Aishwarya College of Education

Course :-BCA
Class :- BCA 2nd Year
Subject :- DBMS
Unit :- I-II-III-IV-V

Contents

Unit-I

DATA:

Data is a collection of facts, figures and statistics related to an object. For example: Students fill an admission form when they get admission in college. The form consists of raw facts about the students. These raw facts are student's name, father name, address etc. The purpose of collecting this data is to maintain the records of the students during their study period in the college.

INFORMATION:

Processed data is called information. OR The manipulated and processed form of data is called information. For example: Data collected from census is used to generate different type of information. The government can use it to determine the literacy rate in the country. Government can use the information in important decision to improve literacy rate.

ADVANTAGES OF THE DBMS:

1. Improved data sharing: The DBMS helps create an environment in which end users have better access to more and better-managed data. Such access makes it possible for end users to respond quickly to changes in their environment.

2. Improved data security: The more users access the data, the greater the risks of data security breaches. Corporations invest considerable amounts of time, effort, and money to ensure that corporate data are used properly. A DBMS provides a framework for better enforcement of data privacy and security policies.

3. Data integration: Wider access to well-managed data promotes an integrated view of the organization’s operations and a clearer view of the big picture. It becomes much easier to see how actions in one segment of the company affect other segments.

4. Minimized data inconsistency: Data inconsistency exists when different versions of the same data appear in different places. For example, data inconsistency exists when a company's sales department stores a sales representative's name as "Bill Brown" and the company's personnel department stores that same person's name as "William G. Brown," or when the company's regional sales office shows the price of a product as $45.95 and its national sales office shows the same product's price as $43.95. The probability of data inconsistency is greatly reduced in a properly designed database.

5. Improved data access: The DBMS makes it possible to produce quick answers to ad hoc queries. From a database perspective, a query is a specific request issued to the DBMS for data manipulation—for example, to read or update the data. Simply put, a query is a question, and an ad hoc query is a spur-of-the-moment question. The DBMS sends back an answer (called the query result set) to the application. For example, end users, when dealing with large amounts of sales data, might want quick answers to questions (ad hoc queries) such as: - What was the dollar volume of sales by product during the past six months? - What is the sales bonus figure for each of our salespeople during the past three months? - How many of our customers have credit balances of $3,000 or more?

6. Improved decision making: Better-managed data and improved data access make it possible to generate better-quality information, on which better decisions are based. The quality of the information generated depends on the quality of the underlying data. Data quality is a comprehensive approach to promoting the accuracy, validity, and timeliness of the data. While the DBMS does not guarantee data quality, it provides a framework to facilitate data quality

7. Increased end-user productivity: The availability of data, combined with the tools that transform data into usable information, empowers end users to make quick, informed decisions that can make the difference between success and failure in the global economy.

8. Controlling Data Redundancy: In non-database systems (traditional computer file processing), each application program has its own files. In this case, the duplicated copies of the same data are created at many places. In DBMS, all the data of an organization is integrated into a single database. The data is recorded at only one place in the database and it is not duplicated. For example, the dean's faculty file and the faculty payroll file contain several items that are identical. When they are converted into database, the data is integrated into a single database so that multiple copies of the same data are reduced to-single copy. In DBMS, the data redundancy can be controlled or reduced but is not removed completely. Sometimes, it is necessary to create duplicate copies of the same data items in order to relate tables with each other. By controlling the data redundancy, you can save storage space. Similarly, it is useful for retrieving data from database using queries.

9. Backup and Recovery Procedures: In a computer file-based system, the user creates the backup of data regularly to protect the valuable data from damaging due to failures to the computer system or application program. It is a time consuming method, if volume of data is large. Most of the DBMSs provide the 'backup and recovery' sub-systems that automatically create the backup of data and restore data if required. For example, if the computer system fails in the middle (or end) of an update operation

of the program, the recovery sub-system is responsible for making sure that the database is restored to the state it was in before the program started executing.

DISADVANTAGES OF DATABASE

1. Increased costs: Database systems require sophisticated hardware and software and highly skilled personnel. The cost of maintaining the hardware, software, and personnel required to operate and manage a database system can be substantial. Training, licensing, and regulation compliance costs are often overlooked when database systems are implemented.

2. Management complexity: Database systems interface with many different technologies and have a significant impact on a company's resources and culture. The changes introduced by the adoption of a database system must be properly managed to ensure that they help advance the company's objectives. Given the fact that database systems hold crucial company data that are accessed from multiple sources, security issues must be assessed constantly.

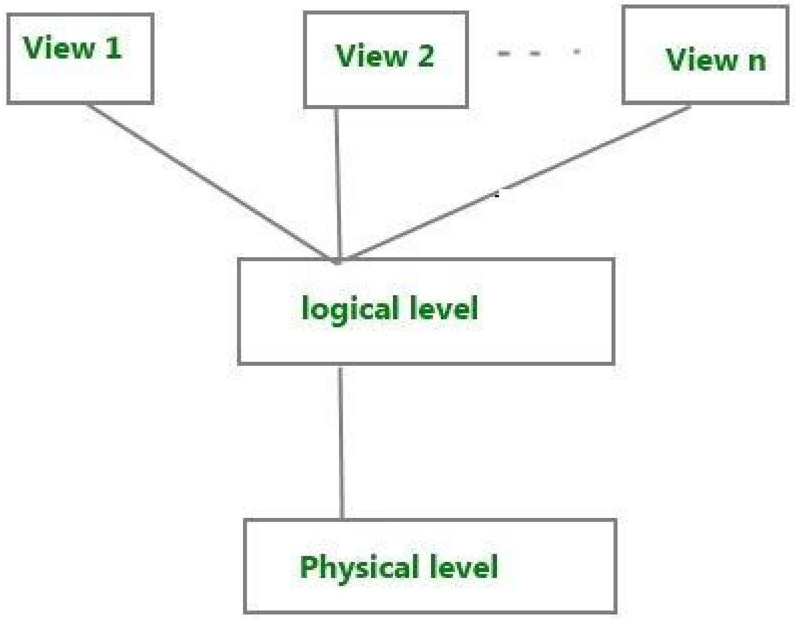
3. Maintaining currency: To maximize the efficiency of the database system, you must keep your system current. Therefore, you must perform frequent updates and apply the latest patches and security measures to all components. Because database technology advances rapidly, personnel training costs tend to be significant. Vendor dependence. Given the heavy investment in technology and personnel training, companies might be reluctant to change database vendors. As a consequence, vendors are less likely to offer pricing point advantages to existing customers, and those customers might be limited in their choice of database system components.

4. Frequent upgrade/replacement cycles: DBMS vendors frequently upgrade their products by adding new functionality. Such new features often come bundled in new upgrade versions of the software. Some of these versions require hardware upgrades. Not only do the upgrades themselves cost money, but it also costs money to train database users and administrators to properly use and manage the new features.

5. Appointing Technical Staff: The trained technical persons such as database administrator and application programmers etc are required to handle the DBMS. You have to pay handsome salaries to these persons. Therefore, the system cost increases.

Data abstraction

Database systems are made-up of complex data structures. To ease the user interaction with database, the developers hide internal irrelevant details from users. This process of hiding irrelevant details from user is called data abstraction.



Three Levels of data abstraction

We have three levels of abstraction:

Physical level: This is the lowest level of data abstraction. It describes how data is actually stored in database. You can get the complex data structure details at this level.

Logical level: This is the middle level of 3-level data abstraction architecture. It describes what data is stored in database.

View level: Highest level of data abstraction. This level describes the user interaction with database system.

Example: Let's say we are storing customer information in a customer table. At physical level these records can be described as blocks of storage (bytes, gigabytes, terabytes etc.) in memory. These details are often hidden from the programmers.

At the logical level these records can be described as fields and attributes along with their data types, their relationship among each other can be logically implemented. The programmers generally work at this level because they are aware of such things about database systems.

At view level, user just interact with system with the help of GUI and enter the details at the screen, they are not aware of how the data is stored and what data is stored; such details are hidden from them.

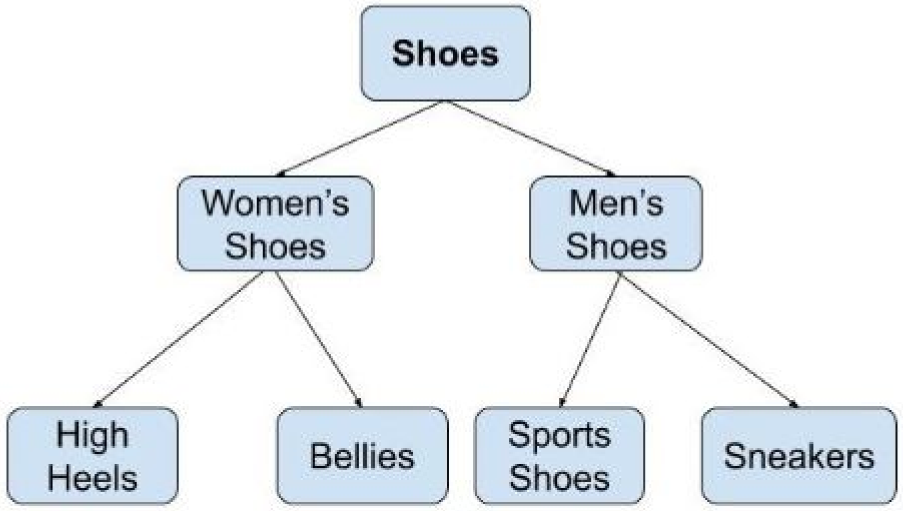
Data model

Data Model gives us an idea that how the final system will look like after its complete implementation. It defines the data elements and the relationships between the data elements. Data Models are used to show how data is stored, connected, accessed and updated in the database management system. Here, we use a set of symbols and text to represent the information so that members of the organisation can communicate and understand it. Though there are many data models being used nowadays but the Relational model is the most widely used model. Apart from the Relational model, there are many other types of data models about which we will study in details in this blog. Some of the Data Models in DBMS are:

1. Hierarchical Model
2. Network Model
3. Entity-Relationship Model
4. Relational Model

Hierarchical Model

Hierarchical Model was the first DBMS model. This model organizes the data in the hierarchical tree structure. The hierarchy starts from the root which has root data and then it expands in the form of a tree adding child node to the parent node. This model easily represents some of the real-world relationships like food recipes, sitemap of a website etc. Example: We can represent the relationship between the shoes present on a shopping website in the following way:



Hierarchical Model

Features of a Hierarchical Model

1. One-to-many relationship: The data here is organized in a tree-like structure where the one-to-many relationship is between the data types. Also, there can be only one path from parent to any node. Example: In the above example, if we want to go to the node sneakers we only have one path to reach there i.e. through men's shoes node.
2. Parent-Child Relationship: Each child node has a parent node but a parent node can have more than one child node. Multiple parents are not allowed.
3. Deletion Problem: If a parent node is deleted then the child node is automatically deleted.
4. Pointers: Pointers are used to link the parent node with the child node and are used to navigate between the stored data. Example: In the above example the 'shoes' node points to the two other nodes 'women shoes' node and 'men's shoes' node.

Advantages of Hierarchical Model

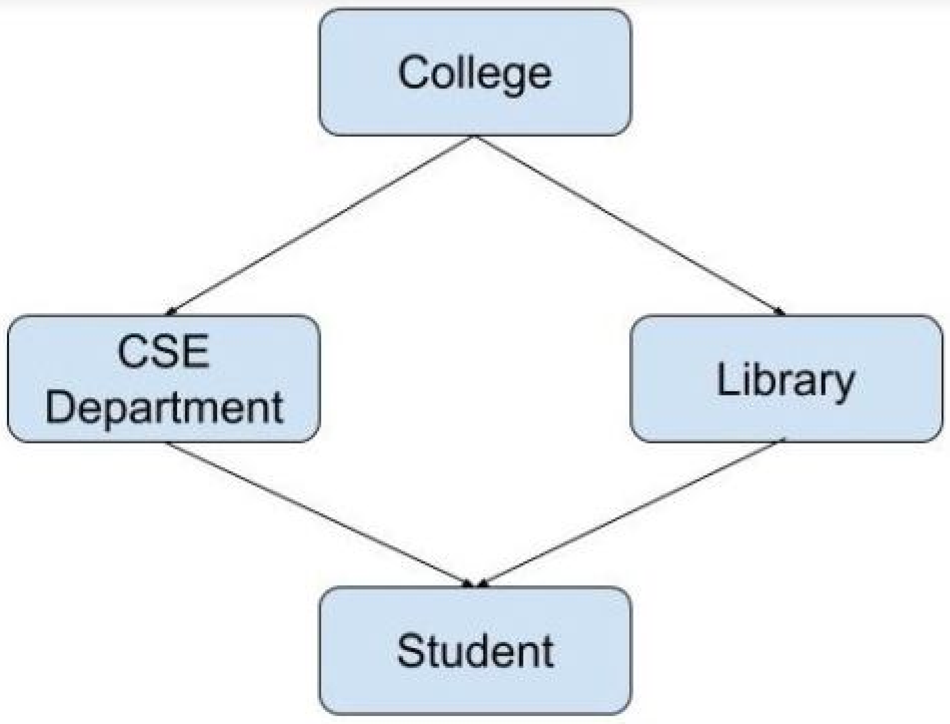
- It is very simple and fast to traverse through a tree-like structure.
- Any change in the parent node is automatically reflected in the child node so, the integrity of data is maintained.

Disadvantages of Hierarchical Model

- Complex relationships are not supported.
- As it does not support more than one parent of the child node so if we have some complex relationship where a child node needs to have two parent node then that can't be represented using this model.
- If a parent node is deleted then the child node is automatically deleted.

Network Model

This model is an extension of the hierarchical model. It was the most popular model before the relational model. This model is the same as the hierarchical model, the only difference is that a record can have more than one parent. It replaces the hierarchical tree with a graph. Example: In the example below we can see that node student has two parents i.e. CSE Department and Library. This was earlier not possible in the hierarchical model.



Network Model

Features of a Network Model

1. Ability to Merge more Relationships: In this model, as there are more relationships so data is more related. This model has the ability to manage one-to-one relationships as well as many-to-many relationships.
2. Many paths: As there are more relationships so there can be more than one path to the same record. This makes data access fast and simple.
3. Circular Linked List: The operations on the network model are done with the help of the circular linked list. The current position is maintained with the help of a program and this position navigates through the records according to the relationship.

Advantages of Network Model

- The data can be accessed faster as compared to the hierarchical model. This is because the data is more related in the network model and there can be more than one path to reach a particular node. So the data can be accessed in many ways.
- As there is a parent-child relationship so data integrity is present. Any change in parent record is reflected in the child record.

Disadvantages of Network Model

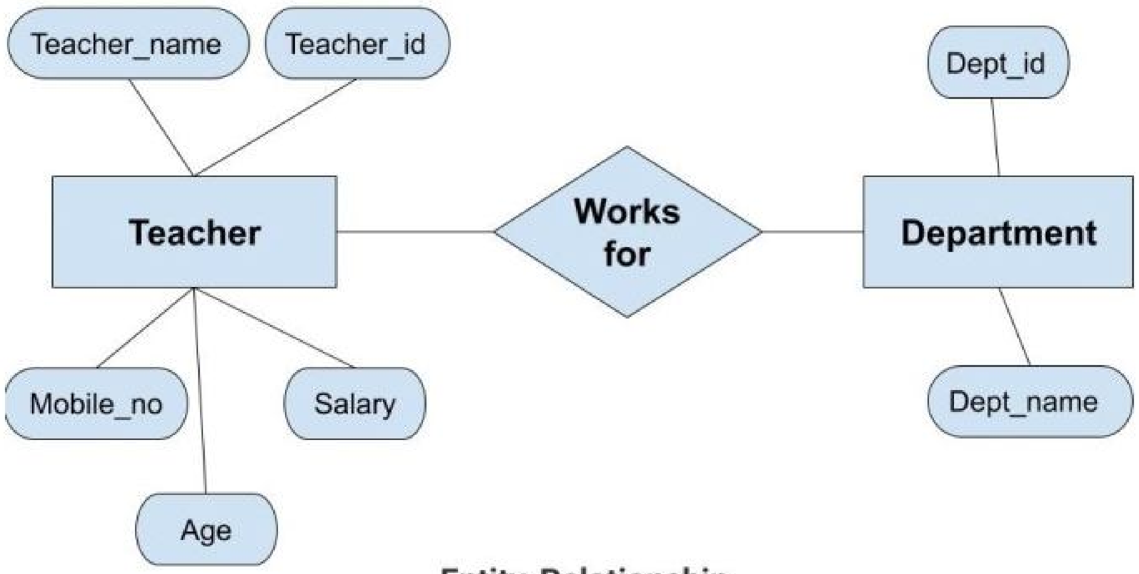
- As more and more relationships need to be handled the system might get complex. So, a user must be having detailed knowledge of the model to work with the model.
- Any change like updation, deletion, insertion is very complex.

Entity-Relationship Model

Entity-Relationship Model or simply ER Model is a high-level data model diagram. In this model, we represent the real-world problem in the pictorial form to make it easy for the stakeholders to understand. It is also very easy for the developers to understand the system by just looking at the ER diagram. We use the ER diagram as a visual tool to represent an ER Model. ER diagram has the following three components:

- ***Entities:*** Entity is a real-world thing. It can be a person, place, or even a concept. *Example:* Teachers, Students, Course, Building, Department, etc are some of the entities of a School Management System.
- ***Attributes:*** An entity contains a real-world property called attribute. This is the characteristics of that attribute. *Example:* The entity teacher has the property like teacher id, salary, age, etc.
- ***Relationship:*** Relationship tells how two attributes are related. *Example:* Teacher works for a department.

Example:



Entity-Relationship Model

In the above diagram, the entities are Teacher and Department. The attributes of ***Teacher*** entity are Teacher_Name, Teacher_id, Age, Salary, Mobile_Number. The attributes of entity ***Department*** entity are Dept_id, Dept_name. The two entities are connected using the relationship. Here, each teacher works for a department**.**

Features of ER Model

- ***Graphical Representation for Better Understanding:*** It is very easy and simple to understand so it can be used by the developers to communicate with the stakeholders.
- ***ER Diagram:*** ER diagram is used as a visual tool for representing the model.
- ***Database Design:*** This model helps the database designers to build the database and is widely used in database design**.**

Advantages of ER Model

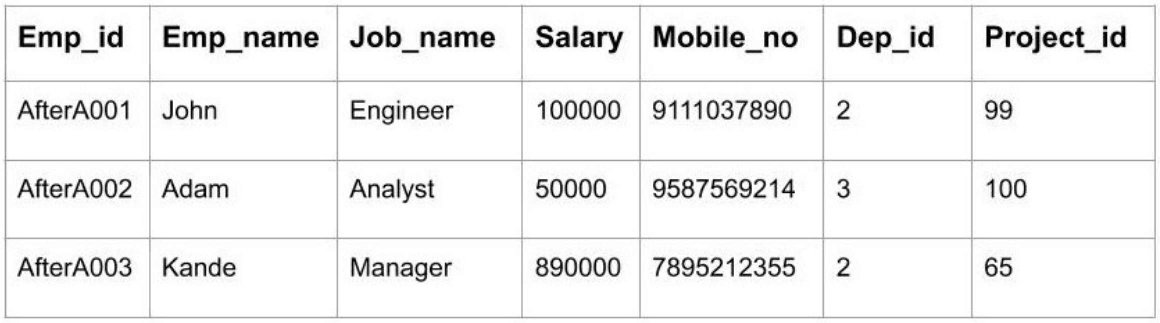
- ***Simple:*** Conceptually ER Model is very easy to build. If we know the relationship between the attributes and the entities we can easily build the ER Diagram for the model.
- ***Effective Communication Tool***: This model is used widely by the database designers for communicating their ideas.
- ***Easy Conversion to any Model***: This model maps well to the relational model and can be easily converted relational model by converting the ER model to the table. This model can also be converted to any other model like network model, hierarchical model etc.

Disadvantages of ER Model

- ***No industry standard for notation:*** There is no industry standard for developing an ER model. So one developer might use notations which are not understood by other developers.
- ***Hidden information:*** Some information might be lost or hidden in the ER model. As it is a high-level view so there are chances that some details of information might be hidden.

Relational Model

Relational Model is the most widely used model. In this model, the data is maintained in the form of a two-dimensional table. All the information is stored in the form of row and columns. The basic structure of a relational model is tables. So, the tables are also called *relations* in the relational model. ***Example:*** In this example, we have an Employee table.



Emp_id	Emp_name	Job_name	Salary	Mobile_no	Dep_id	Project_id
AfterA001	John	Engineer	100000	9111037890	2	99
AfterA002	Adam	Analyst	50000	9587569214	3	100
AfterA003	Kande	Manager	890000	7895212355	2	65

EMPLOYEE TABLE

Features of Relational Model

- ***Tuples***: Each row in the table is called tuple. A row contains all the information about any instance of the object. In the above example, each row has all the information about any specific individual like the first row has information about John.
- ***Attribute or field:*** Attributes are the property which defines the table or relation. The values of the attribute should be from the same domain. In the above example, we have different attributes of the *employee* like Salary, Mobile_no, etc.

Advantages of Relational Model

- ***Simple:*** This model is more simple as compared to the network and hierarchical model.
- ***Scalable:*** This model can be easily scaled as we can add as many rows and columns we want.
- ***Structural Independence:*** We can make changes in database structure without changing the way to access the data. When we can make changes to the database structure without affecting the capability to DBMS to access the data we can say that structural independence has been achieved.

