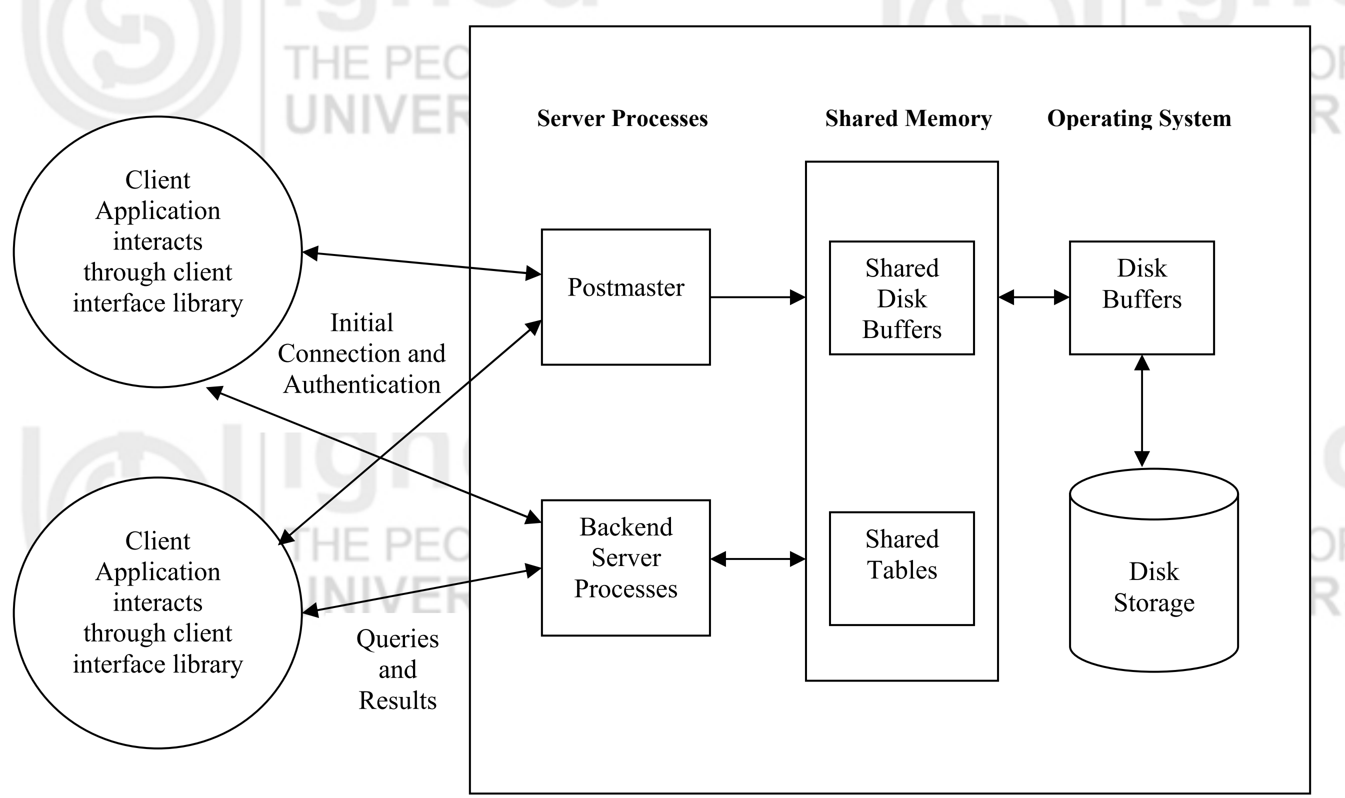

Figure 1: The Architecture of PostGresSQL

The Postgres uses a simple *process per-user* client/server model.

A session on PostgresSQL database consists of the following co-operating processes:

- A supervisory daemon process (also referred to as postmaster),
- The front-end user application process (e.g., the psql program), and
- One or more backend database server processes (the PostGres process itself).

The single postmaster process manages the collection of a database (also called an installation or site) on a single host machine. The client applications that want access to a database stored at a particular installation make calls to the client interface library. The library sends user requests over the network to the postmaster, in turn starts a new backend server process. The postmaster then connects the client process to the new server. Exam this point onwards, the client process and the backend server process communicate with each other without any intervention on the part of the postmaster. Thus, the postmaster process is always running – waiting for requests from the new clients. Please note, the client and server processes will be created and destroyed over a period of time as the need arises.

Can a client process, make multiple connections to a backend server process? The libpq library allows a single client to make multiple connections to backend server processes. However, please note, that these client processes are not multi-threaded processes. At present the multithreaded front-end/backend connections are not supported by libpq. This client server or font-end/back-end combination of processes on different machines files that can be accessed on a client machine that permits may not be accessible (or may only be accessed using a different filename) on the database server machine.

Please, also note that the postmaster and postgres servers run with the user-id of the Postgres *superuser* or the administrator. Please also note, that the Postgres superuser does not have to be a special user and also that the Postgres superuser should definitely not be the UNIX superuser (called root). All the files related to database belong to this Postgres superuser.

Figure 2 shows the establishing of connection in PostgresSQL.

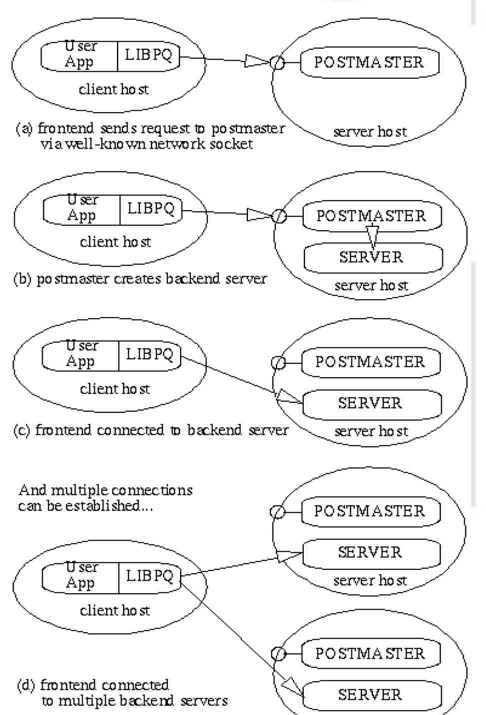

Figure 2: Establishing connection in PostgresSQL

3.4 USER INTERFACES

Having discussed the basic architecture of the PostgreSQL, the question that now arises is, how does one access databases? PostgreSQL have the following interfaces for the access of information:

- **Postgres terminal monitor programs (e.g. psql):** It is a SQL command level interface that allows you to enter, edit, and execute SQL commands interactively.
- **Programming Interface:** You can write a C program using the LIBPQ subroutine library. This allows you to submit SQL commands from the host language - C and get responses and status messages back to your program.

But how would you be referring to this interface? To do so, you need to install PostgreSQL on your machine. Let us briefly point out some facts about the installation of PostgreSQL.

Installing Postgres on Your machine: Since, Postgres is a client/server DBMS, therefore, as a user, you need the client's portion of the installation (an example of a client application interface is the interactive monitor psql). One of the common directory where PostGres may be installed on Unix machines is /usr/local/pgsql. Therefore, we will assume that Postgres has been installed in the directory/usr/local/pgsql. If you have installed PostGres in a different directory then you should substitute this directory name with the name of that directory. All PostGres commands are installed in the directory /usr/local/pgsql/bin. Therefore, you need to add this directory to your shell command path in Unix.

For example, on the Berkeley C shell or its variants such as csh or tcsh, you need to add:

```
% set path = ( /usr/local/pgsql/bin path )
```

in the .login file in the home directory.

On the Bourne shell or its variants such as sh, ksh, or bash, you need to add:

```
% PATH=/usr/local/pgsql/bin PATH
% export PATH
```

to the profile file in your home directory.

Other Interfaces

Some other user interfaces that are available for the PostGres are:

pgAdmin 3 from http://www.pgadmin.org for Windows/Linux/*BSD/*nix (Experimental Mac OS-X port). This interface was released under the Artistic License is a complete PostgreSQL administration interface. It is somewhat similar to Microsoft's Enterprise Manager and written in C++ and wxWindows. It allows administration of almost all database objects and ad-hoc queries.

PGAccess from http://www.pgaccess.org for most platforms is the original PostgreSQL GUI. It is a MS Access-style database browser that has been written in Tcl/Tk. It allows browsing, adding and editing tables, views, functions, sequences, databases, and users, as well as graphically-assisted queries. A form and report designer are also under development.