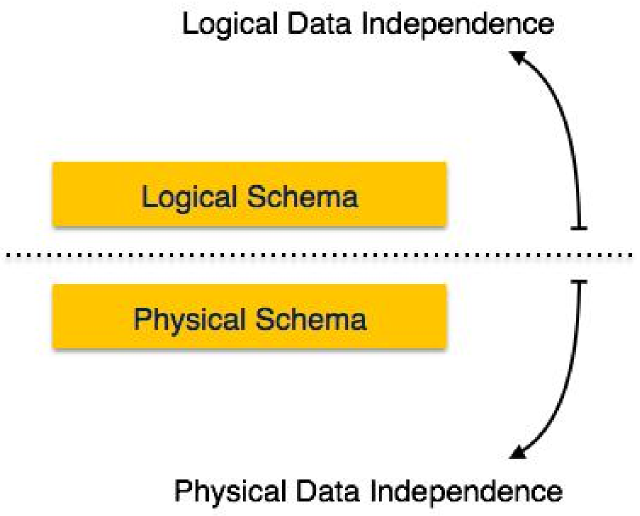
Disadvantages of Relational Model

- ***Hardware Overheads:*** For hiding the complexities and making things easier for the user this model requires more powerful hardware computers and data storage devices.
- ***Bad Design:*** As the relational model is very easy to design and use. So the users don't need to know how the data is stored in order to access it. This ease of design can lead to the development of a poor database which would slow down if the database grows.

But all these disadvantages are minor as compared to the advantages of the relational model. These problems can be avoided with the help of proper implementation and organization.

Data Independence

A database system normally contains a lot of data in addition to users' data. For example, it stores data about data, known as metadata, to locate and retrieve data easily. It is rather difficult to modify or update a set of metadata once it is stored in the database. But as a DBMS expands, it needs to change over time to satisfy the requirements of the users. If the entire data is dependent, it would become a tedious and highly complex job.



Metadata itself follows a layered architecture, so that when we change data at one layer, it does not affect the data at another level. This data is independent but mapped to each other.

Logical Data Independence

Logical data is data about database, that is, it stores information about how data is managed inside. For example, a table (relation) stored in the database and all its constraints, applied on that relation.

Logical data independence is a kind of mechanism, which liberalizes itself from actual data stored on the disk. If we do some changes on table format, it should not change the data residing on the disk.

Physical Data Independence

All the schemas are logical, and the actual data is stored in bit format on the disk. Physical data independence is the power to change the physical data without impacting the schema or logical data.

For example, in case we want to change or upgrade the storage system itself − suppose we want to replace hard-disks with SSD − it should not have any impact on the logical data or schemas

Data definition language

DDL(Data Definition Language) : DDL or Data Definition Language actually consists of the SQL commands that can be used to define the database schema. It simply deals with descriptions of the database schema and is used to create and modify the structure of database objects in the database.

Examples of DDL commands:

- CREATE – is used to create the database or its objects (like table, index, function, views, store procedure and triggers).
- DROP – is used to delete objects from the database.
- ALTER-is used to alter the structure of the database.
- TRUNCATE–is used to remove all records from a table, including all spaces allocated for the records are removed.
- COMMENT –is used to add comments to the data dictionary.
- RENAME –is used to rename an object existing in the database

Data manipulation language

DML(Data Manipulation Language) : The SQL commands that deals with the manipulation of data present in the database belong to DML or Data Manipulation Language and this includes most of the SQL statements.

Examples of DML:

- INSERT – is used to insert data into a table.
- UPDATE – is used to update existing data within a table.
- DELETE – is used to delete records from a database table.

DATABASE ADMINISTRATOR

the people responsible for managing databases are called database administrators. Each database administrator, dubbed DBA for the sake of brevity may be engaged in performing various database manipulation tasks such as archiving, testing, running, security control etc. all related to the environmental side of the databases.

The responsibilities of a DBA

1. Designing the logical and physical schemas, as well as widely-used portions of the external schema.

2. Security and authorization.

3. Data availability and recovery from failures.

4. Database tuning: The DBA is responsible for evolving the database, in particular the conceptual and physical schemas to ensure adequate performance as user requirements change.

USERS IN DBMS

Users are of 4 types:

1. Application programmers or Ordinary users

2. End users

3. Database Administrator (DBA)

4. System Analyst

1. Application programmers

or Ordinary users: These users write application programs to interact with the database. Application programs can be written in some programming language such a COBOL, PL/I, C++, JAVA or some higher level fourth generation language. Such programs access the database by issuing the appropriate request, typically a SQL statement to DBMS.

2. End Users:

End users are the users, who use the applications developed. End users need not know about the working, database design, the access mechanism etc. They just use the system to get their task done. End users are of two types:

a) Direct users

b) Indirect users

a) Direct users: Direct users are the users who se the computer, database system directly, by following instructions provided in the user interface. They interact using the application programs already developed, for getting the desired result. E.g. People at railway reservation counters, who directly interact with database.

b) Indirect users: Indirect users are those users, who desire benefit form the work of DBMS indirectly. They use the outputs generated by the programs, for decision making or any other purpose. They are just concerned with the output and are not bothered about the programming part.

3. Database Administrator (DBA):

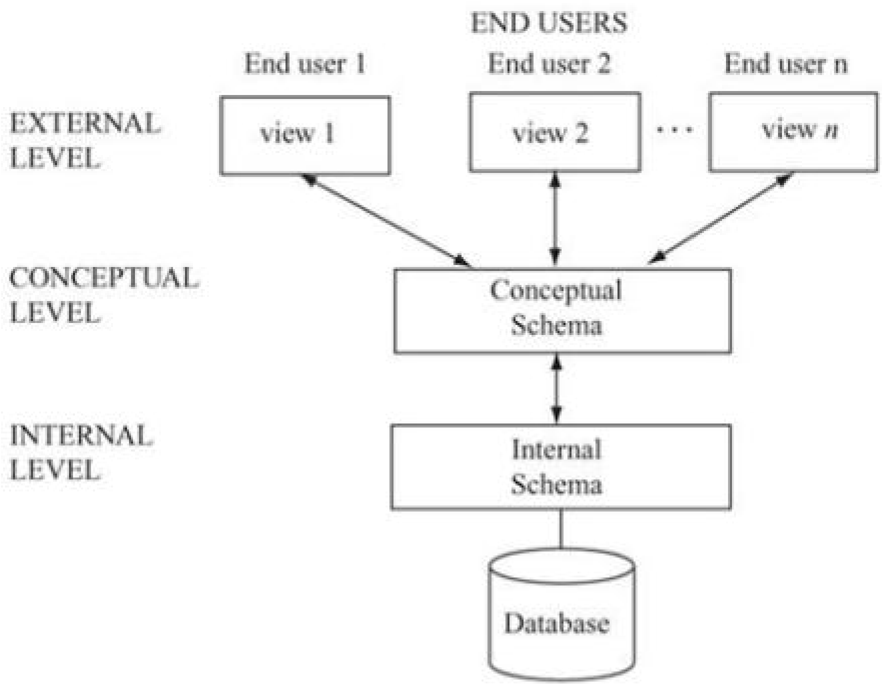
Database Administrator (DBA) is the person which makes the strategic and policy decisions regarding the data of the enterprise, and who provide the necessary technical support for implementing these decisions. Therefore, DBA is responsible for overall control of the system at a technical level. In database environment, the primary resource is the database itself and the secondary resource is the DBMS and related software administering these resources is the responsibility of the Database Administrator (DBA).

4. System Analyst:

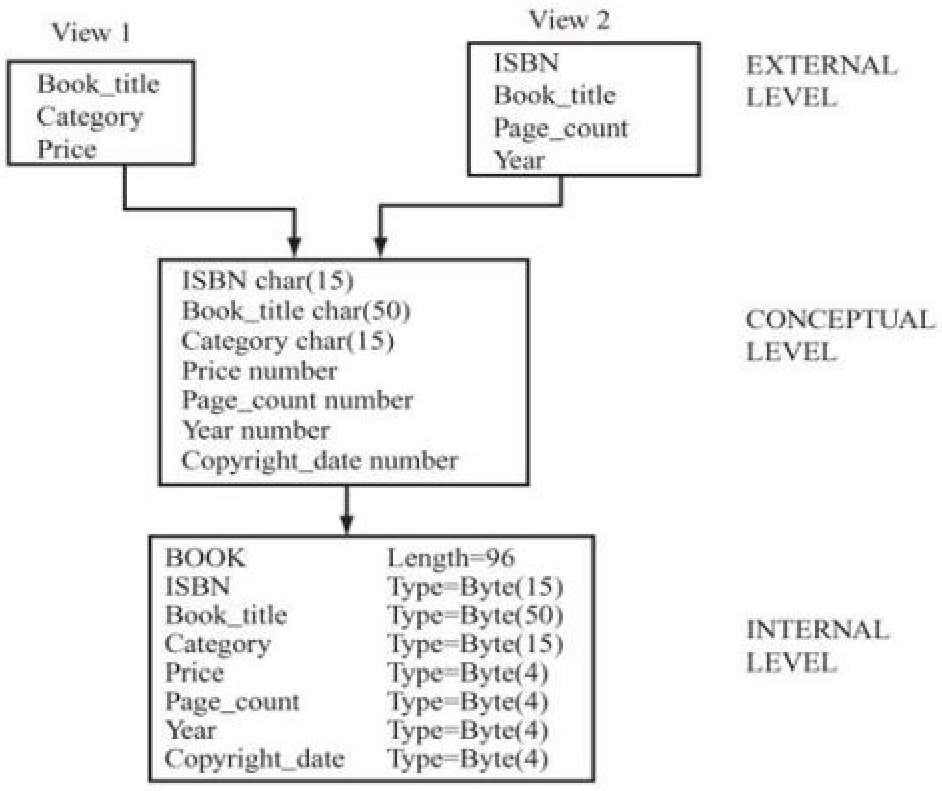
System Analyst determines the requirement of end users, especially naive and parametric end users and develops specifications for transactions that meet these requirements. System Analyst plays a major role in database design, its properties; the structure prepares the system requirement statement, which involves the feasibility aspect, economic aspect, technical aspect etc. of the system.

Overall structure

3 LEVEL ARCHITECTURE / THREE-SCHEMA ARCHITECTURE: In this architecture, the overall database description can be defined at three levels namely internal, conceptual, and external levels. This is shown below:



Can be easily understandable by



Internal level:

It is the lowest level of data abstraction that deals with the physical representation of the database on the computer and thus, is also known as physical level. It describes how the data is physically stored and organized on the storage medium. At this level, various aspects are considered to achieve optimal runtime performance and storage space utilization. These aspects include storage space allocation techniques for data and indexes, access paths such as indexes, data compression and encryption techniques, and record placement.

Conceptual level:

This level of abstraction deals with the logical structure of the entire database and it is also known as logical level. It describes what data is stored in the database, the relationships among the data and complete view of the user's requirements without any concern for the physical implementation. It hides the complexity of physical storage structures. The conceptual view is the overall view of the database and it includes all the information that is going to be represented in the database.

External level:

It is the highest level of abstraction that deals with the user's view of the database and it is also known as view level. Most of the users and application programs do not require the entire data stored in the database. The external level describes a part of the database for a particular group of users. It permits users to access data in a way that is customized according to their needs, so that the same data can be seen by different users in different ways at the same time. It provides a powerful and flexible security mechanism by hiding the parts of the database from certain users, as the user is not aware of existence of any attributes that are missing from the view.

Difference between file system and dbms(DBMS vs file system)

File system	DBMS
File system is a collection of data. Any management with the file system, user has to write the procedures.	DBMS is a collection of data and user is not required to write the procedures for managing the database.
File system gives the details of the data representation and Storage of data.	DBMS provides an abstract view of data that hides the details.
In File system storing and retrieving of data cannot be done efficiently.	DBMS is efficient to use since there are wide varieties of sophisticated techniques to store and retrieve the data.

Concurrent access to the data in the file system has many problems like : Reading the file while other form of locking. deleting some information, updating some information	DBMS takes care of Concurrent access using some form of locking.
File system doesn't provide crash recovery mechanism. Eg. While we are entering some data into the file if System crashes then content of the file is lost	DBMS has crash recovery mechanism, DBMS protects user from the effects of system failures.
Protecting a file under file system is very difficult.	DBMS has a good protection mechanism.

SCHEMA

A schema is a collection of named objects. Schemas are generally stored in a data dictionary. Although a schema is defined in text database language, the term is often used to refer to a graphical depiction of the database structure. In relational database technology, schemas provide a logical classification of objects in the database. Some of the objects that a schema may contain include tables, views, aliases, indexes, triggers, and structured types.

INSTANCES

The data in the database at a particular moment of time is called an instance or a database state. In a given instance, each schema construct has its own current set of instances. Many instances or database states can be constructed to correspond to a particular database schema. Every time we update (i.e., insert, delete or modify) the value of a data item in a record, one state of the database changes into another state. The following figure shows an instance of the ITEM relation in a database schema.

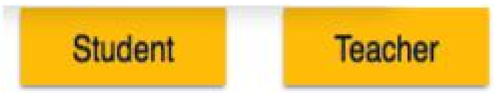
ITEM		
ITEM-ID	ITEM_DESC	ITEM_COST
1111A	Nutt	3
1112A	Bolt	5

ENTITY RELATIONSHIP MODEL & BASIC CONCEPTS:

It is a semantic data model that is used for the graphical representation of the conceptual database design. Entity relationship model defines the conceptual view of database. It works around real world entity and association among them. At view level, ER model is considered well for designing databases. The Entity Relationship (ER) model consists of different types of entities. The existence of an entity may depends on the existence of one or more other entities, such an entity is said to be existence dependent. Entities whose existence not depending on any other entities is termed as not existence dependent. Entities based on their characteristics are classified as follows.

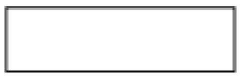
ENTITY:

A real-world thing either animate or inanimate that can be easily identifiable and distinguishable, called entity. Entities are represented by means of rectangles. Rectangles are named with the entity set they represent.

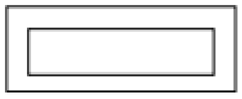


TYPES OF ENTITY:

Strong entity: An entity set that has a primary key is called as Strong entity set. Strong entity represented by rectangle which is shown below.

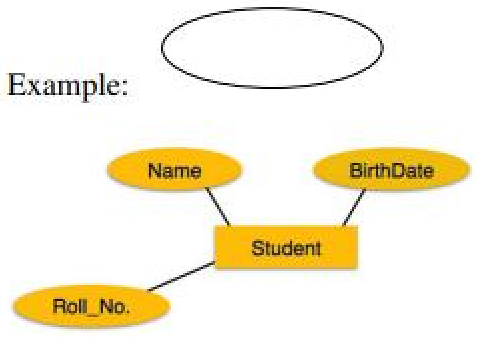


Weak entity: An entity set that does not have sufficient attributes to form a primary key is termed as a weak entity set. Weak entity represented by double rectangle which is shown below.



ATRIBUTE:

Attributes are properties of entities. Attributes are represented by means of ellipse. Every ellipse represents one attribute and is directly connected to its entity (rectangle). For example the employee is the entity and employee's name, age, address, salary and job etc are the attribute. Attribute is represented by ellipse.



TYPES OF ATTRIBUTES:

Simple attribute:

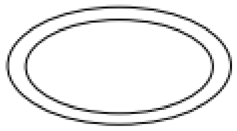
Simple attribute consists of a single atomic value. A simple attribute cannot be subdivided. For example the attributes age, sex etc is simple attributes. Composite attribute: A composite attribute is an attribute that can be further subdivided. For example the attribute ADDRESS can be subdivided into street, city, state, and zip code.

Single-valued attribute:

A single valued attribute can have only a single value. For example a person can have only one 'date of birth', 'age' etc. That is a single valued attributes can have only single value. But it can be simple or composite attribute. That is 'date of birth' is a composite attribute; 'age' is a simple attribute. But both are single valued attributes.

Multi-value attributes:

Multivalve attributes can have multiple values. For instance a person may have multiple phone numbers, multiple degrees etc. Multivalve attributes are shown by a double line connecting to the entity in the ER diagram.

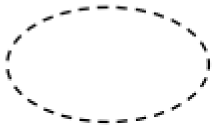


Stored attribute:

The value for the derived attribute is derived from the stored attribute. For example 'Date of birth' of a person is a stored attribute. The value for the attribute 'AGE' can be derived by subtracting the 'Date of Birth'(DOB) from the current date. Stored attribute supplies a value to the related attribute.

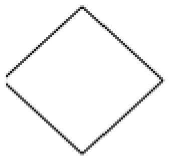
Derived attribute:

An attribute that's value is derived from a stored attribute. Example: age, and it's value is derived from the stored attribute Date of Birth. It is represented by dotted ellipse.

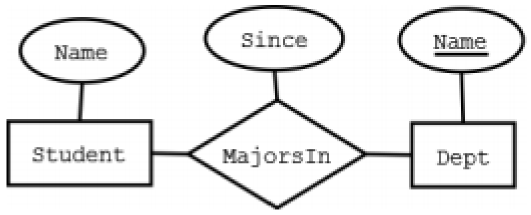


RELATIONSHIP:

The association among entities is called relationship. For example, employee entity has relation works_at with department. Another example is for student who enrolls in some course. Works_at and Enrolls are called relationship. Relationship is represented by diamond box.



Example: Student (entity type) is related to Department (entity type) by MajorsIn (relationship type).

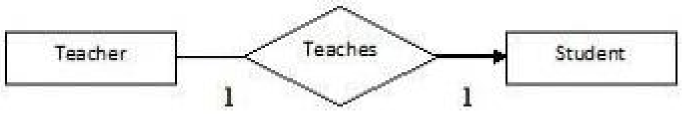


MAPPING CONSTRAINTS / CARDINALITY:

While creating relationship between two entities. We may often need to face the cardinality problem. This simply means that how many entities of the first set are related to how many entities of the second set. Cardinality can be of the following four types.

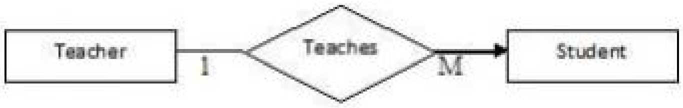
One-to-One:

Only one entity of the first set is related to only one entity of the second set. Example A teacher teaches a student. Only one teacher is teaching only one student.



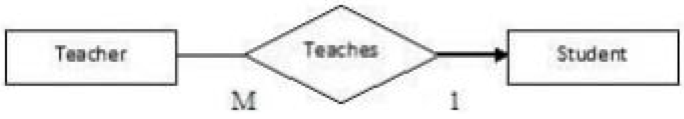
One-to-Many:

Only one entity of the first set is related to multiple entities of the second set. Example A teacher teaches students. Only one teacher is teaching many students.



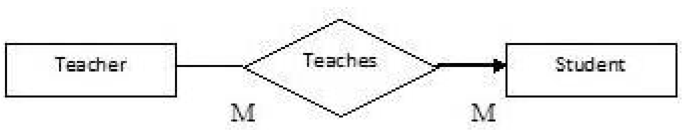
Many-to-One:

Multiple entities of the first set are related to multiple entities of the second set. Example Teachers teach a student. Many teachers are teaching only one student.



Many-to-Many:

Multiple entities of the first set is related to multiple entities of the second set. Example Teachers teach students. In any school or college many teachers are teaching many students. This can be considered as a two way one-to-many relationship.



KEYS:

A key is an attribute of a table which helps to identify a row. There can be many different types of keys:

Super Key or Candidate Key:

It is such an attribute of a table that can uniquely identify a row in a table. Generally they contain unique values and can never contain NULL values. There can be more than one super key or candidate key in a table Example within a STUDENT table Roll and Mobile No. can both serve to uniquely identify a student.

Primary Key:

It is one of the candidate keys that are chosen to be the identifying key for the entire table. Example although there are two candidate keys in the STUDENT table, the college would obviously use Roll as the primary key of the table.

Alternate Key:

This is t1he candidate key which is not chosen as the primary key of the table. They are named so because although not the primary key, they can still identify a row.

Composite Key:

Sometimes one key is not enough to uniquely identify a row. Example in a single class Roll is enough to find a student but in the entire school merely searching by the Roll is not enough because there could be 10 classes in the school and each one of them may contain a certain roll no 5. To uniquely identify the student we have to say something like “class VII, roll no 5”. So a combination of two or more attributes is combined to create a unique combination of values such as Class + Roll.

Foreign Key:

Sometimes we may have to work with an attribute that does not have a primary key of its own. To identify its rows, we have to use the primary attribute of a related table. Such a copy of another related table’s primary key is called foreign key

Unit-II

E-R DIAGRAM (ERD):

An entity-relationship diagram (ERD) is a data modeling technique that graphically illustrates an information system’s entities and the relationships between those entities. An ERD is a conceptual and representational model of data used to represent the entity framework infrastructure.

The elements of an ERD are:

Entities:

A real-world thing either animate or inanimate that can be easily identifiable and distinguishable.

Attributes:

Entities are represented by means of their properties, called attributes.

Relationships:

The association among entities is called relationship.

Creating an E-R DIAGRAM includes:

1. Identifying and defining the entities

2. Determining all interactions between the entities

3. Analyzing the nature of interactions/determining the cardinality of the relationships

4. Creating the ERD

DESIGN OF AN E-R DATABASE SCHEMA:

Database design is the process of producing a detailed data model of a database. This logical data model contains all the needed logical and physical design choices and physical storage parameters needed to generate a design in a data definition language, which can then be used to create a database. A fully attributed data model contains detailed attributes for each entity. The term database design can be used to describe many different parts of the design of an overall database system. Principally, and most correctly, it can be thought of as the logical design of the base data structures used to store the data. In the relational model these are the tables and view. In an object database the entities and relationships map directly to object classes and named relationships. However, the term database design could also be used to apply to the overall process of designing, not just the base data structures, but also the forms and queries used as part of the overall database application within the database management system

1. Determine the purpose of the database - This helps prepare for the remaining steps.

2. Find and organize the information required - Gather all of the types of information to record in the database, such as product name and order number.

3. Divide the information into tables - Divide information items into major entities or subjects, such as Products or Orders. Each subject then becomes a table.

4. Turn information items into columns - Decide what information needs to be stored in each table. Each item becomes a field, and is displayed as a column in the table. For example, an Employees table might include fields such as Last Name and Hire Date.

5. Specify primary keys - Choose each table’s primary key. The primary key is a column, or a set of columns, that is used to uniquely identify each row. An example might be Product ID or Order ID.

6. Set up the table relationships - Look at each table and decide how the data in one table is related to the data in other tables. Add fields to tables or create new tables to clarify the relationships, as necessary.

7. Refine the design - Analyze the design for errors. Create tables and add a few records of sample data. Check if results come from the tables as expected. Make adjustments to the design, as needed. 8. Apply the normalization rules - Apply the data normalization rules to see if tables are structured correctly. Make adjustments to the tables

REDUCTION OF E-R SCHEMA TO TABLE:

A database which conforms to an E R diagram can be represented by collection of table's .For each entity set and for each relationship set in the database, we will create unique tables, which is assigned the name of the corresponding entity set or relationship sets. Each table has a no. of columns which have unique names. Each row in the table corresponds to an entity or a relationship. A database that conforms to an E-R database schema can be represented by a collection of tables. For each entity set and for each relationship set, there is a unique table. A table is a chart with rows and columns. The set of all possible rows is the Cartesian product of all columns. A row is also known as a tuple or a record. A table has an unlimited number of rows. Each column is also known as a field. There are few points to reduction of ER schema.

1. Strong Entity Sets: It is common practice for the table to have the same name as the entity set. There is one column for each attribute.

2. Weak Entity Sets: There is one column for each attribute, plus the attribute(s) the form the primary key of the strong entity set that the weak entity set depends upon.

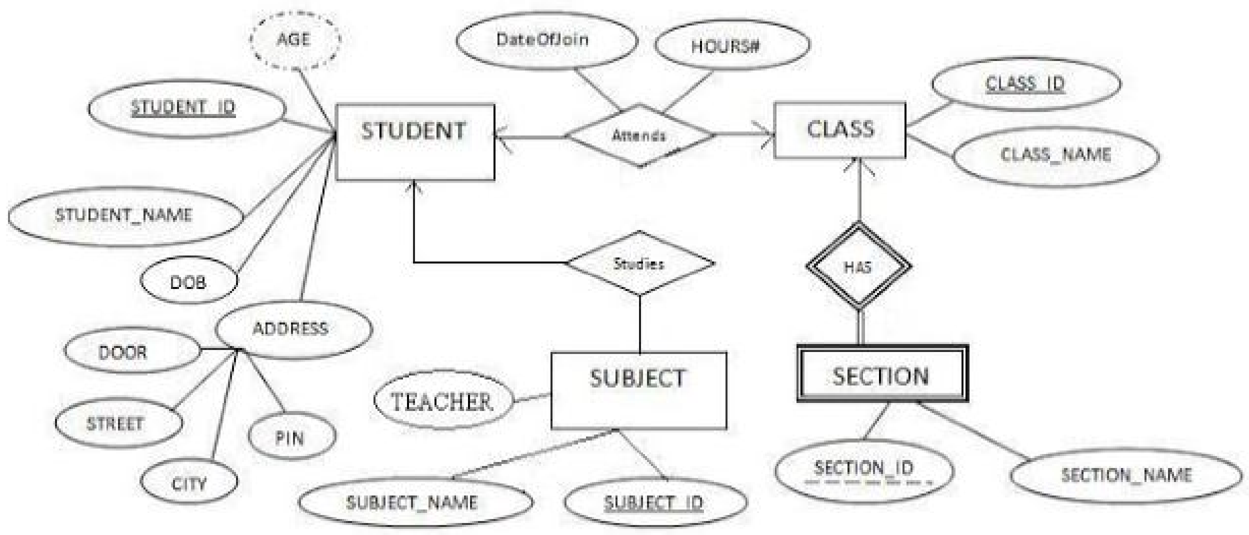
3. Relationship Sets: We represent a relationship with a table that includes the attributes of each of the primary keys plus any descriptive attributes (if any). There is a problem that if one of the entities in the relationship is a weak entity set. There would be no unique information in the relationship table and therefore may be omitted. Another problem can occur if there is an existence dependency. In that case you can combine the two tables.

4. Multivalve Attributes: When an attribute is multivalve, remove the attribute from the table and create a new table with the primary key and the attribute, but each value will be a separate row.

5. Generalization: Create a table for the higher-level entity set. For each lower-level entity set, create a table with the attributes for that specialization and include the primary key from the higher-level entity set.

Reduce the ER diagram to relation table solved exercise
How to reduce an ER diagram to tables? Reduction of ERD to relation schema, Convert Entity Relationship diagram to set of tables, ERD to relation schema examples, mapping ER diagram to relational tables

Reduce (convert) the following ER diagram to relational schema



ER components	Given component	Result
Strong Entity Set *Rule: Strong entity set can be directly converted into table.*	(a) STUDENT (b) SUBJECT (c) CLASS	(a) STUDENT (Student_ID, Student_Name, DOB, Address) (b) SUBJECT (Subject_ID, Subject_Name, Teacher) (C) CLASS (Class_ID, Class_Name)
Derived attribute *Rule: No need to create a column in the table for derived attribute.*	Age in STUDENT table	No changes
Composite attribute *Rule: Replace the composite attribute with its component attributes.*	Address in STUDENT table	**STUDENT (Student_ID, Student_Name, DOB, Door, Street, City, Pin)**
1-1, 1-n, and n-1 Relationships *Rule: Include the*	Attends (1-1 from STUDENT to CLASS)	CLASS (Class_ID, Class_Name, Student_ID) **SUBJECT (Subject_ID,**

primary key of one side entity set as the foreign key of other side entity set.	Studies (1-n from STUDENT to SUBJECT)	**Subject_Name, Teacher, Student_ID)**
Descriptive attribute *Rule: An attribute that is part of a relationship is descriptive. Include the descriptive attributes to 1 side as shown above.*	DateOfJoin, Hours# of Attends relationship.	**CLASS (Class_ID, Class_Name, Student_ID, DateOfJoin, Hours#)**
Weak entity set *Rule: Weak entity set is totally participated (existence dependent) on the strong entity set. Include the primary key of strong entity set into the weak entity set as foreign key.*	(d) SECTION	**SECTION (Section_ID, Section_Name, Class_ID)**
Weak relationship *Rule: No need to create as a table. If created, then the table is redundant.*	Has	No changes

Final set of relation schemas: (Primary keys are underlined)

STUDENT (<u>Student_ID</u>, Student_Name, DOB, Door, Street, City, Pin)

CLASS (<u>Class_ID</u>, Class_Name, Student_ID, DateOfJoin, Hours#)

Student_ID is the foreign key refers STUDENT table

SUBJECT (<u>Subject_ID</u>, Subject_Name, Teacher, Student_ID)

Student_ID is the foreign key refers STUDENT table

SECTION (<u>Section_ID, Class_ID</u>, Section_Name)

Class_ID is the foreign key refers CLASS table

Generalization, Specialization and Aggregation in ER Model

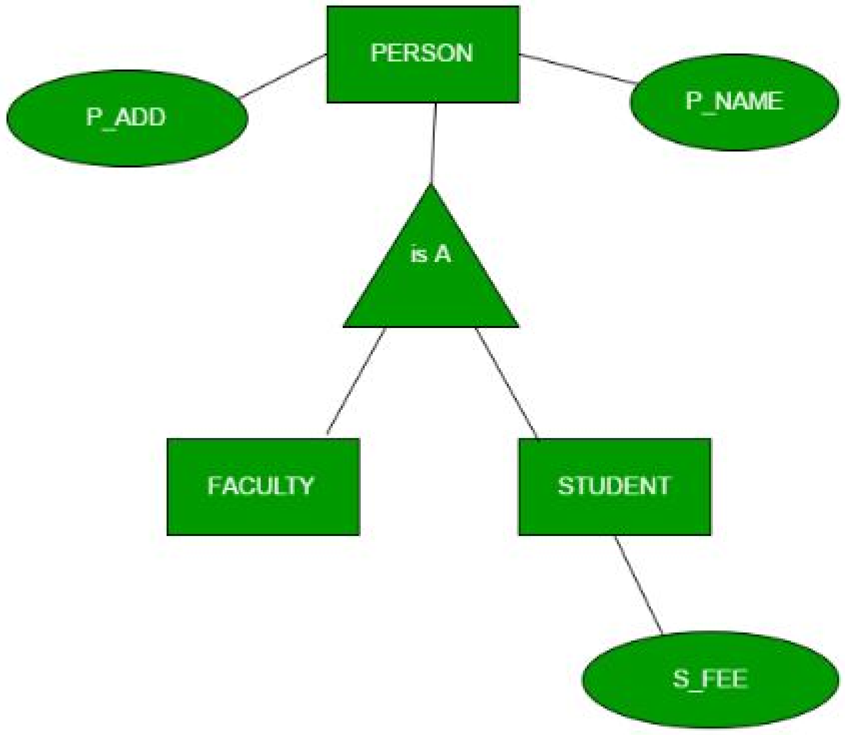
Prerequisite

– Introduction of ER Model

Generalization, Specialization and Aggregation in ER model are used for data abstraction in which abstraction mechanism is used to hide details of a set of objects.

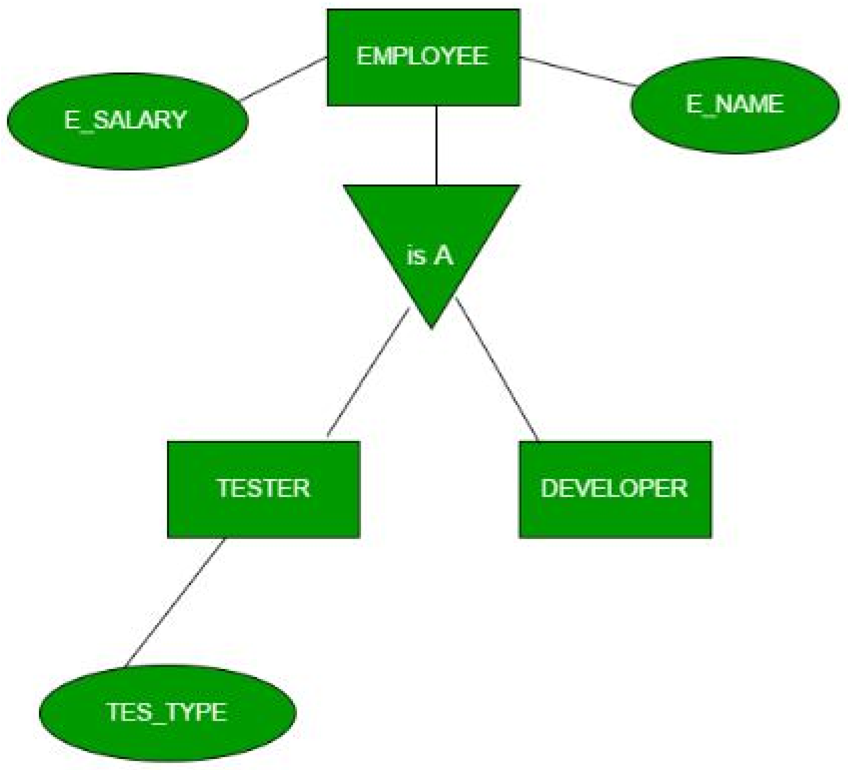
Generalization

Generalization is the process of extracting common properties from a set of entities and create a generalized entity from it. It is a bottom-up approach in which two or more entities can be generalized to a higher level entity if they have some attributes in common. For Example, STUDENT and FACULTY can be generalized to a higher level entity called PERSON as shown in Figure 1. In this case, common attributes like P_NAME, P_ADD become part of higher entity (PERSON) and specialized attributes like S_FEE become part of specialized entity (STUDENT).



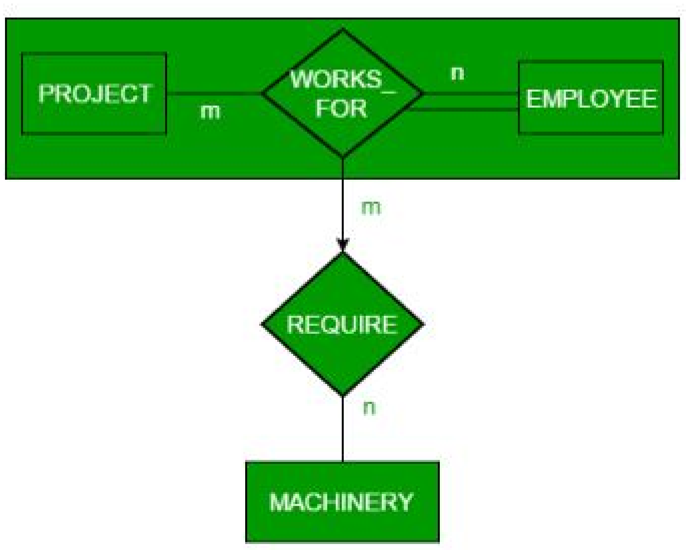
Specialization

In specialization, an entity is divided into sub-entities based on their characteristics. It is a top-down approach where higher level entity is specialized into two or more lower level entities. For Example, EMPLOYEE entity in an Employee management system can be specialized into DEVELOPER, TESTER etc. as shown in Figure 2. In this case, common attributes like E_NAME, E_SAL etc. become part of higher entity (EMPLOYEE) and specialized attributes like TES_TYPE become part of specialized entity (TESTER).

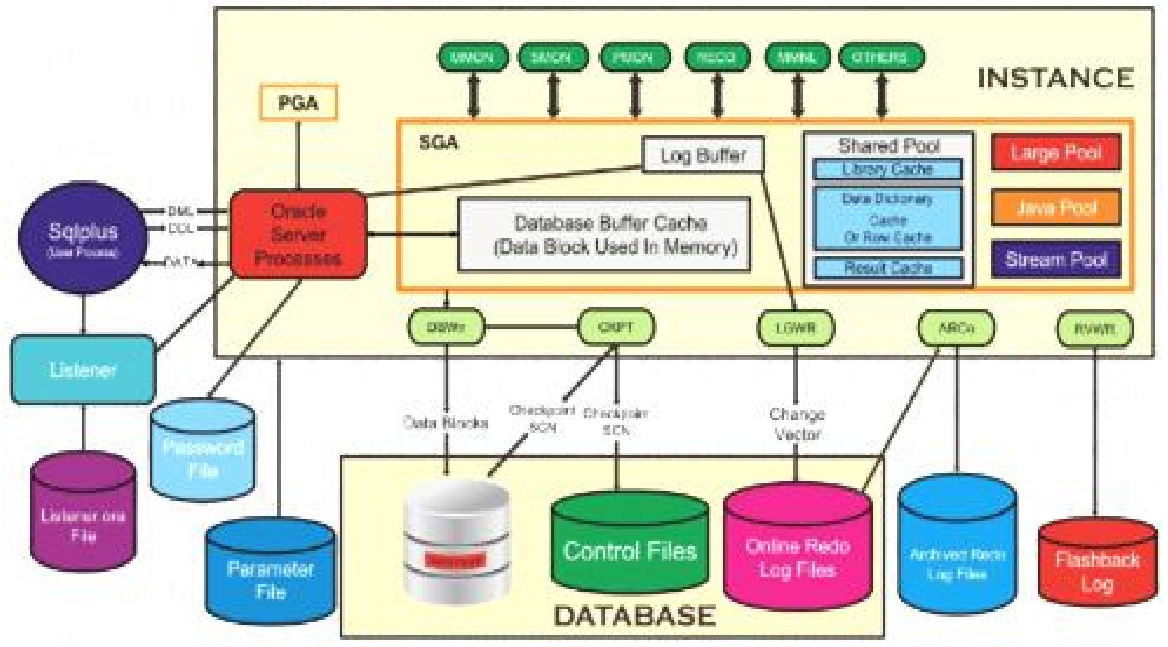


Aggregation

An ER diagram is not capable of representing relationship between an entity and a relationship which may be required in some scenarios. In those cases, a relationship with its corresponding entities is aggregated into a higher level entity. For Example, Employee working for a project may require some machinery. So, REQUIRE relationship is needed between relationship WORKS_FOR and entity MACHINERY. Using aggregation, WORKS_FOR relationship with its entities EMPLOYEE and PROJECT is aggregated into single entity and relationship REQUIRE is created between aggregated entity and MACHINERY.



Oracle Architecture



Oracle server: An Oracle server includes an **Oracle Instance** and an **Oracle database**.

· An Oracle database includes several different types of files: datafiles, control files, redo log files and archive redo log files. The Oracle server also accesses parameter files and password files.

· This set of files has several purposes.

o One is to enable system users to process SQL statements.

o Another is to improve system performance.

o Still another is to ensure the database can be recovered if there is a software/hardware failure.

· The database server must manage large amounts of data in a multi-user environment.

· The server must manage concurrent access to the same data.

· The server must deliver high performance. This generally means fast response times.

Oracle instance:

An Oracle Instance consists of **two** different sets of components:

· The first component set is the set of **background processes** (PMON, SMON, RECO, DBW0, LGWR, CKPT, D000 and others).

o These will be covered later in detail – each background process is a computer program.

o These processes perform input/output and monitor other Oracle processes to provide good performance and database reliability.

· The second component set includes the **memory structures** that comprise the Oracle instance.

o When an instance starts up, a memory structure called the System Global Area (SGA) is allocated.

o At this point the background processes also start.

· An Oracle Instance provides access to one and only one Oracle database.

Oracle database:

An Oracle database consists of files.

· Sometimes these are referred to as operating system files, but they are actually **database files** that store the database information that a firm or organization needs in order to operate.

· The **redo log files** are used to recover the database in the event of application program failures, instance failures and other minor failures.

· The **archived redo log files** are used to recover the database if a disk fails.

· Other files not shown in the figure include:

o The required **parameter file** that is used to specify parameters for configuring an Oracle instance when it starts up.

o The optional **password file** authenticates special users of the database – these are termed **privileged users** and include database administrators.

o **Alert** and **Trace Log Files** – these files store information about errors and actions taken that affect the configuration of the database.

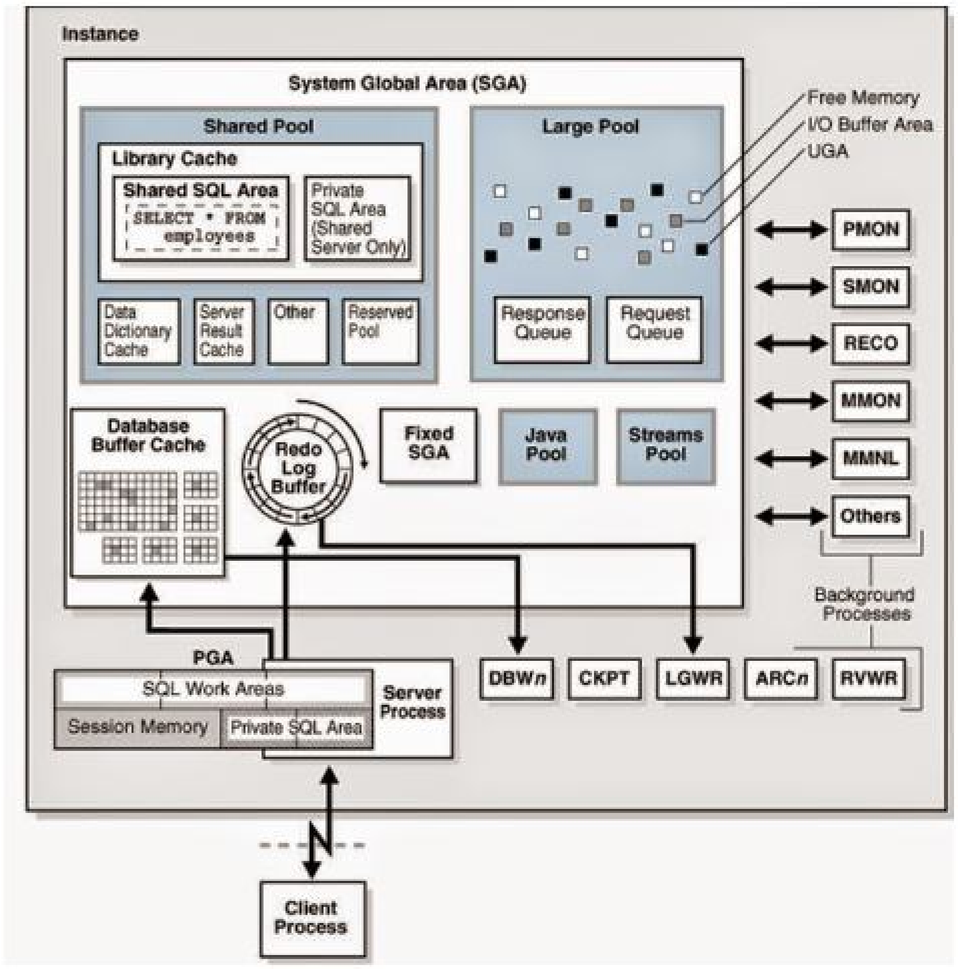
User and server processes:

The processes shown in the figure are called **user** and **server processes**. These processes are used to manage the execution of SQL statements.