Many similar open source tools are available on the Internet.
Let us now describe the most commonly used interface for PostgreSQL i.e., psql in more details.

Starting the Interactive Monitor (psql)

You can process an application from a client if:

- the site administrator has properly started the postmaster process, and
- you are authorised to use the database with the proper user id and password.

As of Postgres v6.3, two different styles of connections are supported. These are:

- TCP/IP network connections or
- Restricted database access to local (same-machine) socket connections only.

These choices are significant in case, you encounter problems in connecting to a database. For example, in case, you get the following error message from a Postgres command (such as psql or createdb):

% psql template1
Connection to database 'postgres' failed.
connectDB() failed: Is the postmaster running and accepting connections at 'UNIX Socket' on port '5432'?

or

% psql -h localhost template1
Connection to database 'postgres' failed.
connectDB() failed: Is the postmaster running and accepting TCP/IP (with -i) connections at 'localhost' on port '5432'?

It is because of the fact that either the postmaster is not running, or you are attempting to connect to the wrong server host. Similarly, the following error message means that the site administrator has started the postmaster as the wrong user.

FATAL 1:Feb 17 23:19:55:process userid (2360) != database owner (268)

Accessing a Database

Once you have a valid account then the next thing is to start accessing the database. To access the database with PostGres mydb database you can use the command:

% psql mydb

You may get the following message:

Welcome to the POSTGRESQL interactive sql monitor:
Please read the file COPYRIGHT for copyright terms of POSTGRESQL

type \? for help on slash commands
type \q to quit
type \g or terminate with semicolon to execute query
You are currently connected to the database: template1

mydb=>

The prompt indicates that the terminal monitor is ready for your SQL queries. These queries need to be input into a workspace maintained by the terminal monitor. The psql program also responds to escape codes (you must have used them while programming in C) that begin with the backslash character, "\". For example, you can get help on the syntax of PostGres SQL commands by typing:

mydb=> \h

Once you have completed the query you can pass the contents of the workspace to the Postgres server by typing:

mydb=> \g

This indicates to the server that it may process the query. In case you terminate the query with a semicolon, the "\g" is not needed. psql automatically processes the queries that are terminated by a semicolon.

You can store your queries in a file. To read your queries from such a file you may type:

mydb=> \i filename

To exit psql and return to UNIX, type

mydb=> \q

White space (i.e., spaces, tabs and new line characters) may be used in SQL queries. You can also enter comments. Single-line comments are denoted by "--". Multiple-line comments, and comments within a line, are denoted by "/* ... */".

☞ **Check Your Progress 1**

1) What are the basic features of PostgreSQL?

……………………………………………………………………………………

……………………………………………………………………………………

……………………………………………………………………………………

2) What are the basic processes in PostgreSQL?

……………………………………………………………………………………

……………………………………………………………………………………

……………………………………………………………………………………

3) What are the different types of interfaces in PostgreSQL?

……………………………………………………………………………………

……………………………………………………………………………………

……………………………………………………………………………………

3.5 SQL VARIATION AND EXTENSIONS

The SQL was standardised by American National Standards Institute (ANSI) in 1986. The International Standards Organisation (ISO) standardised it in 1987. The United States Government's Federal Information Processing Standard (*FIPS*) adopted the ANSI/ISO standard in 1989, a revised standard known commonly as *SQL89* or *SQL1*, was published.

The SQL89 standard was intentionally left incomplete to accommodate commercial DBMS developer interests. However, the standard was strengthened by the ANSI committee, with the *SQL92* standard that was ratified in 1992 (also called *SQL2*). This standard addressed several weaknesses in SQL89 and set forth conceptual SQL features which at that time exceeded the capabilities of the RDBMSs of that time. In fact, the SQL92 standard was approximately six times the length of its predecessor. Since SQL 92 was large, therefore, the authors of the standards defined three levels of SQL92 compliance: *Entry-level conformance* (only the barest improvements to SQL89), *Intermediate-level conformance* (a generally achievable set of major advancements), and *Full conformance* (total compliance with the SQL92 features).

More recently, in 1999, the ANSI/ISO released the *SQL99* standard (also called *SQL3*). This standard addresses some of the more advanced and previously ignored areas of modern SQL systems, such as object-relational database concepts, call level interfaces, and integrity management. SQL99 replaces the SQL92 levels of compliance with its own degrees of conformance: *Core SQL99* and *Enhanced SQL99*. PostgreSQL presently conforms to most of the Entry-level SQL92 standard, as well as many of the Intermediate- and Full-level features. Additionally, many of the features new in SQL99 are quite similar to the object-relational concepts pioneered by PostgreSQL (arrays, functions, and inheritance).

PostgreSQL also provide several extenstions to standard SQL. Some of these extensions are:

- PostgreSQL supports many non-standard types. These include abstract data types like complex, domains, cstring, record, trigger, void etc. It also includes polymorphic types like any array.

- It supports triggers. It also allows creation of functions which can be stored and executed on the server.

- It supports many procedural programming languages like PL/pqSQL, PL/Tcl, PL/Python etc.

3.6 TRANSACTION MANAGEMENT

Every SQL query is executed as a transaction. This results in some desirable properties, while processing such queries are all-or-nothing types when they are making modifications. This ensures the integrity and recoverability of queries. For example, consider the query:

```
UPDATE students SET marks = marks +10;
```

Assume that the query above has modified first 200 records and is in the process of modifying 201st record out of the 2000 records. Suppose a user terminates the query at this moment by resetting the computer, then, on the restart of the database, the recovery mechanism will make sure that none of the records of the student is modified. The query is required to be run again to make the desired update of marks. Thus, the PostGres has made sure that the query causes no recovery or integrity related problems.

This is a very useful feature of this DBMS. Suppose you were executing a query to increase the salary of employees of your organisation by Rs.500 and there is a power failure during the update procedure. Without transactions support, the query may have updated records of some of the persons, but not all. It would be difficult to know where the UPDATE failed. You would like to know: “Which records were updated, and which ones were not?” You cannot simply re-execute the query, because some people who may have already received their Rs. 500 increment would also get another increase by Rs. 500/-. With the transactions mechanism in place, you need not bother about it for when the DBMS starts again, first it will recover from the failure thus, undoing any update to the data. Thus, you can simply re-execute the query.

Multistatement Transactions

By default in PostGres each SQL query runs in its own transaction. For example, consider two identical queries:

```
mydb=> INSERT INTO table1 VALUES (1);
INSERT 1000 1
OR
mydb=> BEGIN WORK;
BEGIN
mydb=> INSERT INTO table1 VALUES (1);
INSERT 1000 1
mydb=> COMMIT WORK;
COMMIT
```

The former is a typical INSERT query. Before PostgreSQL starts the INSERT it automatically begins a transaction. It performs the INSERT, and then commits the transaction. This step occurs automatically for any query with no explicit transaction. However, in the second version, the INSERT uses explicit transaction statements. BEGIN WORK starts the transaction, and COMMIT WORK commits the transaction. Both the queries results in same database state, the only difference being the implied BEGIN WORK...COMMIT WORK statements. However, the real utility of these transactions related statements can be seen in the ability to club multiple queries into a single transaction. In such a case, either all the queries will execute to completion or none at all. For example, in the following transaction either both INSERTs will succeed or neither.

```
mydb=> BEGIN WORK;
BEGIN
mydb=> INSERT INTO table1 VALUES (1);
INSERT 1000 1
mydb=> INSERT INTO table1 VALUES (2);
INSERT 2000 1
mydb=> COMMIT WORK;
COMMIT
```

The PostgreSQL has implemented both two-phase locking and multi-version concurrency control protocols. The multi-version concurrency control supports all the isolation levels of SQL standards. These levels are:

- Read uncommitted
- Read committed
- Repeatable Read, and
- Serialisable.

☞ Check Your Progress 2

1) List the add-on non-standard types in PostgreSQL?

……………………………………………………………………………………

……………………………………………………………………………………

……………………………………………………………………………………

2) How are the transactions supported in PostgreSQL?

……………………………………………………………………………………

……………………………………………………………………………………

……………………………………………………………………………………

3) State True or False

a) PostgreSQL is fully compliant with SQL 99 standard. ☐

b) PostgreSQL supports server-based triggers. ☐

c) PostgreSQL supports all levels of isolation as defined by SQL standard ☐

d) COMMIT is not a keyword in PostgrSQL. ☐

3.7 STORAGE AND INDEXING

After discussing the basics of transactions let us discuss some of the important concepts used in PostgreSQL from the point of view of storage and indexing of tables. An interesting point here is that PostgreSQL defines a number of *system columns* in all tables. These system columns are normally invisible to the user, however, explicit queries can report these entries. These columns, in general, contains *meta-data* i.e., data about data contained in the records of a table.