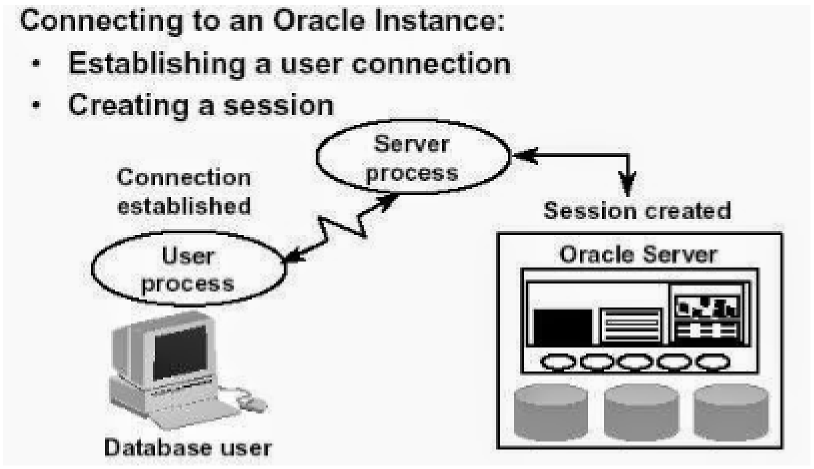
· A **Shared Server Process** can share memory and variable processing for multiple user processes.

· A **Dedicated Server Process** manages memory and variables for a single user process.

This figure from the *Oracle Database Administration Guide* provides another way of viewing the **SGA**.



Connecting to an Oracle Instance – Creating a Session



System users can connect to an Oracle database through SQLPlus or through an application program like the Internet Developer Suite (the program becomes the system user). This connection enables users to execute SQL statements.

The act of connecting creates a communication pathway between a user process and an Oracle Server. As is shown in the figure above, the User Process communicates with the Oracle Server through a Server Process. The User Process executes on the client computer. The Server Process executes on the server computer, and actually executes SQL statements submitted by the system user.

The figure shows a one-to-one correspondence between the User and Server Processes. This is called a **Dedicated Server** connection. An alternative configuration is to use a **Shared Server** where more than one User Process shares a Server Process.

<u>Sessions</u>: When a user connects to an Oracle server, this is termed a session. The **User Global Area** is session memory and these memory structures are described later in this document. The session starts when the Oracle server validates the user for connection. The session ends when the user logs out (disconnects) or if the connection terminates abnormally (network failure or client computer failure).

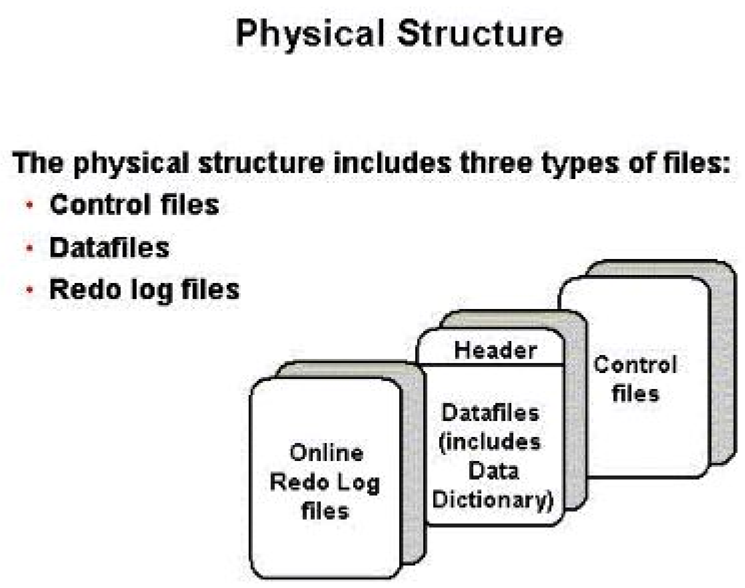
A user can typically have more than one concurrent session, e.g., the user may connect using SQLPlus and also connect using Internet Developer Suite tools at the same time. The limit of concurrent session connections is controlled by the DBA.

If a system users attempts to connect and the Oracle Server is not running, the system user receives the **Oracle Not Available** error message.

Physical Structure – Database Files

As was noted above, an Oracle database consists of physical files. The database itself has:
· **Datafiles** – these contain the organization's actual data.
· **Redo log files** – these contain a chronological record of changes made to the database, and enable recovery when failures occur.

· **Control files** – these are used to synchronize all database activities and are covered in more detail in a later module.



Other key files as noted above include:

· Parameter file – there are two types of parameter files.

o The **init.ora** file (also called the **PFILE**) is a **static parameter file**. It contains parameters that specify how the database instance is to start up. For example, some parameters will specify how to allocate memory to the various parts of the system global area.

o The **spfile.ora** is a **dynamic parameter file**. It also stores parameters to specify how to startup a database; however, its parameters can be modified while the database is running.

· Password file – specifies which *special* users are authenticated to startup/shut down an Oracle Instance.

· Archived redo log files – these are copies of the redo log files and are necessary for recovery in an online, transaction-processing environment in the event of a disk failure.

Memory Management and Memory Structures

Oracle Database Memory Management

Memory management - focus is to maintain optimal sizes for memory structures.

· Memory is managed based on memory-related initialization parameters.

· These values are stored in the init.ora file for each database.

Three basic options for memory management are as follows:

· **Automatic memory management**:

o DBA specifies the target size for instance memory.

o The database instance automatically tunes to the target memory size.

o Database redistributes memory as needed between the SGA and the instance PGA.

· **Automatic shared memory management**:

o This management mode is partially automated.

o DBA specifies the target size for the SGA.

o DBA can optionally set an aggregate target size for the PGA or managing PGA work areas individually.

· **Manual memory management**:

o Instead of setting the total memory size, the DBA sets many initialization parameters to manage components of the SGA and instance PGA individually.

If you create a database with Database Configuration Assistant (DBCA) and choose the basic installation option, then automatic memory management is the default.

The memory structures include three areas of memory:

· System Global Area (SGA) – this is allocated when an Oracle Instance starts up.

· Program Global Area (PGA) – this is allocated when a Server Process starts up.

· User Global Area (UGA) – this is allocated when a user connects to create a session.

System Global Area

The **SGA** is a read/write memory area that stores information shared by all database processes and by all users of the database (sometimes it is called the **Shared Global Area**).

- This information includes both organizational data and control information used by the Oracle Server.
- The SGA is allocated in memory and virtual memory.
- The size of the SGA can be established by a DBA by assigning a value to the parameter **SGA_MAX_SIZE** in the parameter file—this is an optional parameter.

The SGA is allocated when an Oracle instance (database) is started up based on values specified in the initialization parameter file (either PFILE or SPFILE).

The SGA has the following mandatory memory structures:

- Database Buffer Cache
- Redo Log Buffer
- Java Pool
- Streams Pool
- Shared Pool – includes two components:
  - Library Cache
  - Data Dictionary Cache
- Other structures (for example, lock and latch management, statistical data)

Additional optional memory structures in the SGA include:

- Large Pool

The **SHOW SGA** SQL command will show you the SGA memory allocations.

· This is a recent clip of the SGA for the DBORCL database at SIUE.

· In order to execute SHOW SGA you must be connected with the special privilege **SYSDBA** (which is only available to user accounts that are members of the DBA Linux group).

Early versions of Oracle used a **Static SGA**. This meant that if modifications to memory management were required, the database had to be shutdown, modifications were made to the **init.ora** parameter file, and then the database had to be restarted.

Oracle 11g uses a **Dynamic SGA**. Memory configurations for the system global area can be made without shutting down the database instance. The DBA can resize the Database Buffer Cache and Shared Pool dynamically.

Several initialization parameters are set that affect the amount of random access memory dedicated to the SGA of an Oracle Instance. These are:

· **SGA_MAX_SIZE**: This optional parameter is used to set a limit on the amount of **virtual memory** allocated to the SGA – a typical setting might be **1 GB**; however, if the value for SGA_MAX_SIZE in the initialization parameter file or server parameter file is less than the sum the memory allocated for all components, either explicitly in the parameter file or by default, at the time the instance is initialized, then the database ignores the setting for SGA_MAX_SIZE. For optimal performance, the entire SGA should fit in real memory to eliminate paging to/from disk by the operating system.

· **DB_CACHE_SIZE**: This optional parameter is used to tune the amount memory allocated to the Database Buffer Cache in standard database blocks. Block sizes vary among operating systems. The DBORCL database uses **8 KB** blocks. The total blocks in the cache defaults to **48 MB** on LINUX/UNIX and **52 MB** on Windows operating systems.

· **LOG_BUFFER**: This optional parameter specifies the number of bytes allocated for the Redo Log Buffer.

· **SHARED_POOL_SIZE**: This optional parameter specifies the number of bytes of memory allocated to shared SQL and PL/SQL. The default is **16 MB**. If the operating system is based on a **64 bit** configuration, then the default size is **64 MB**.

- **LARGE_POOL_SIZE**: This is an optional memory object – the size of the Large Pool defaults to zero. If the init.ora parameter **PARALLEL_AUTOMATIC_TUNING** is set to **TRUE**, then the default size is automatically calculated.
- **JAVA_POOL_SIZE**: This is another optional memory object. The default is **24 MB** of memory.

The size of the SGA cannot exceed the parameter **SGA_MAX_SIZE** minus the combination of the size of the additional parameters, **DB_CACHE_SIZE**, **LOG_BUFFER**, **SHARED_POOL_SIZE**, **LARGE_POOL_SIZE**, and **JAVA_POOL_SIZE**.

Memory is allocated to the SGA as contiguous virtual memory in units termed granules. Granule size depends on the estimated total size of the SGA, which as was noted above, depends on the SGA_MAX_SIZE parameter. Granules are sized as follows:

- If the SGA is less than **1 GB** in total, each granule is **4 MB**.
- If the SGA is greater than **1 GB** in total, each granule is **16 MB**.

Granules are assigned to the Database Buffer Cache, Shared Pool, Java Pool, and other memory structures, and these memory components can dynamically grow and shrink. Using contiguous memory improves system performance. The actual number of granules assigned to one of these memory components can be determined by querying the database view named **V$BUFFER_POOL**.

Granules are allocated when the Oracle server starts a database instance in order to provide memory addressing space to meet the SGA_MAX_SIZE parameter. The minimum is 3 granules: one each for the fixed SGA, Database Buffer Cache, and Shared Pool. In practice, you'll find the SGA is allocated much more memory than this. The SELECT statement shown below shows a current_size of 1,152 granules.

```
SELECT name, block_size, current_size, prev_size, prev_buffers

FROM v$buffer_pool;


NAME             BLOCK_SIZE CURRENT_SIZE  PREV_SIZE PREV_BUFFERS

-------------------- ---------- ------------ ---------- ------------
```

```
DEFAULT              8192      560      576      71244
```

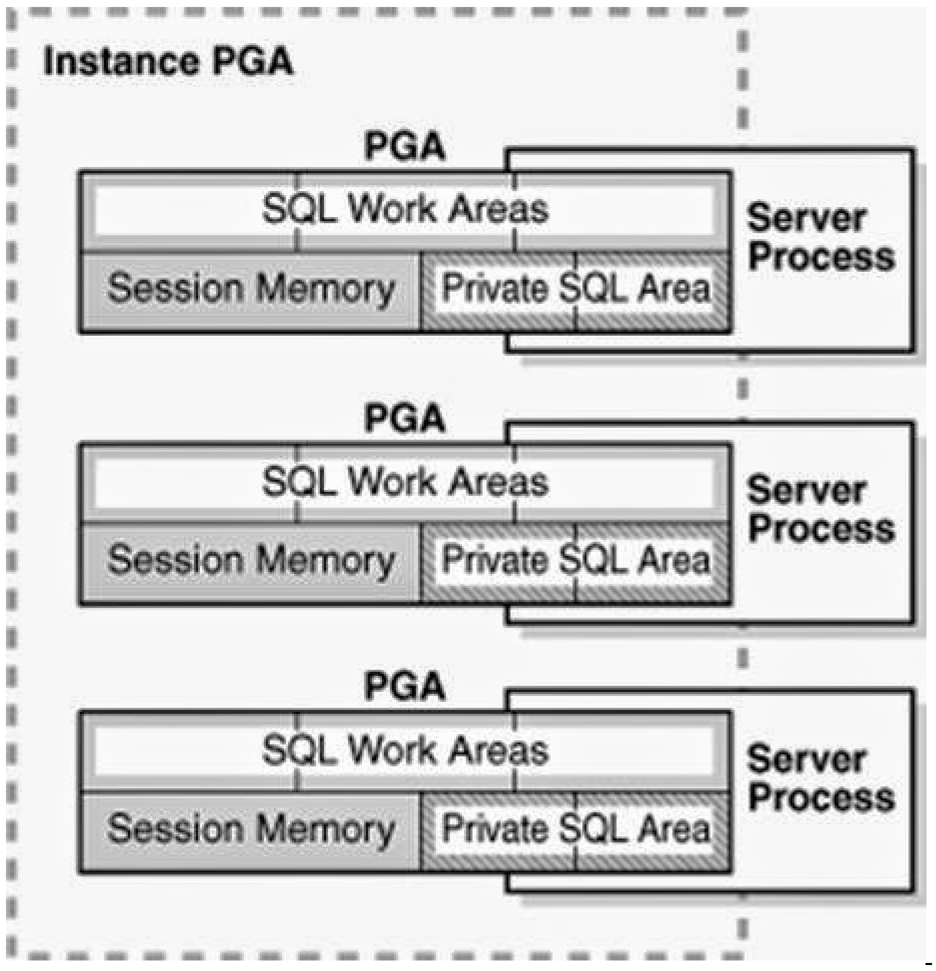
For additional information on the dynamic SGA sizing, enroll in Oracle's *Oracle11g Database Performance Tuning* course.

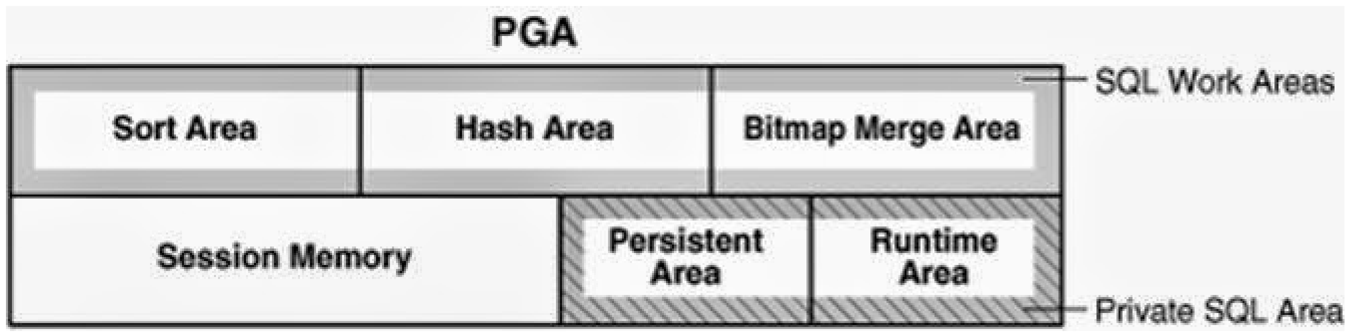
Program Global Area (PGA)

A PGA is:

- a **nonshared** memory region that contains data and control information exclusively for use by an Oracle process.
- A PGA is created by Oracle Database when an Oracle process is started.
- One PGA exists for each **Server Process** and each **Background Process**. It stores data and control information for a single **Server Process** or a single **Background Process**.
- It is allocated when a process is created and the memory is scavenged by the operating system when the process terminates. This is **NOT** a shared part of memory – one PGA to each process only.
- The collection of individual PGAs is the total instance PGA, or instance PGA.
- Database initialization parameters set the size of the instance PGA, not individual PGAs.

The **Program Global Area** is also termed the **Process Global Area (PGA**) and is a part of memory allocated that is outside of the **Oracle Instance**.





The content of the PGA varies, but as shown in the figure above, generally includes the following:

- **Private SQL Area:** Stores information for a parsed SQL statement – stores bind variable values and runtime memory allocations. A user session issuing SQL statements has a Private SQL Area that may be associated with a Shared SQL Area if the same SQL statement is being executed by more than one system user. This often happens in OLTP environments where many users are executing and using the same application program.

- o **Dedicated Server environment** – the Private SQL Area is located in the Program Global Area.
- o **Shared Server environment** – the Private SQL Area is located in the System Global Area.

· **Session Memory**: Memory that holds session variables and other session information.

· **SQL Work Areas**: Memory allocated for sort, hash-join, bitmap merge, and bitmap create types of operations.

o Oracle 9i and later versions enable automatic sizing of the SQL Work Areas by setting the **WORKAREA_SIZE_POLICY = AUTO** parameter (this is the default!) and **PGA_AGGREGATE_TARGET = n** (where n is some amount of memory established by the DBA). However, the DBA can let the Oracle DBMS determine the appropriate amount of memory.

Automatic Shared Memory Management

Prior to Oracle 10G, a DBA had to manually specify SGA Component sizes through the initialization parameters, such as SHARED_POOL_SIZE, DB_CACHE_SIZE, JAVA_POOL_SIZE, and LARGE_POOL_SIZE parameters.

Automatic Shared Memory Management enables a DBA to specify the total SGA memory available through the **SGA_TARGET** initialization parameter. The Oracle Database automatically distributes this memory among various subcomponents to ensure most effective memory utilization.

The **DBORCL** database **SGA_TARGET** is set in the **initDBORCL.ora** file:

sga_target=1610612736

With automatic SGA memory management, the different SGA components are flexibly sized to adapt to the SGA available.

Setting a single parameter simplifies the administration task – the DBA only specifies the amount of SGA memory available to an instance – the DBA can forget about the sizes of individual components. No out of memory errors are generated unless the system has actually run out of memory. No manual tuning effort is needed.

The **SGA_TARGET** initialization parameter reflects the total size of the SGA and includes memory for the following components:

- Fixed SGA and other internal allocations needed by the Oracle Database instance
- The log buffer
- The shared pool
- The Java pool
- The buffer cache
- The keep and recycle buffer caches (if specified)
- Nonstandard block size buffer caches (if specified)
- The Streams Pool

If **SGA_TARGET** is set to a value greater than **SGA_MAX_SIZE** at startup, then the SGA_MAX_SIZE value is bumped up to accommodate SGA_TARGET.

When you set a value for **SGA_TARGET**, Oracle Database 11*g* automatically sizes the most commonly configured components, including:

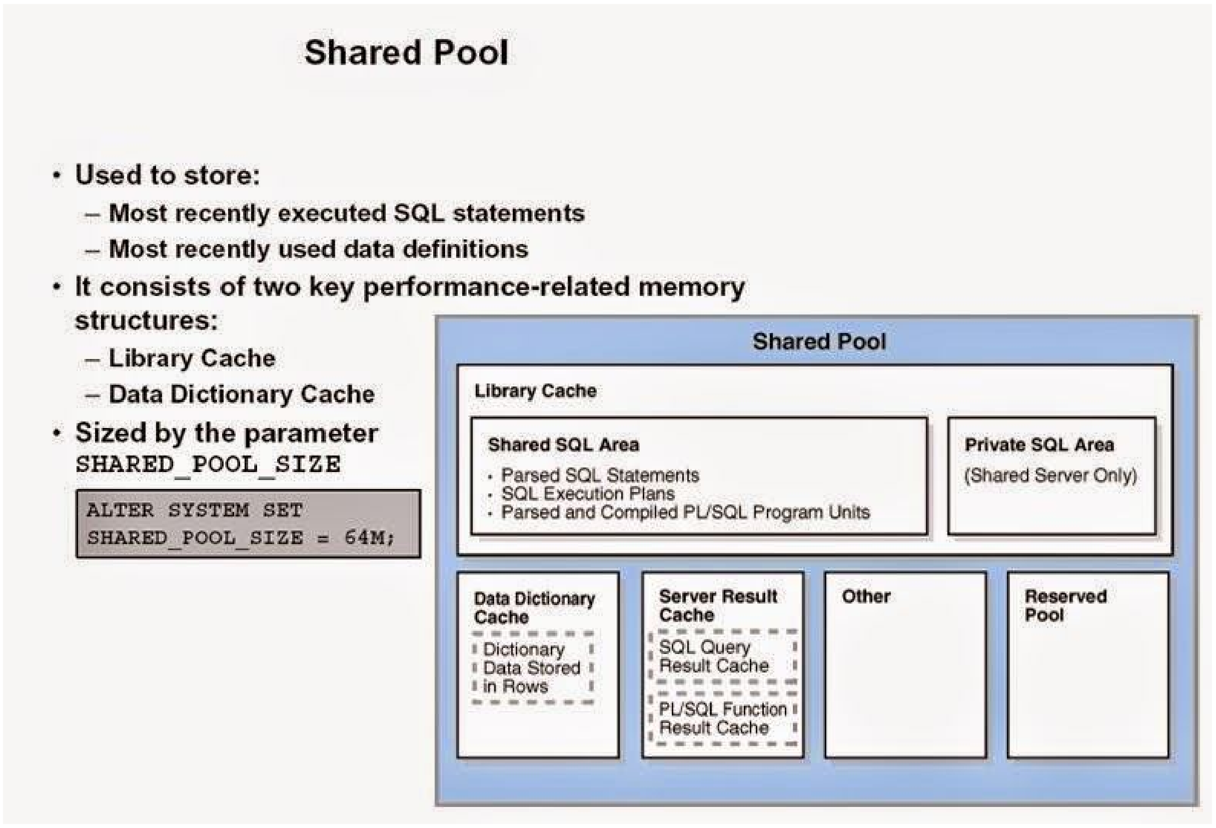
- The shared pool (for SQL and PL/SQL execution)
- The Java pool (for Java execution state)
- The large pool (for large allocations such as RMAN backup buffers)
- The buffer cache

There are a few SGA components whose sizes are not automatically adjusted. The DBA must specify the sizes of these components explicitly, if they are needed by an application. Such components are:

- Keep/Recycle buffer caches (controlled by **DB_KEEP_CACHE_SIZE** and **DB_RECYCLE_CACHE_SIZE**)
- Additional buffer caches for non-standard block sizes (controlled by **DB_*n*K_CACHE_SIZE**, *n* = {2, 4, 8, 16, 32})
- Streams Pool (controlled by the new parameter **STREAMS_POOL_SIZE**)

The granule size that is currently being used for the SGA for each component can be viewed in the view **V$SGAINFO**. The size of each component and the time and type of the last resize operation performed on each component can be viewed in the view **V$SGA_DYNAMIC_COMPONENTS**.

Shared-Pool



Shared Pool

- Used to store:
  - Most recently executed SQL statements
  - Most recently used data definitions
- It consists of two key performance-related memory structures:
  - Library Cache
  - Data Dictionary Cache
- Sized by the parameter `SHARED_POOL_SIZE`

```
ALTER SYSTEM SET
SHARED_POOL_SIZE = 64M;
```

The **Shared Pool** is a memory structure that is shared by all system users.

· It caches various types of program data. For example, the shared pool stores parsed SQL, PL/SQL code, system parameters, and data dictionary information.

· The shared pool is involved in almost every operation that occurs in the database. For example, if a user executes a SQL statement, then Oracle Database accesses the shared pool.

· It consists of both fixed and variable structures.

· The variable component grows and shrinks depending on the demands placed on memory size by system users and application programs.

Memory can be allocated to the Shared Pool by the parameter **SHARED_POOL_SIZE** in the parameter file. The default value of this parameter is **8MB** on 32-bit platforms and **64MB** on 64-bit platforms. Increasing the value of this parameter increases the amount of memory reserved for the shared pool.

You can alter the size of the shared pool dynamically with the **ALTER SYSTEM SET** command. An example command is shown in the figure below. You must keep in mind that the total memory allocated to the SGA is set by the **SGA_TARGET** parameter (and may also be limited by the **SGA_MAX_SIZE** if it is set), and since the Shared Pool is part of the SGA, you cannot exceed the maximum size of the SGA. It is recommended to let Oracle optimize the Shared Pool size.

The Shared Pool stores the most recently executed SQL statements and used data definitions. This is because some system users and application programs will tend to execute the same SQL statements often. Saving this information in memory can improve system performance.

The Shared Pool includes several cache areas described below.

Library Cache

Memory is allocated to the **Library Cache** whenever an SQL statement is parsed or a program unit is called. This enables storage of the most recently used SQL and PL/SQL statements.

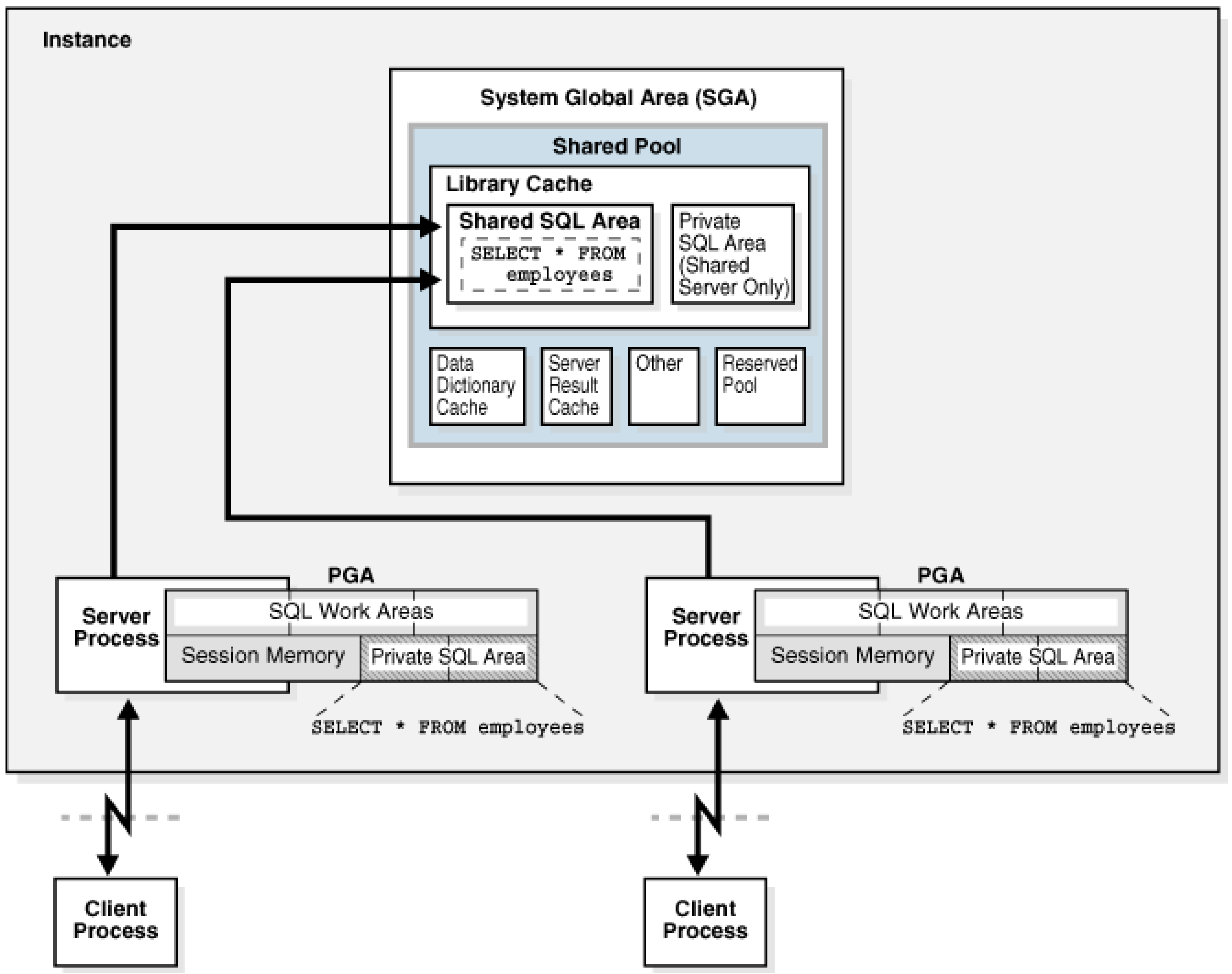
If the Library Cache is too small, the Library Cache must purge statement definitions in order to have space to load new SQL and PL/SQL statements. Actual management of this memory structure is through a **Least-Recently-Used (LRU) algorithm**. This means that the SQL and PL/SQL statements that are oldest and least recently used are purged when more storage space is needed.

The Library Cache is composed of two memory subcomponents:

· **Shared SQL**: This stores/shares the execution plan and parse tree for SQL statements, as well as PL/SQL statements such as functions, packages, and triggers. If a system user executes an identical statement, then the statement does not have to be parsed again in order to execute the statement.

- **Private SQL Area:** With a shared server, each session issuing a SQL statement has a private SQL area in its PGA.

  - Each user that submits the same statement has a private SQL area pointing to the same shared SQL area.
  - Many private SQL areas in separate PGAs can be associated with the same shared SQL area.
  - This figure depicts two different client processes issuing the same SQL statement – the parsed solution is already in the Shared SQL Area.



-

Data Dictionary Cache

The Data Dictionary Cache is a memory structure that caches data dictionary information that has been recently used.

· This cache is necessary because the data dictionary is accessed so often.

· Information accessed includes user account information, datafile names, table descriptions, user privileges, and other information.

The database server manages the size of the Data Dictionary Cache internally and the size depends on the size of the Shared Pool in which the Data Dictionary Cache resides. If the size is too small, then the data dictionary tables that reside on disk must be queried often for information and this will slow down performance.

Server Result Cache

The Server Result Cache holds result sets and not data blocks. The server result cache contains the SQL query result cache and PL/SQL function result cache, which share the same infrastructure.

SQL Query Result Cache

This cache stores the results of queries and query fragments.

· Using the cache results for future queries tends to improve performance.

· For example, suppose an application runs the same SELECT statement repeatedly. If the results are cached, then the database returns them immediately.

· In this way, the database avoids the expensive operation of rereading blocks and recomputing results.

PL/SQL Function Result Cache

The PL/SQL Function Result Cache stores function result sets.

· Without caching, 1000 calls of a function at 1 second per call would take 1000 seconds.

- With caching, 1000 function calls with the same inputs could take 1 second total.
- Good candidates for result caching are frequently invoked functions that depend on relatively static data.
- PL/SQL function code can specify that results be cached.

Buffer Caches

A number of buffer caches are maintained in memory in order to improve system response time.

Database Buffer Cache

The **Database Buffer Cache** is a fairly large memory object that stores the actual data blocks that are retrieved from datafiles by system queries and other data manipulation language commands.

The purpose is to optimize physical input/output of data.

When **Database Smart Flash Cache (flash cache)** is enabled, part of the buffer cache can reside in the flash cache.

- This buffer cache extension is stored on a **flash disk device**, which is a solid state storage device that uses flash memory.
- The database can improve performance by caching buffers in flash memory instead of reading from magnetic disk.
- Database Smart Flash Cache is available only in Solaris and Oracle Enterprise Linux.

A query causes a **Server Process** to look for data.

- The first look is in the Database Buffer Cache to determine if the requested information happens to already be located in memory – thus the information would not need to be retrieved from disk and this would speed up performance.

- If the information is not in the Database Buffer Cache, the Server Process retrieves the information from disk and stores it to the cache.
- Keep in mind that information read from disk is read a **block at a time**, **NOT** a **row at a time**, because a database block is the smallest addressable storage space on disk.

Database blocks are kept in the Database Buffer Cache according to a **Least Recently Used (LRU) algorithm** and are aged out of memory if a buffer cache block is not used in order to provide space for the insertion of newly needed database blocks.

There are three buffer states:

- **Unused** - a buffer is available for use - it has never been used or is currently unused.
- **Clean** - a buffer that was used earlier - the data has been written to disk.
- **Dirty** - a buffer that has modified data that has not been written to disk.

Each buffer has one of two access modes:

- **Pinned** - a buffer is pinned so it does not age out of memory.
- **Free** (unpinned).

The buffers in the cache are organized in two lists:

- the write list and,
- the least recently used (LRU) list.

The **write list** (also called a **write queue**) holds dirty buffers – these are buffers that hold that data that has been modified, but the blocks have not been written back to disk.

The **LRU list** holds unused, free clean buffers, pinned buffers, and free dirty buffers that have not yet been moved to the write list. **Free clean buffers** do not contain any useful data and are available for use. **Pinned buffers** are currently being accessed.

When an Oracle process accesses a buffer, the process moves the buffer to the **most recently used (MRU)** end of the LRU list – this causes dirty buffers to age toward the LRU end of the LRU list.

When an Oracle user process needs a data row, it searches for the data in the database buffer cache because memory can be searched more quickly than hard disk can be accessed. If the data row is already in the cache (a **cache hit**), the process reads the data from memory; otherwise a **cache miss** occurs and data must be read from hard disk into the database buffer cache.

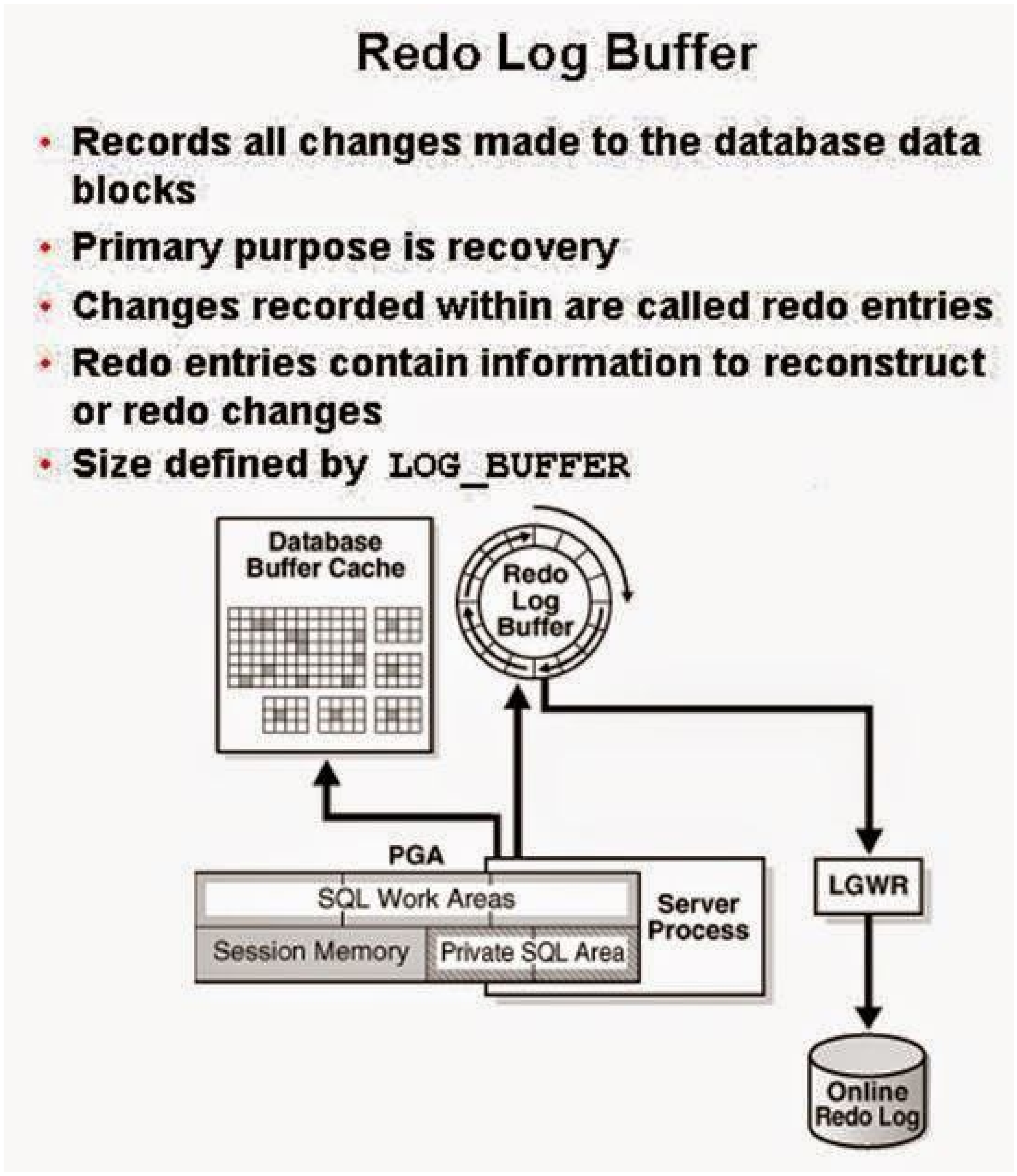
Before reading a data block into the cache, the process must first find a free buffer. The process searches the LRU list, starting at the LRU end of the list. The search continues until a free buffer is found or until the search reaches the threshold limit of buffers.

Each time a user process finds a dirty buffer as it searches the LRU, that buffer is moved to the write list and the search for a free buffer continues.

When a user process finds a free buffer, it reads the data block from disk into the buffer and moves the buffer to the MRU end of the LRU list.

If an Oracle user process searches the threshold limit of buffers without finding a free buffer, the process stops searching the LRU list and signals the DBWn background process to write some of the dirty buffers to disk. This frees up some buffers.

Redo Log Buffer



Redo Log Buffer

- Records all changes made to the database data blocks
- Primary purpose is recovery
- Changes recorded within are called redo entries
- Redo entries contain information to reconstruct or redo changes
- Size defined by `LOG_BUFFER`

Database Buffer Cache
Redo Log Buffer
PGA
SQL Work Areas
Session Memory
Private SQL Area
Server Process
LGWR
Online Redo Log

The **Redo Log Buffer** memory object stores images of all changes made to database blocks.

· Database blocks typically store several table rows of organizational data. This means that if a single column value from one row in a block is changed, the block image is stored. Changes include INSERT, UPDATE, DELETE, CREATE, ALTER, or DROP.

· LGWR writes redo sequentially to disk while DBWn performs scattered writes of data blocks to disk.

o Scattered writes tend to be much slower than sequential writes.

o Because LGWR enable users to avoid waiting for DBWn to complete its slow writes, the database delivers better performance.

The Redo Log Buffer as a circular buffer that is reused over and over. As the buffer fills up, copies of the images are stored to the **Redo Log Files** that are covered in more detail in a later module.

Large Pool

The **Large Pool** is an optional memory structure that primarily relieves the memory burden placed on the Shared Pool. The Large Pool is used for the following tasks if it is allocated:

· Allocating space for session memory requirements from the User Global Area where a Shared Server is in use.

· Transactions that interact with more than one database, e.g., a distributed database scenario.

· Backup and restore operations by the Recovery Manager (RMAN) process.

o RMAN uses this only if the **BACKUP_DISK_IO = n** and **BACKUP_TAPE_IO_SLAVE = TRUE** parameters are set.

o If the Large Pool is too small, memory allocation for backup will fail and memory will be allocated from the Shared Pool.

· Parallel execution message buffers for parallel server operations. The **PARALLEL_AUTOMATIC_TUNING = TRUE** parameter must be set.

The Large Pool size is set with the **LARGE_POOL_SIZE** parameter – this is not a dynamic parameter. It does not use an LRU list to manage memory.

Java Pool

The Java Pool is an **optional** memory object, but is required if the database has Oracle Java installed and in use for Oracle JVM (Java Virtual Machine).

· The size is set with the **JAVA_POOL_SIZE** parameter that defaults to 24MB.

· The Java Pool is used for memory allocation to parse Java commands and to store data associated with Java commands.

· Storing Java code and data in the Java Pool is analogous to SQL and PL/SQL code cached in the Shared Pool.

Streams Pool

This pool stores data and control structures to support the Oracle Streams feature of Oracle Enterprise Edition.

· Oracle Steams manages sharing of data and events in a distributed environment.

· It is sized with the parameter **STREAMS_POOL_SIZE**.

  · If STEAMS_POOL_SIZE is not set or is zero, the size of the pool grows dynamically.

Processes

You need to understand three different types of Processes:

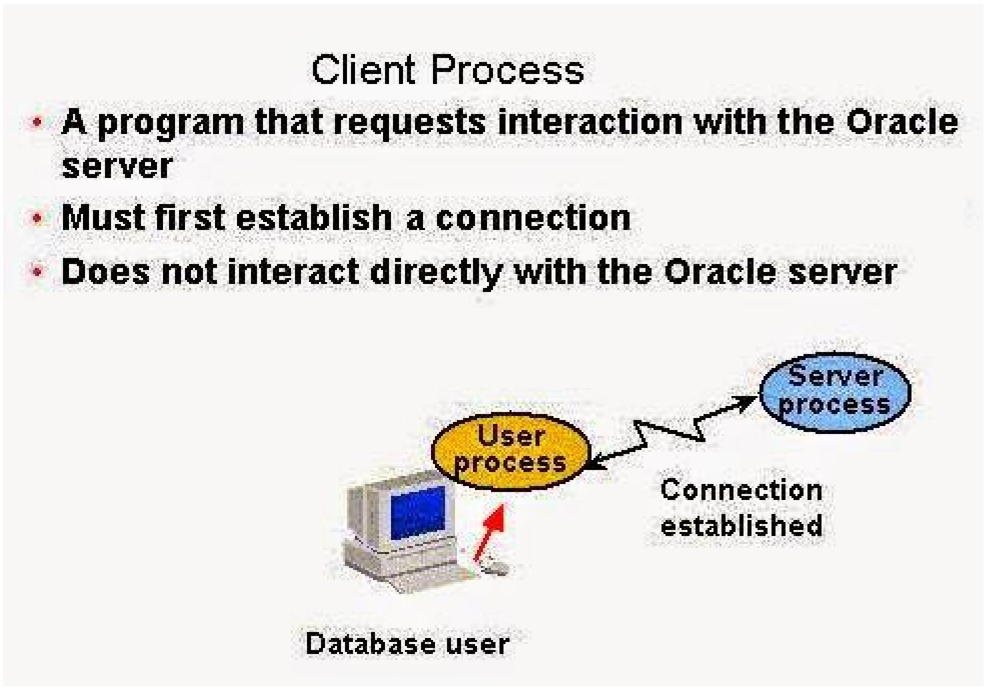
· **User Process**: Starts when a database user requests to connect to an Oracle Server.

· **Server Process**: Establishes the Connection to an Oracle Instance when a User Process requests connection – makes the connection for the User Process.

· **Background Processes**: These start when an Oracle Instance is started up.