Thus, any record would have attribute values for the system-defined columns as well as the user-defined columns of a table. The following table lists the system columns.

Column Name	Description
oid (object identifier)	The unique object identifier of a record. It is automatically added to all records. It is a 4-byte number. It is never re-used within the same table.
tableoid (table object identifier)	The oid of the table that contains a row. The pg_class system table relates the name and oid of a table.
xmin (transaction minimum)	The transaction identifier of the inserting transaction of a tuple.
Cmin (command minimum)	The command identifier, starting at 0, is associated with the inserting transaction of a tuple.
xmax (transaction maximum)	The transaction identifier of a tuple's deleting transaction. If a tuple has not been deleted then this is set to zero.
cmax (command maximum)	The command identifier is associated with the deleting transaction of a tuple. Like xmax, if a tuple has not been deleted then this is set to zero.
ctid (tuple identifier)	This identifier describes the physical location of the tuple within the database. A pair of numbers are represented by the ctid: the block number, and tuple index within that block.

Figure 3: System Columns

If the database creator does not create a primary key explicitly, it would become difficult to distinguish between two records with identical column values. To avoid such a situation PostgreSQL appends every record with its own *object identifier* number, or *OID*, which is unique to that table. Thus, no two records in the same table will ever have the same OID, which, also mean that no two records are identical in a table. The oid makes sure of this.

Internally, PostgreSQL stores data in operating system files. Each table has its own file, and data records are stored in a sequence in the file. You can create an index on the database. An index is stored as a separate file that is sorted on one or more columns as desired by the user. Let us discuss indexes in more details.

Indexes

Indexes allow fast retrieval of specific rows from a table. For a large table using an index, finding a specific row takes fractions of a second while non-indexed entries will require more time to process the same information. PostgreSQL does not create indexes automatically. Indexes are user defined for attributes or columns that are frequently used for retrieving information.

For example, you can create an index such as:

```
mydb=> CREATE INDEX stu_name ON Student (first_name);
```

Although you can create many indexes they should justify the benefits they provide for retrieval of data from the database. Please note that an index adds on overhead in terms of disk space, and performance as a record update may also require an index update. You can also create an index on multiple columns. Such multi-column indexes are sorted by the first indexed column and then the second indexed column.

PostgreSQL supports many types of index implementations. These are:

- **B-Tree Indexes:** These are the default type index. These are useful for comparison and range queries.
- **Hash Indexes:** This index uses linear hashing. Such indexes are not preferred in comparison with B-tree indexes.
- **R-Tree indexes:** Such index are created on built-in spatial data types such as box, circle for determining operations like overlap etc.
- **GiST Indexes:** These indexes are created using Generalised search trees. Such indexes are useful for full text indexing, and are thus useful for retrieving information.

3.8 QUERY PROCESSING AND EVALUATION

This section provides a brief introduction to the query processing operations of PostgreSQL. It will define the basic steps involved in query processing and evaluation in the PostgreSQL. The query once submitted to PostgreSQL undergoes the following steps, (in sequential order) for solving the query:

- A connection from the client application program to the PostgreSQL server is established. The application program transmits the query to the server and waits for the server to process the query and return the results.

- The *parser* at the server checks the syntax of the query received from the client and creates a *query tree.*

- The *rewrite system* takes the query tree created by the parser as the input, and selects the *rules* stored in the *system catalogues* that may apply to the query tree. It then performs the transformation given as per the *rules*. It also rewrites any query made against a view to a query accessing the base tables.

- The *planner/optimiser* takes the (rewritten) query tree and creates a *query plan* that forms the input to the *executor*. It creates a structured list of all the possible *paths* leading to the same result. Finally, the cost for the execution of each path is estimated and the cheapest path is chosen. This cheapest path is expanded into a complete query evaluation plan that the executor can use.

- The executor recursively steps through the query evaluation *plan tree* supplied by the planner and creates the desired output.