Client Process

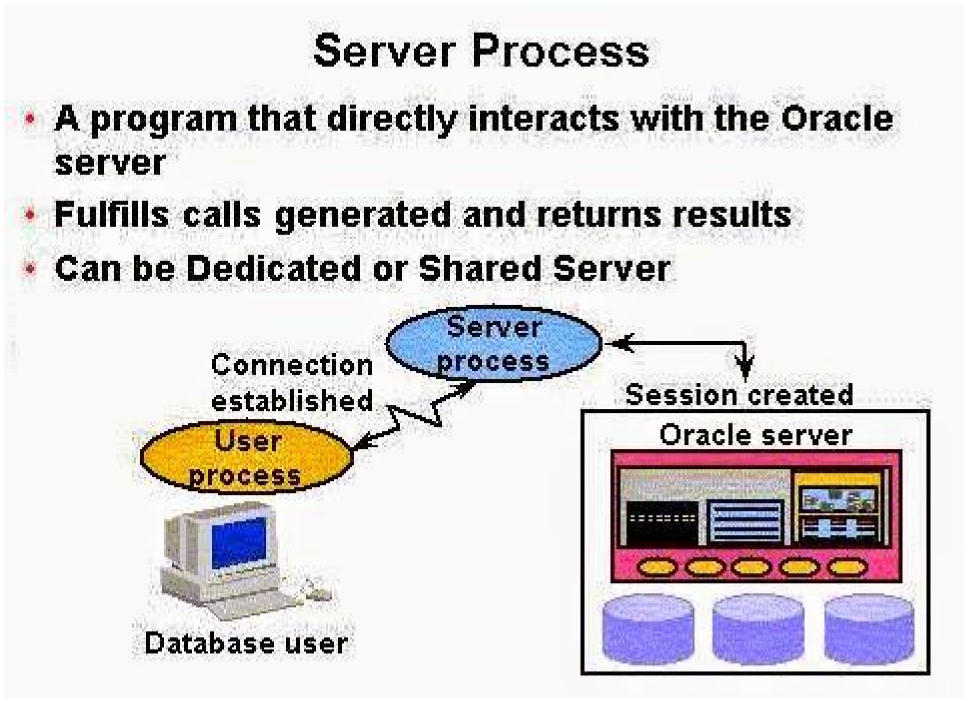
In order to use Oracle, you must connect to the database. This must occur whether you're using SQLPlus, an Oracle tool such as Designer or Forms, or an application program. The client process is also termed the user process in some Oracle documentation.



Client Process

- A program that requests interaction with the Oracle server
- Must first establish a connection
- Does not interact directly with the Oracle server

Server Process



Server Process

- A program that directly interacts with the Oracle server
- Fulfills calls generated and returns results
- Can be Dedicated or Shared Server

A Server Process is the go-between for a Client Process and the Oracle Instance.

· Dedicated Server environment – there is a single Server Process to serve each Client Process.

· Shared Server environment – a Server Process can serve several User Processes, although with some performance reduction.

· Allocation of server process in a dedicated environment versus a shared environment is covered in further detail in the *Oracle11g Database Performance Tuning* course offered by Oracle Education.

Background Processes

As is shown here, there are both mandatory, optional, and slave background processes that are started whenever an Oracle Instance starts up. These background processes serve all system users. We will cover mandatory process in detail.

Mandatory Background Processes

- **Process Monitor Process (PMON)**
- **System Monitor Process (SMON)**
- **Database Writer Process (DBWn)**
- **Log Writer Process (LGWR)**
- **Checkpoint Process (CKPT)**
- **Manageability Monitor Processes (MMON and MMNL)**
- **Recover Process (RECO)**

Optional Processes

- **Archiver Process (ARCn)**
- **Coordinator Job Queue (CJQ0)**
- **Dispatcher (number “nnn”) (Dnnn)**
- **Others**

This query will display all background processes running to serve a database:

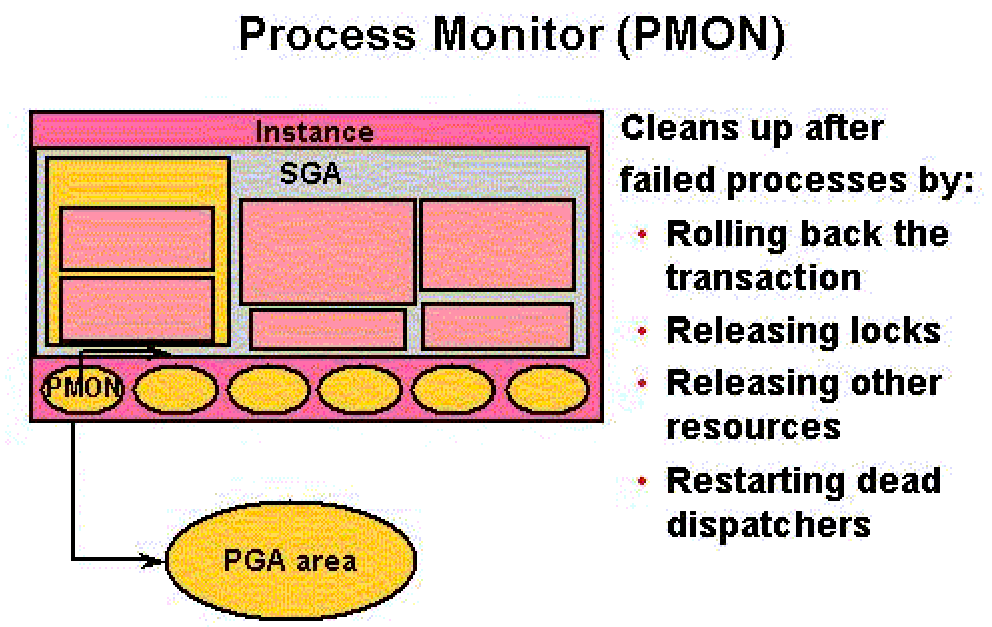
```
SELECT PNAME
FROM   V$PROCESS
WHERE  PNAME IS NOT NULL
ORDER BY PNAME;
```

PMON

The **Process Monitor (PMON)** monitors other background processes.

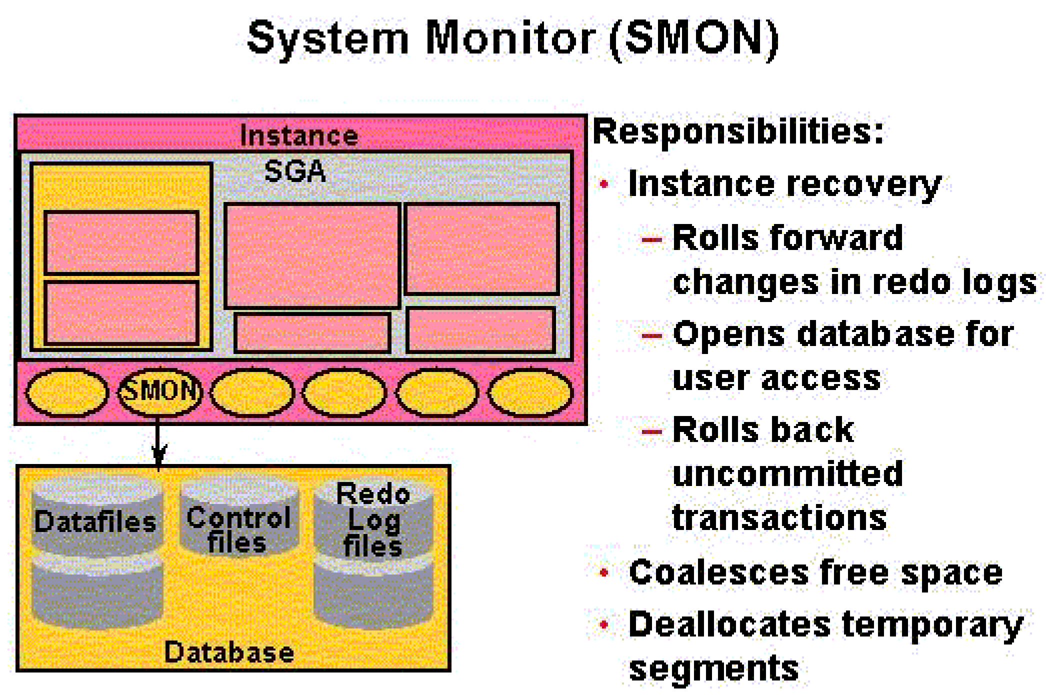
- It is a cleanup type of process that cleans up after failed processes.

- Examples include the dropping of a user connection due to a network failure or the abnormal termination (ABEND) of a user application program.
- It cleans up the database buffer cache and releases resources that were used by a failed user process.
- It does the tasks shown in the figure below.


Process Monitor (PMON)
Instance
SGA
PMON
PGA area
Cleans up after failed processes by:
• Rolling back the transaction
• Releasing locks
• Releasing other resources
• Restarting dead dispatchers

SMON

- The **System Monitor (SMON)** does system-level cleanup duties.
- It is responsible for instance recovery by applying entries in the online redo log files to the datafiles. Other processes can call SMON when it is needed.
- It also performs other activities as outlined in the figure shown below.



System Monitor (SMON)

Responsibilities:

- **Instance recovery**
  - **Rolls forward changes in redo logs**
  - **Opens database for user access**
  - **Rolls back uncommitted transactions**
- **Coalesces free space**
- **Deallocates temporary segments**

If an Oracle Instance fails, all information in memory not written to disk is lost. SMON is responsible for recovering the instance when the database is started up again. It does the following:

- Rolls forward to recover data that was recorded in a Redo Log File, but that had not yet been recorded to a datafile by DBWn. SMON reads the Redo Log Files and applies the changes to the data blocks. This recovers all transactions that were committed because these were written to the Redo Log Files prior to system failure.
- Opens the database to allow system users to logon.
- Rolls back uncommitted transactions.

SMON also does limited space management. It combines (coalesces) adjacent areas of free space in the database's datafiles for tablespaces that are dictionary managed.

It also deallocates temporary segments to create free space in the datafiles.

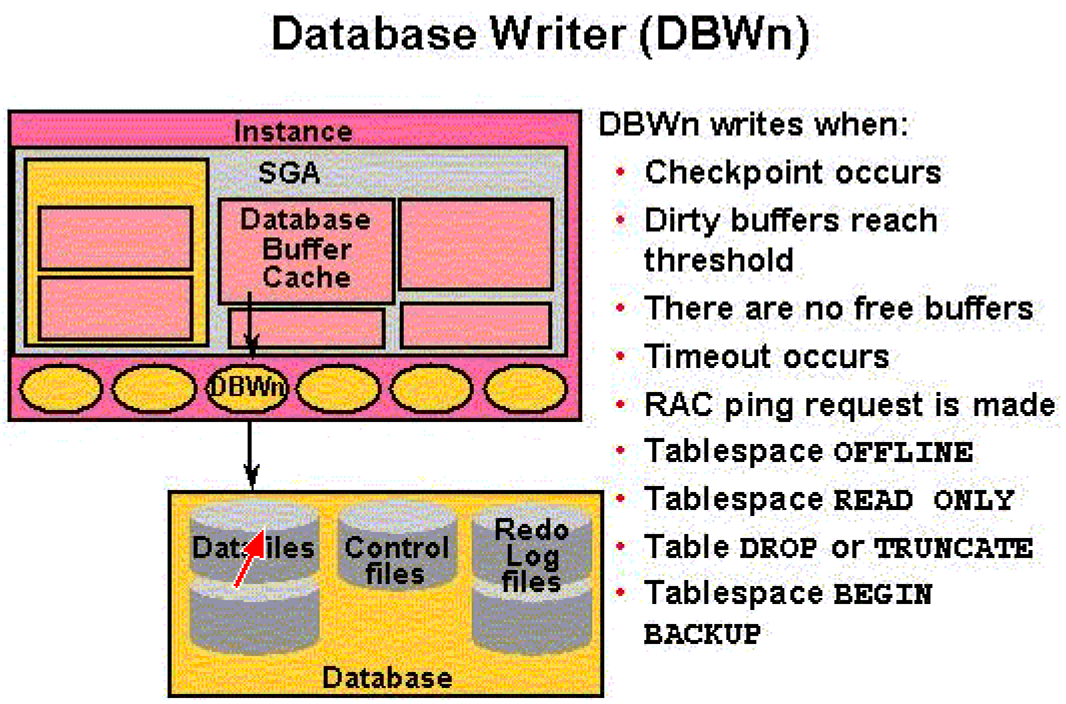
DBWn (also called DBWR in earlier Oracle Versions)

The **Database Writer** writes modified blocks from the database buffer cache to the datafiles.

- One database writer process (DBW0) is sufficient for most systems.
- A DBA can configure up to 20 DBWn processes (DBW0 through DBW9 and DBWa through DBWj) in order to improve write performance for a system that modifies data heavily.
- The initialization parameter **DB_WRITER_PROCESSES** specifies the number of DBW*n* processes.

The purpose of **DBWn** is to improve system performance by caching writes of database blocks from the **Database Buffer Cache** back to datafiles.

- Blocks that have been modified and that need to be written back to disk are termed "**dirty blocks**."
- The DBWn also ensures that there are enough free buffers in the Database Buffer Cache to service Server Processes that may be reading data from datafiles into the Database Buffer Cache
- Performance improves because by delaying writing changed database blocks back to disk, a Server Process may find the data that is needed to meet a User Process request already residing in memory!
- DBWn writes to datafiles when one of these events occurs that is illustrated in the figure below.

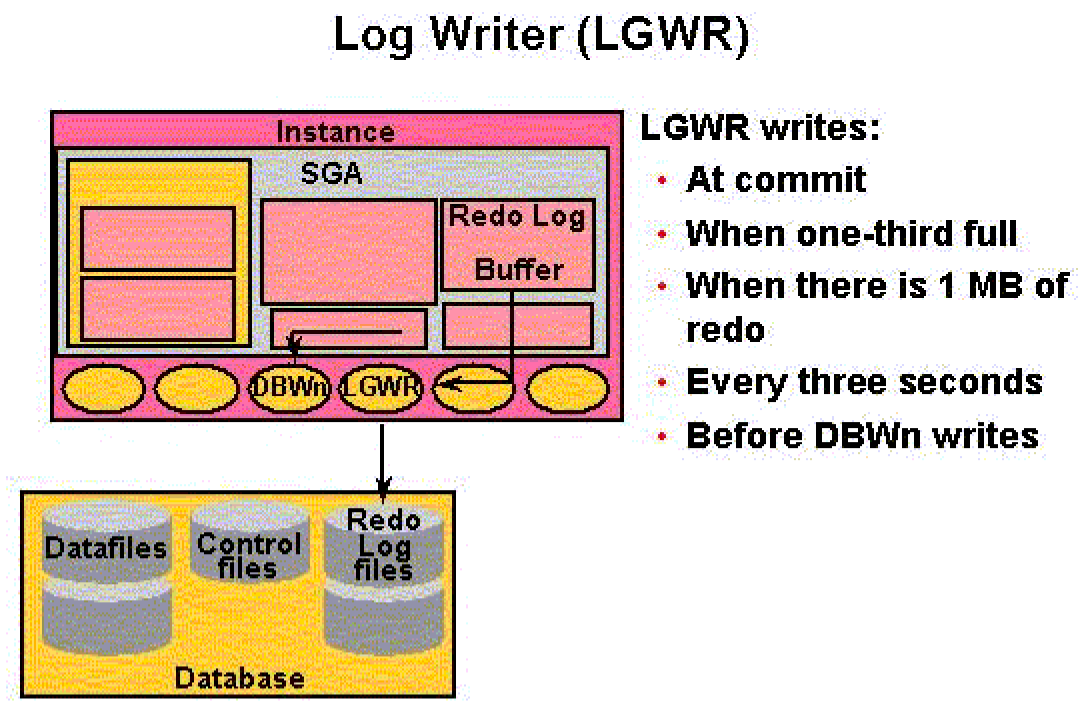


LGWR

The **Log Writer (LGWR)** writes contents from the Redo Log Buffer to the Redo Log File that is in use.

- These are sequential writes since the Redo Log Files record database modifications based on the actual time that the modification takes place.

- LGWR actually writes before the DBWn writes and only confirms that a COMMIT operation has succeeded when the Redo Log Buffer contents are successfully written to disk.
- LGWR can also call the DBWn to write contents of the Database Buffer Cache to disk.
- The LGWR writes according to the events illustrated in the figure shown below.



Log Writer (LGWR)

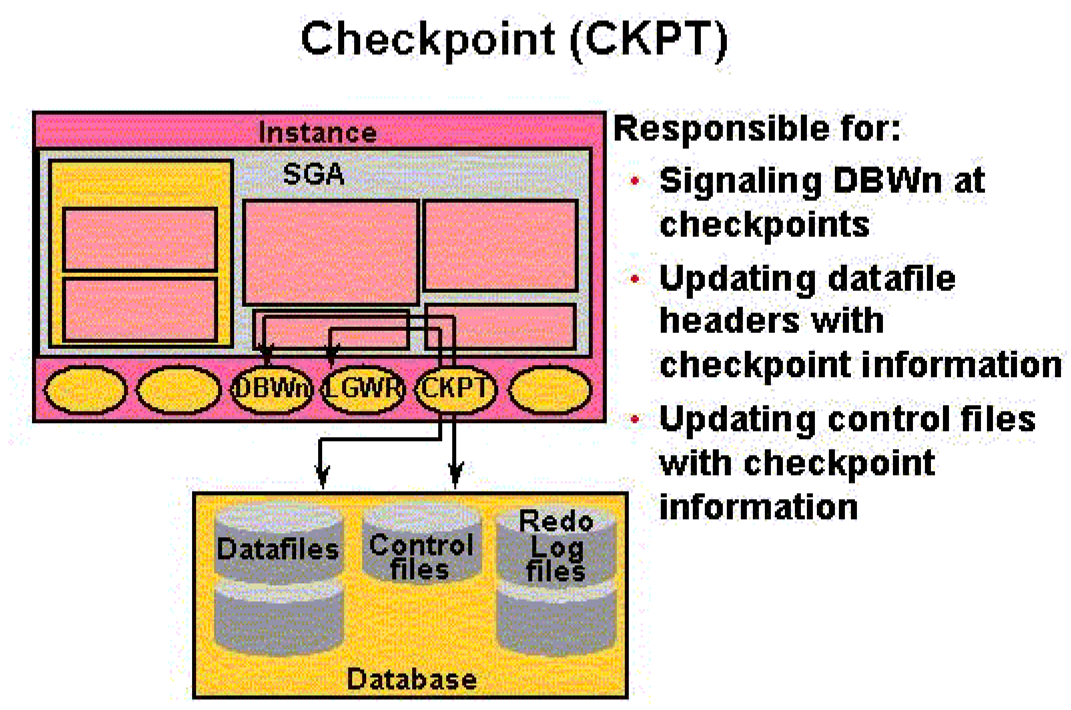
LGWR writes:

- At commit
- When one-third full
- When there is 1 MB of redo
- Every three seconds
- Before DBWn writes

CKPT

The **Checkpoint (CPT)** process writes information to update the database control files and headers of datafiles.

- A checkpoint identifies a point in time with regard to the **Redo Log Files** where instance recovery is to begin should it be necessary.
- It can tell DBWn to write blocks to disk.
- A checkpoint is taken at a minimum, once every <u>**three seconds**</u>.



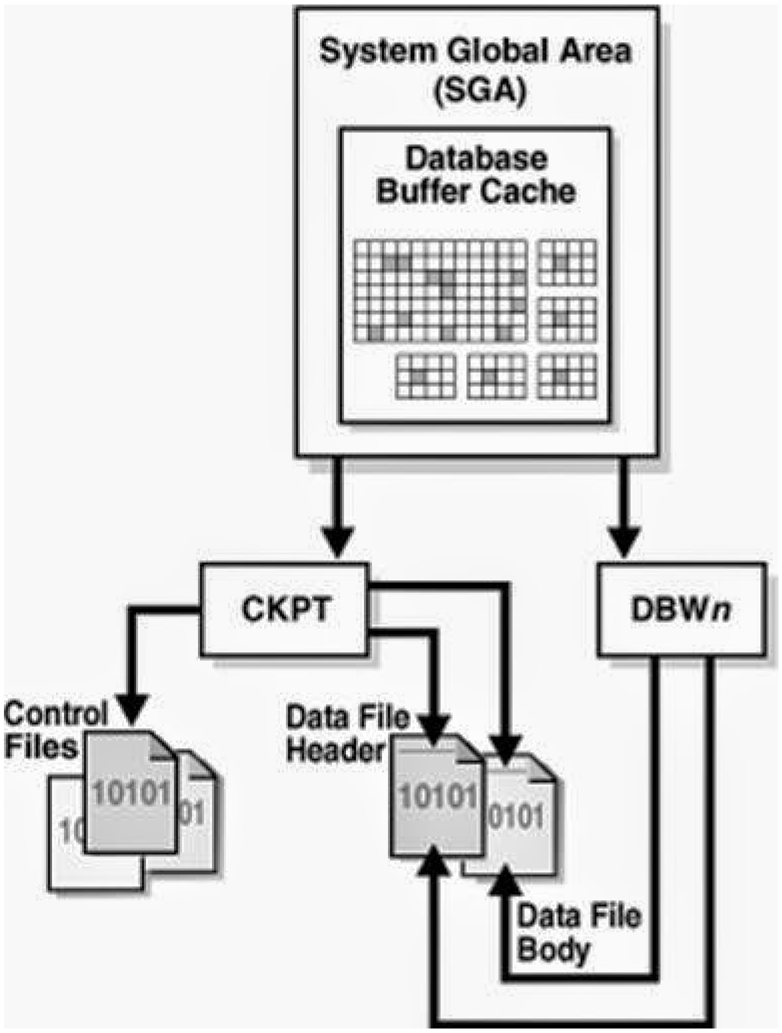
Checkpoint (CKPT)

Responsible for:

- **Signaling DBWn at checkpoints**
- **Updating datafile headers with checkpoint information**
- **Updating control files with checkpoint information**

Think of a checkpoint record as a starting point for recovery. DBWn will have completed writing all buffers from the Database Buffer Cache to disk prior to the checkpoint, thus those records will not require recovery. This does the following:

- Ensures modified data blocks in memory are regularly written to disk – CKPT can call the DBWn process in order to ensure this and does so when writing a checkpoint record.
- Reduces Instance Recovery time by minimizing the amount of work needed for recovery since only Redo Log File entries processed since the last checkpoint require recovery.
- Causes all committed data to be written to datafiles during database shutdown.


System Global Area (SGA)
Database Buffer Cache
CKPT
DBWn
Control Files
Data File Header
Data File Body
10101

Unit – IVth

SQL

• SQL stands for Structured Query Language

• SQL lets you access and manipulate databases

• SQL is an ANSI (American National Standards Institute) standard

SQL is a declarative (non-procedural)language. SQL is (usually) not case-sensitive,

but we'll write SQL keywords in upper case for emphasis.

Some database systems require a semicolon at the end of each SQL statement.

A table is database object that holds user data. Each column of the table will have specified data type bound to it. Oracle ensures that only data, which is identical to the datatype of the column, will be stored within the column.

SQL DML and DDL

SQL can be divided into two parts:

The Data Definition Language (DDL) and the Data Manipulation Language (DML).

Data Definition Language (DDL)

It is a set of SQL commands used to create, modify and delete database structure

but not data. It also define indexes (keys), specify links between tables, and

impose constraints between tables. DDL commands are auto COMMIT.

The most important DDL statements in SQL are:

• CREATE TABLE - creates a new table

• ALTER TABLE - modifies a table

TRUNCATE TABLE- deletes all records from a table

DROP TABLE - deletes a table

Data Manipulation Language (DML)

It is the area of SQL that allows changing data within the database. The query and update commands form the DML part of SQL:

• INSERT - inserts new data into a database

• SELECT - extracts data from a database

• UPDATE - updates data in a database

• DELETE - deletes data from a database

Data Control Language (DCL)

It is the component of SQL statement that control access to data and to the database. Occasionally DCL statements are grouped with DML Statements.

COMMIT –Save work done.

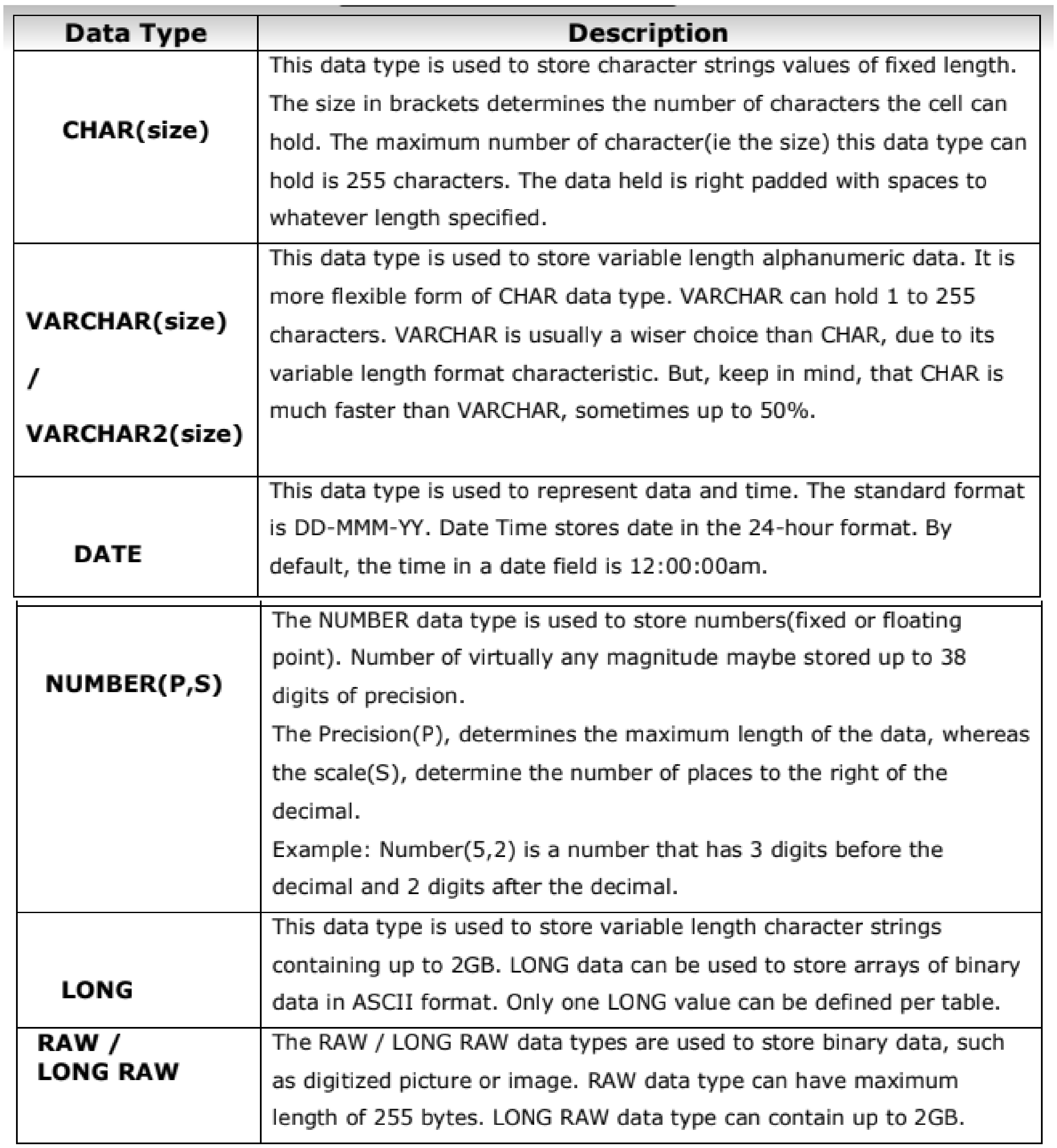
SAVEPOINT – Identify a point in a transaction to which you can later rollback.

ROLLBACK – Restore database to original since the last COMMIT.

GRANT – gives user's access privileges to database.

REVOKE – withdraw access privileges given with GRANT command.

Basic Data Types



Data Type	Description
CHAR(size)	This data type is used to store character strings values of fixed length. The size in brackets determines the number of characters the cell can hold. The maximum number of character(ie the size) this data type can hold is 255 characters. The data held is right padded with spaces to whatever length specified.
VARCHAR(size) / VARCHAR2(size)	This data type is used to store variable length alphanumeric data. It is more flexible form of CHAR data type. VARCHAR can hold 1 to 255 characters. VARCHAR is usually a wiser choice than CHAR, due to its variable length format characteristic. But, keep in mind, that CHAR is much faster than VARCHAR, sometimes up to 50%.
DATE	This data type is used to represent data and time. The standard format is DD-MMM-YY. Date Time stores date in the 24-hour format. By default, the time in a date field is 12:00:00am.
NUMBER(P,S)	The NUMBER data type is used to store numbers(fixed or floating point). Number of virtually any magnitude maybe stored up to 38 digits of precision. The Precision(P), determines the maximum length of the data, whereas the scale(S), determine the number of places to the right of the decimal. Example: Number(5,2) is a number that has 3 digits before the decimal and 2 digits after the decimal.
LONG	This data type is used to store variable length character strings containing up to 2GB. LONG data can be used to store arrays of binary data in ASCII format. Only one LONG value can be defined per table.
RAW / LONG RAW	The RAW / LONG RAW data types are used to store binary data, such as digitized picture or image. RAW data type can have maximum length of 255 bytes. LONG RAW data type can contain up to 2GB.

The CREATE TABLE Command:

The CREATE TABLE command defines each column of the table uniquely. Each column has a minimum of three attributes, name, datatype and size(i.e column width).each table column definition is a single clause in the create table syntax. Each table column definition is separated from the other by a comma. Finally, the

SQL statement is terminated with a semi colon.

Rules for Creating Tables

- A name can have maximum upto 30 characters.
- Alphabets from A-Z, a-z and numbers from 0-9 are allowed.
- A name should begin with an alphabet.
- The use of the special character like _(underscore) is allowed.
- SQL reserved words not allowed. For example: create, select, alter.

Syntax:

```
CREATE TABLE <tablename>

(<columnName1> <Datatype>(<size>),

<columnName2> <Datatype>(<size>), ....... );
```

Example:

```
CREATE TABLE gktab

(Regno NUMBER(3),

Name VARCHAR(20),

Gender CHAR,

Dob DATE,

Course CHAR(5));
```

Inserting Data into Tables

Once a table is created, the most natural thing to do is load this table with data to

be manipulated later.

When inserting a single row of data into the table, the insert operation:

- Creates a new row(empty) in the database table.

- Loads the values passed(by the SQL insert) into the columns specified.

Syntax:

INSERT INTO <tablename>(<columnname1>, <columnname2>, ..)

Values(<expression1>,<expression2>...);

Example:

INSERT INTO gktab(regno,name,gender,dob,course)

VALUES(101,’Varsh G Kalyan’,’F’,’20-Sep-1985’,’BCA’);

Or you can use the below method to insert the data into table.

INSERT INTO gktab VALUES(102,’Mohith G Kalyan’,’M’,’20-Aug-1980’,’BBM’);

INSERT INTO gktab VALUES(106,’Nisarga’,’F’,’15-Jul-1983’,’BCom’);

INSERT INTO gktab VALUES(105,’Eenchara’,’F’,’04-Dec-1985’,’BCA’);

INSERT INTO gktab VALUES(103,’Ravi K’,’M’,’29-Mar-1989’,’BCom’);

INSERT INTO gktab VALUES(104,’Roopa’,’F’,’17-Jan-1984’,’BBM’);

Whenever you work on the data which has data types like

CHAR,VARCHAR/VARCHAR2, DATE should be used between single quote(‘)

Viewing Data in the Tables

Once data has been inserted into a table, the next most logical operation would be

to view what has been inserted. The SELECT SQL verb is used to achieve this. The

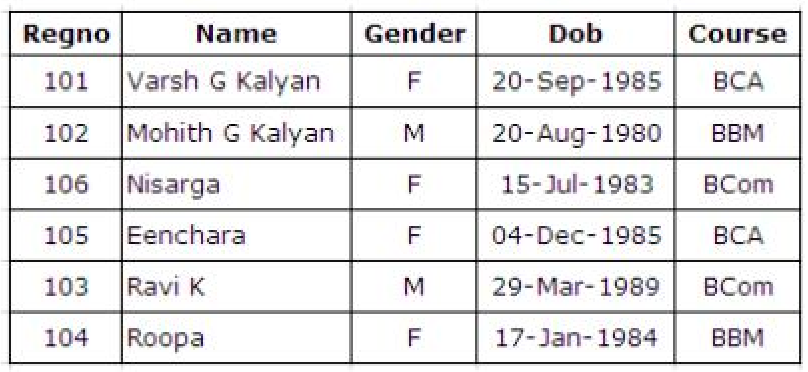
SELECT command is used to retrieve rows selected from one or more tables.

All Rows and All Columns

SELECT * FROM <tablename>

SELECT * FROM gktab;

It shows all rows and column data in the table



Regno	Name	Gender	Dob	Course
101	Varsh G Kalyan	F	20-Sep-1985	BCA
102	Mohith G Kalyan	M	20-Aug-1980	BBM
106	Nisarga	F	15-Jul-1983	BCom
105	Eenchara	F	04-Dec-1985	BCA
103	Ravi K	M	29-Mar-1989	BCom
104	Roopa	F	17-Jan-1984	BBM

Filtering Table Data

While viewing data from a table it is rare that all the data from the table will be required each time. Hence, SQL provides a method of filtering table data that is not required.

The ways of filtering table data are:

- Selected columns and all rows
- Selected rows and all columns
- Selected columns and selected rows

Selected Columns and All Rows

The retrieval of specific columns from a table can be done as shown below.

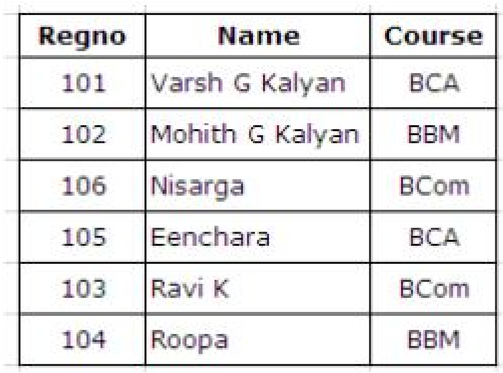
Syntax

SELECT <columnname1>, <Columnname2> FROM <tablename>

Example

Show only Regno, Name and Course from gktab.

SELECT Regno, Name, Course FROM gktab;



Regno	Name	Course
101	Varsh G Kalyan	BCA
102	Mohith G Kalyan	BBM
106	Nisarga	BCom
105	Eenchara	BCA
103	Ravi K	BCom
104	Roopa	BBM

Selected Rows and All Columns

The WHERE clause is used to extract only those records that fulfill a specified criterion.

When a WHERE clause is added to the SQL query, the Oracle engine compares each record in the table with condition specified in the WHERE clause. The Oracle engine displays only those records that satisfy the specified condition.

Syntax

SELECT * FROM <tablename> WHERE <condition>;

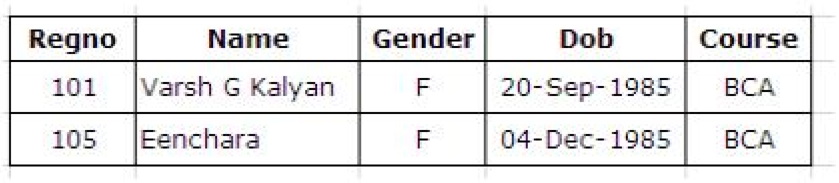
Here, <condition> is always quantified as <columnname=value>

When specifying a condition in the WHERE clause all standard operators such as logical, arithmetic and so on, can be used.

Example-1:

Display all the students from BCA.

SELECT * FROM gktab WHERE Course='BCA';

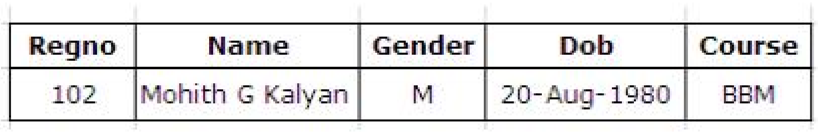


Regno	Name	Gender	Dob	Course
101	Varsh G Kalyan	F	20-Sep-1985	BCA
105	Eenchara	F	04-Dec-1985	BCA

Example-2:

Display the student whose regno is 102.

SELECT * FROM gktab WHERE Regno=102;



Regno	Name	Gender	Dob	Course
102	Mohith G Kalyan	M	20-Aug-1980	BBM

Selected Columns and Selected Rows

To view a specific set of rows and columns from a table

When a WHERE clause is added to the SQL query, the Oracle engine compares

each record in the table with condition specified in the WHERE clause. The Oracle

engine displays only those records that satisfy the specified condition.

Syntax

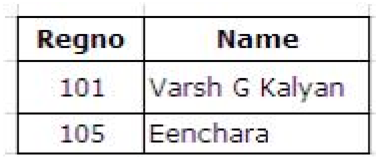
SELECT <columnname1>, <Columnname2> FROM <tablename>

WHERE <condition>;

Example-1:

List the student's Regno, Name for the Course BCA.

SELECT Regno, Name FROM gktab WHERE Course='BCA';

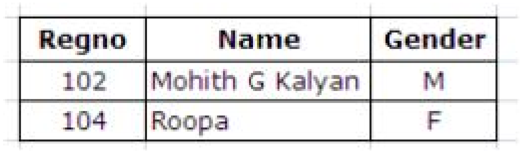


Regno	Name
101	Varsh G Kalyan
105	Eenchara

Example-2:

List the student's Regno, Name, Gender for the Course BBM.

SELECT Regno, Name, Gender FROM gktab WHERE Course='BBM';



Regno	Name	Gender
102	Mohith G Kalyan	M
104	Roopa	F

Eliminating Duplicate Rows when using a SELECT statement

A table could hold duplicate rows. In such a case, to view only unique rows the

DISTINCT clause can be used.

The DISTINCT clause allows removing duplicates from the result set. The

DISTINCT clause can only be used with SELECT statements.

The DISTINCT clause scans through the values of the column/s specified and

displays only unique values from amongst them.

Syntax

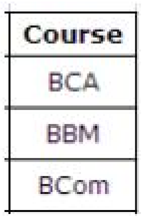
SELECT DISTINCT <columnname1>, <Columnname2>

FROM <Tablename>;

Example:

Show different courses from gktab

SELECT DISTINCT Course from gktab;



Course
BCA
BBM
BCom

Sorting Data in a Table

Oracle allows data from a table to be viewed in a sorted order. The rows retrieved from the table will be sorted in either ascending or descending order depending

on the condition specified in the SELECT sentence.

Syntax

SELECT * FROM <tablename>

ORDER BY <Columnname1>,<Columnname2> <[Sort Order]>;

The ORDER BY clause sorts the result set based on the column specified. The

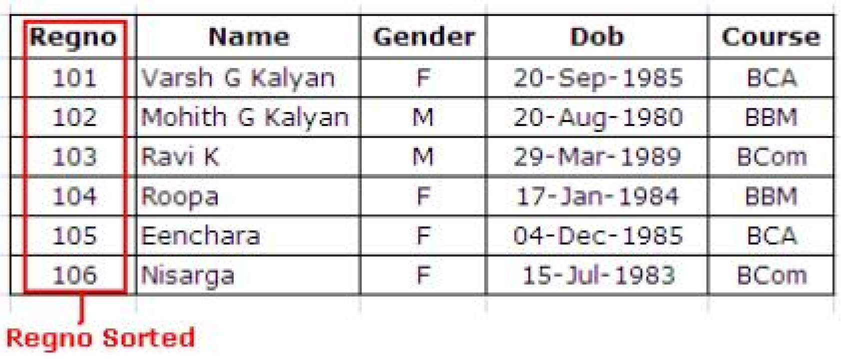
ORDER BY clause can only be used in SELECT statements.

The Oracle engine sorts in ascending order by default

Example-1:

Show details of students according to Regno.

SELECT * FROM gktab ORDER BY Regno;



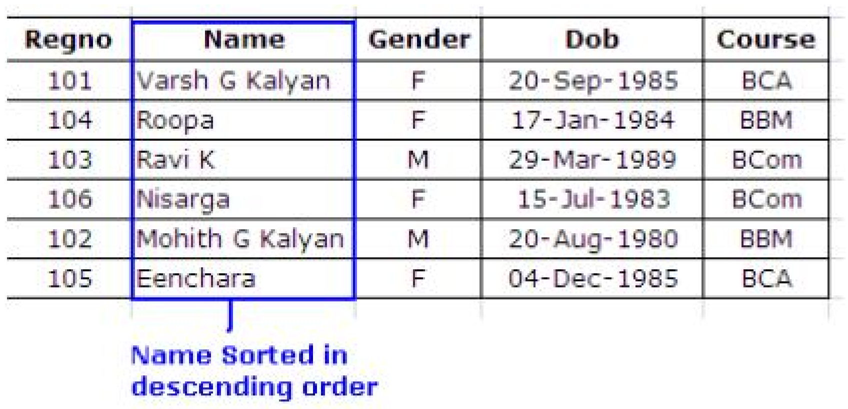
Regno	Name	Gender	Dob	Course
101	Varsh G Kalyan	F	20-Sep-1985	BCA
102	Mohith G Kalyan	M	20-Aug-1980	BBM
103	Ravi K	M	29-Mar-1989	BCom
104	Roopa	F	17-Jan-1984	BBM
105	Eenchara	F	04-Dec-1985	BCA
106	Nisarga	F	15-Jul-1983	BCom

Regno Sorted

Example-2:

Show the details of students names in descending order.

SELECT * FROM gktab ORDER BY Name DESC;



Regno	Name	Gender	Dob	Course
101	Varsh G Kalyan	F	20-Sep-1985	BCA
104	Roopa	F	17-Jan-1984	BBM
103	Ravi K	M	29-Mar-1989	BCom
106	Nisarga	F	15-Jul-1983	BCom
102	Mohith G Kalyan	M	20-Aug-1980	BBM
105	Eenchara	F	04-Dec-1985	BCA

Name Sorted in descending order

DELETE Operations

The DELETE command deletes rows from the table that satisfies the condition

provided by its WHERE clause, and returns the number of records deleted.

The verb DELETE in SQL is used to remove either

Specific row(s) from a table

OR

All the rows from a table

Removal of Specific Row(s)

Syntax:

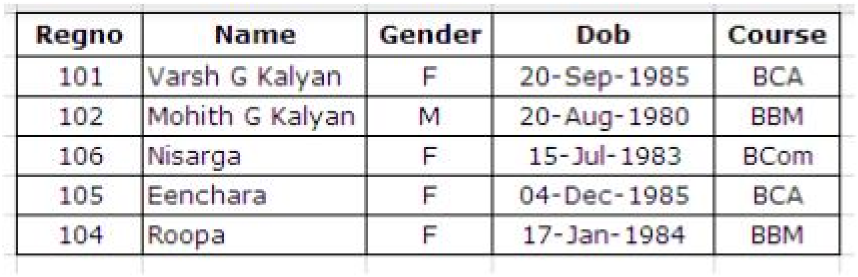
DELETE FROM tablename WHERE Condition;

Example:

DELETE FROM gktab WHERE Regno=103;

1 rows deleted

SELECT * FROM gktab;



Regno	Name	Gender	Dob	Course
101	Varsh G Kalyan	F	20-Sep-1985	BCA
102	Mohith G Kalyan	M	20-Aug-1980	BBM
106	Nisarga	F	15-Jul-1983	BCom
105	Eenchara	F	04-Dec-1985	BCA
104	Roopa	F	17-Jan-1984	BBM

In the above table, the Regno 103 is deleted from the table

Remove of ALL Rows

Syntax

DELETE FROM tablename;

Example

DELETE FROM gktab;

6 rows deleted

SELECT * FROM gktab;

no rows selected

Once the table is deleted, use Rollback to undo the above operations.

UPDATING THE CONTENTS OF A TABLE

The UPDATE Command is used to change or modify data values in a table. The verb update in SQL is used to either updates:

ALL the rows from a table.

OR

A select set of rows from a table.

Updating all rows

The UPDATE statement updates columns in the existing table's rows with a new values. The SET clause indicates which column data should be modified and the new values that they should hold. The WHERE clause, if given, specifies which rows should be updated. Otherwise, all table rows are updated.