A detailed description of the process is given below:

A query is sent to the backend (it may be the query processor) via data packets that may result from database access request through TCP/IP on a remote database access or local Unix Domain sockets. The query is then loaded into a string, and passed to the parser, where the lexical scanner, **scan.l**, tokenises the words of the query string. The parser then uses another component **gram.y** and the tokens to identify the query type, such as, Create or Select queries. Now the proper query-specific structure is loaded.

The statement is then identified as complex (*SELECT / INSERT / UPDATE / DELETE*) or as simple, e.g., *CREATE USER, ANALYSE* etc. Simple utility commands are processed by statement-specific functions, however, for handling complex statements it need further detailing and processing.

Complex queries could be, *SELECT,* return columns of data or specify columns that need to be modified like *INSERT* and *UPDATE.* The references of these columns are converted to TargetEntry entries that may, be linked together to create the *target list* of the query. The target list is stored in **Query.targetList.**

Now, the Query is modified for the desired *VIEWS* or implementation of *RULES* that may apply to the query.

The optimiser then, creates the query execution plan based on the Query structure and the operations to be performed in order to execute the query. The Plan is passed to the executor for execution, and the results are returned to the client.

Query Optimisation

The task of the *planner/optimizer* is to create an optimal execution plan out of the available alternatives. The query tree of a given SQL query can be actually executed in a wide variety of different ways, each of which essentially producing the same set of results. It is not possible for the query optimiser to examine each of these possible execution plans to choose the execution plan that is expected to run the fastest. Thus, the optimiser must find a reasonable (non optimal) query plan in the predefined time and space complexity. Here a Genetic query optimiser is used by the PostgreSQL.

After the cheapest path is determined, a full-fledged *plan tree* is built in order to pass it to the executor. This represents the desired execution plan in sufficient detail for the executor to run it.

☞ Check Your Progress 3

1) What are system columns?

……………………………………………………………………………………

……………………………………………………………………………………

……………………………………………………………………………………

2) What are the different types of indexes in PostgreSQL?

……………………………………………………………………………………

……………………………………………………………………………………

……………………………………………………………………………………

3) What are the different steps for query evaluation?

…………………………………………………………………………………………

…………………………………………………………………………………………

…………………………………………………………………………………………

3.9 SUMMARY

This unit provided an introduction to some of the basic features of the PostgreSQL DBMS. This is very suitable example of a DBMS available as Open Source software. This database provides most of the basic features of the relational database management systems. It also supports some of the features of object oriented programming like inheritance, complex data types, declarations of functions and polymorphism. Thus, it is in the category of object relational DBMS. PostgreSQL has features such as, the client server architecture with the Postmaster, server and client as the main processes. It supports all the basic features of SQL 92 and many more features recommended by SQL 99. PostgreSQL treats even single SQL queries as transaction implicitly. This helps in maintaining the integrity of the database at all times. PostgreSQL places many system related attributes in the tables defined by the users. Such attributes help the database with tasks that may be useful for indexing and linking of records. It also supports different types of indexes, which includes B-Tree, Hash, R-Tree and GiST types of indexes. Thus, making it suitable for many different types of application like spatial, multi-dimensional database applications. It has a standard process for query evaluation and optimisation.

3.10 SOLUTIONS/ANSWERS

Check Your Progress 1

1) PostgreSQL supports the following features:

- ANSI SQL 2 compliance,
- Support for transactions, triggers, referential integrity and constraints,
- High level language support,
- Inheritance, complex data types and polymorphism,
- Built in complex data types like IP address,
- Aggregate functions, collections and sequences,
- Portability, and
- ODBC and JDBC drivers.

2) PostgreSQL has the following three processes:

- Postmaster
- Server process
- Client processes

3) PostgreSQL supports both the terminal monitor interfaces and program driven interfaces.

Check Your Progress 2

1) Some of these types are: complex, domains, cstring, record, trigger, void etc.

2) Each SQL statement is treated as a transaction. It also has provision for multi-statement transactions.

3) (a) False (b) True (c) True (d) False

Check Your Progress 3

1) System columns are added by PostgrSQL to all the tables. These are oid (object identifier), tableoid, xmin, xmax, cmax, ctid.

2) PostgreSQL supports the following four types of indexes: B-Tree, Hash, R-Tree and GiST.

3) The steps are:

   - Query submission, the
   - Parsing by the parser to query tree,
   - Transformation of query by rewrite system
   - Creation of query evaluation plan by the optimiser, and
   - Execution by executor.