Syntax:

UPDATE tablename

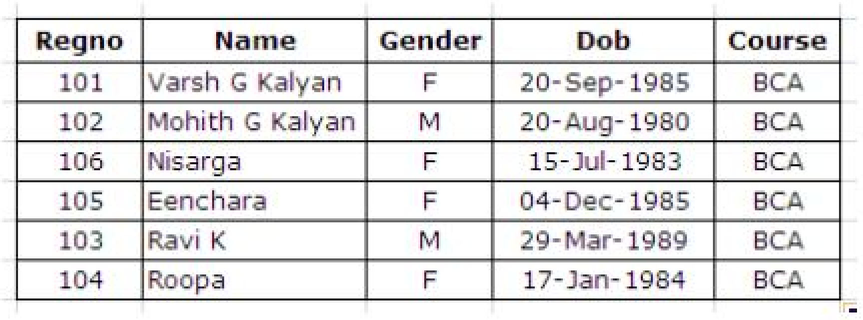
SET columnname1=expression1, columnname2=expression2;

Example: update the gktab table by changing its course to BCA.

UPDATE gktab SET course='BCA';

6 rows updated

SELECT * FROM gktab;



Regno	Name	Gender	Dob	Course
101	Varsh G Kalyan	F	20-Sep-1985	BCA
102	Mohith G Kalyan	M	20-Aug-1980	BCA
106	Nisarga	F	15-Jul-1983	BCA
105	Eenchara	F	04-Dec-1985	BCA
103	Ravi K	M	29-Mar-1989	BCA
104	Roopa	F	17-Jan-1984	BCA

In the above table, the course is changed to BCA for all the rows in the table.

Updating Records Conditionally

If you want to update a specific set of rows in table, then WHERE clause is used.

Syntax:

UPDATE tablename

SET Columnname1=Expression1, Columnname2=Expression2

WHERE Condition;

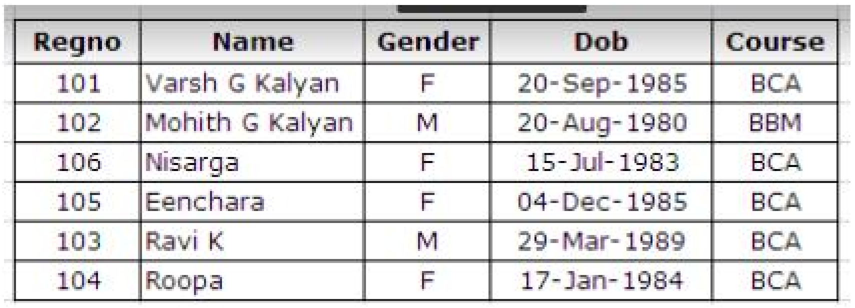
Example:

Update gktab table by changing the course BCA to BBM for Regno 102.

UPDATE gktab SET Course='BBM' WHERE Regno=102;

1 rows updated

```
SELECT * FROM gktab;
```



Regno	Name	Gender	Dob	Course
101	Varsh G Kalyan	F	20-Sep-1985	BCA
102	Mohith G Kalyan	M	20-Aug-1980	BBM
106	Nisarga	F	15-Jul-1983	BCA
105	Eenchara	F	04-Dec-1985	BCA
103	Ravi K	M	29-Mar-1989	BCA
104	Roopa	F	17-Jan-1984	BCA

MODIFYING THE STRUCTURE OF TABLES

The structure of a table can be modified by using the ALTER TABLE command. ALTER TABLE allows changing the structure of an existing table. With ALTER TABLE if is possible to add or delete columns, create or destroy indexes, change the data type of existing columns, or rename columns or the table itself.

ALTER TABLE works by making a temporary copy of the original table. The

alteration is performed on the copy, then the original table is deleted and the new

one is renamed. While ALTER TABLE is executing, the original table is still

readable by the users of ORACLE.

Restrictions on the ALTER TABLE

The following task cannot be performed when using the ALTER TABLE Clause:

- Change the name of the table.
- Change the name of the Column.
- Decrease the size of a column if table data exists.

ALTER TABLE Command can perform

- Adding New Columns.
- Dropping A Column from a Table.

- Modifying Existing Columns.

Adding New Columns

Syntax:

ALTER TABLE tablename

ADD(NewColumnname1 Datatype(size),

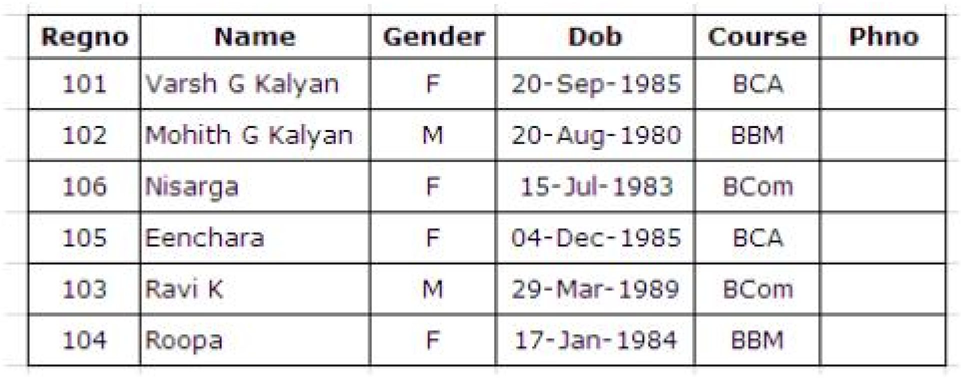
NewColumnname2 Datatype(size).....);

Example: Enter a new filed Phno to gktab.

ALTER TABLE gktab ADD(Phno number(10));

The table is altered with new column Phno

Select * from gktab;



Regno	Name	Gender	Dob	Course	Phno
101	Varsh G Kalyan	F	20-Sep-1985	BCA	
102	Mohith G Kalyan	M	20-Aug-1980	BBM	
106	Nisarga	F	15-Jul-1983	BCom	
105	Eenchara	F	04-Dec-1985	BCA	
103	Ravi K	M	29-Mar-1989	BCom	
104	Roopa	F	17-Jan-1984	BBM	

You can also use DESC gktab, to see the new column added to table.

Dropping A Column from a Table.

Syntax:

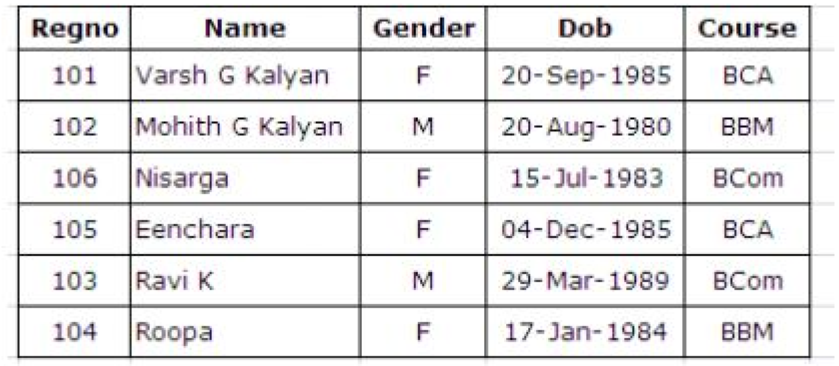
ALTER TABLE tablename DROP COLUMN Columnname;

Example: Drop the column Phno from gktab.

ALTER TABLE gktab DROP COLUMN Phno;

The table is altered, the column Phno is removed from the table.

Select * from gktab;



Regno	Name	Gender	Dob	Course
101	Varsh G Kalyan	F	20-Sep-1985	BCA
102	Mohith G Kalyan	M	20-Aug-1980	BBM
106	Nisarga	F	15-Jul-1983	BCom
105	Eenchara	F	04-Dec-1985	BCA
103	Ravi K	M	29-Mar-1989	BCom
104	Roopa	F	17-Jan-1984	BBM

You can also use DESC gktab, to see the column removed from the table.

Modifying Existing Columns.

Syntax:

ALTER TABLE tablename

MODIFY(Columnname Newdatatype(Newsize));

Example:

ALTER TABLE gktab MODIFY(Name VARCHAR(25));

The table altered with new size value 25.

DESC gktab;

RENAMING TABLES

Oracle allows renaming of tables. The rename operation is done atomically, which means that no other thread can access any of the tables while the rename process is running.

Syntax

RENAME tablename TO newtablename;

TRUNCATING TABLES

TRUNCATE command deletes the rows in the table permanently.

Syntax:

TRUNCATE TABLE tablename;

The number of deleted rows are not returned. Truncate operations drop and re-create the table, which is much faster than deleting rows one by one.

Example:

TRUNCATE TABLE gktab;

Table truncated i.e., all the rows are deleted permanently.

DESTROYING TABLES

Sometimes tables within a particular database become obsolete and need to be discarded. In such situation using the DROP TABLE statement with table name can destroy a specific table.

Syntax:

DROP TABLE tablename;

Example:

DROP TABLE gktab;

If a table is dropped all the records held within and the structure of the table is lost and cannot be recovered.

COMMIT and ROLLBACK

Commit

Commit command is used to permanently save any transaction into database.

```
SQL> commit;
```

Rollback

Rollback is used to undo the changes made by any command but only before a commit is done. We can't Rollback data which has been committed in the database with the help of the commit keyword or DDL Commands, because DDL commands are auto commit commands.

```
SQL> Rollback;
```

Difference between DELETE and DROP.

The DELETE command is used to remove rows from a table. After performing a DELETE operation you need to COMMIT or ROLLBACK the transaction to make the change permanent or to undo it.

The DROP command removes a table from the database. All the tables' rows, indexes and privileges will also be removed. The operation cannot be rolled back.

Difference between DELETE and TRUNCATE.

The DELETE command is used to remove rows from a table. After performing a DELETE operation you need to COMMIT or ROLLBACK the transaction to make the change permanent or to undo it.

TRUNCATE removes all rows from a table. The operation cannot be rolled back.

Difference between CHAR and VARCHAR.

CHAR

1. Used to store fixed length data.

2. The maximum characters the data type can hold is 255 characters.

3. It's 50% faster than VARCHAR.

4. Uses static memory allocation.

VARCHAR

1. Used to store variable length data.

2. The maximum characters the data type can hold is up to 4000 characters.

3. It's slower than CHAR.

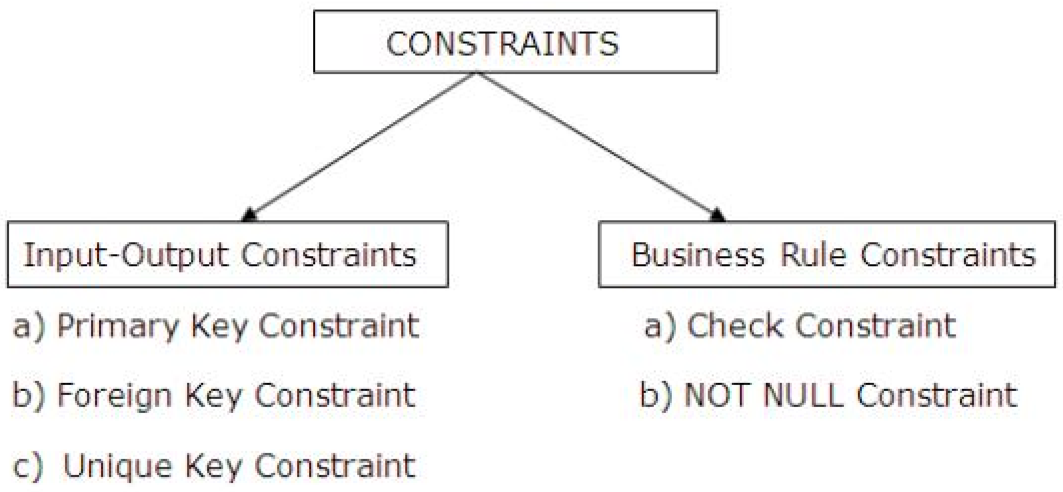
4. Uses dynamic memory allocation.

DATA CONSTRINTS

Oracle permits data constraints to be attached to table column via SQL syntax that checks data for integrity prior storage. Once data constraints are part of a table column construct, the oracle database engine checks the data being entered into a table column against the data constraints. If the data passes this check, it is stored in the table column, else the data is rejected. Even if a single column of the record being entered into the table fails a constraint, the entire record is rejected and not stored in the table.

Both CREATE TABLE and ALTER TABLE SQL verbs can be used to write SQL sentences that attach constraints to a table column. The constraints are a keyword. The constraint is rules that restrict the values for one or more columns in a table. The Oracle Server uses constraints to prevent invalid data entry into tables. The constraints store the validate data and without constraints we can just store invalid data. The constraints are an important part of the table.

Types of DATA CONSTRAINTS



Primary Key Constraint

A primary key can consist of one or more columns on a table. Primary key constraints define a column or series of columns that uniquely identify a given row in a table. Defining a primary key on a table is optional and you can only define a single primary key on a table. A primary key constraint can consist of one or many

columns (up to 32). When multiple columns are used as a primary key, they are called a composite key. Any column that is defined as a primary key column is automatically set with a NOT NULL status. The Primary key constraint can be applied at column level and table level.

Foreign Key Constraint

A foreign key constraint is used to enforce a relationship between two tables. A foreign key is a column (or a group of columns) whose values are derived from the Primary key or unique key of some other table. The table in which the foreign key is defined is called a Foreign table or Detail table. The table that defines primary key or unique key and is referenced by the foreign key is called Primary table or Master table. The master table can be referenced in the foreign key definition by using the clause REFERENCES Tablename.ColumnName when defining the foreign key,column attributes, in the detail table. The foreign key constraint can be applied at

column level and table level.

Unique Key Constraint

Unique key will not allow duplicate values. A table can have more than one Unique key. A unique constraint defines a column, or series of columns, that must be

unique in value. The UNIQUE constraint can be applied at column level and table

level.

CHECK Constraint

Business Rule validation can be applied to a table column by using CHECk constraint. CHECK constraints must be specified as a logical expression that evaluates either to TRUE or FALSE.

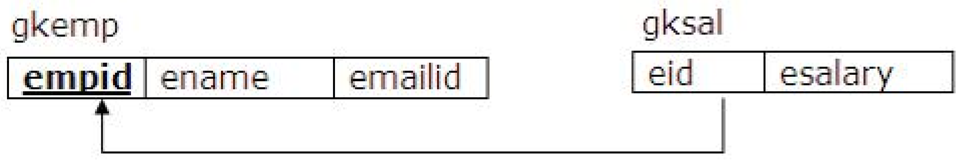
The CHECK constraint ensures that all values in a column satisfy certain conditions. Once defined, the database will only insert a new row or update an existing row if the new value satisfies the CHECK constraint. The CHECK constraint is used to ensure data quality. A CHECK constraint takes substantially longer to execute as compared to NOT NULL, PRIMARY KEY, FOREIGN KEY or UNIQUE. The CHECK constraint can be applied at column level and table level.

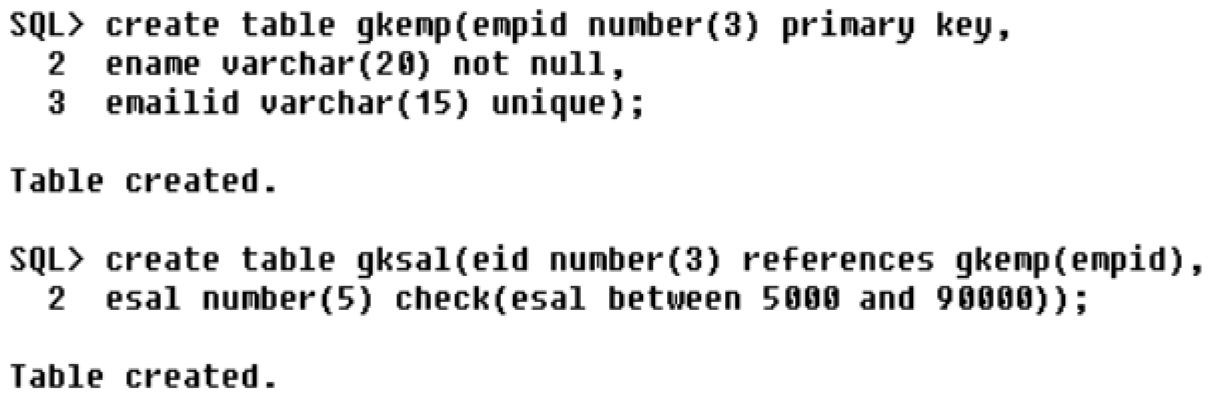
NOT NULL Constraint

The NOT NULL column constraint ensures that a table column cannot be left empty. When a column is defined as not null, then that column becomes a mandatory column. The NOT NULL constraint can only be applied at column level.

Example on Constraints

Consider the Table shown below





```
SQL> create table gkemp(empid number(3) primary key,
  2  ename varchar(20) not null,
  3  emailid varchar(15) unique);

Table created.

SQL> create table gksal(eid number(3) references gkemp(empid),
  2  esal number(5) check(esal between 5000 and 90000));

Table created.
```

Arithmetic Operators

Oracle allows arithmetic operators to be used while viewing records from a table or while performing data manipulation operations such as insert, updated and delete.

These are:

+ Addition

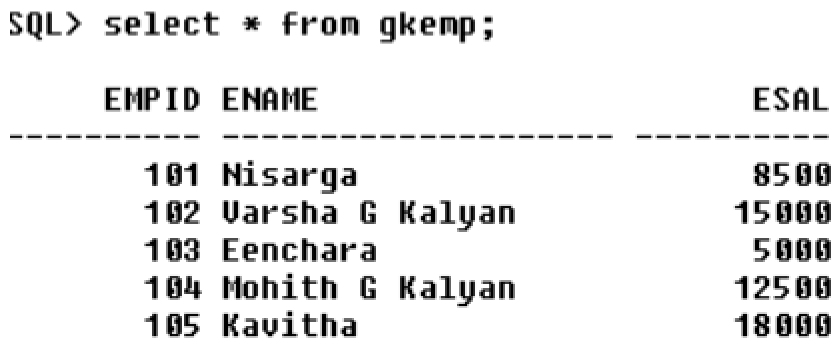
- Subtraction

/ Division

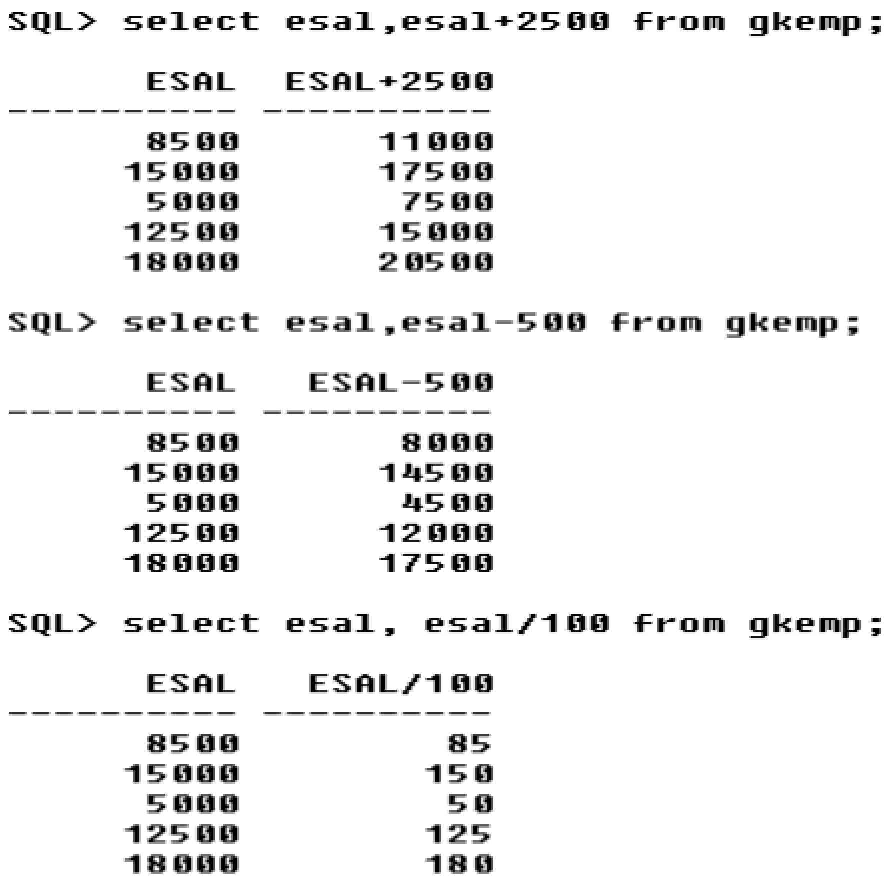
* Multiplication

() Enclosed Operations

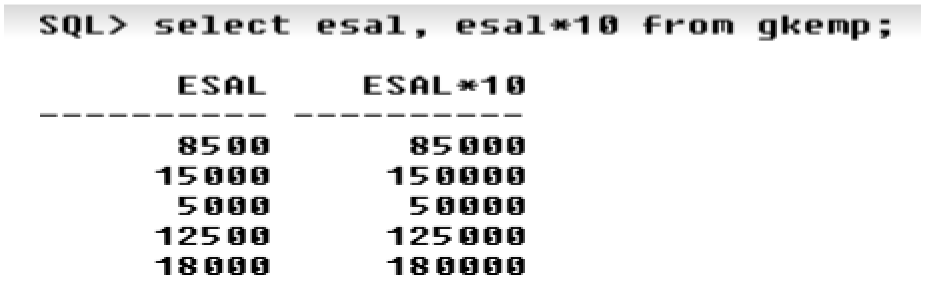
Consider the below employee table(gkemp)



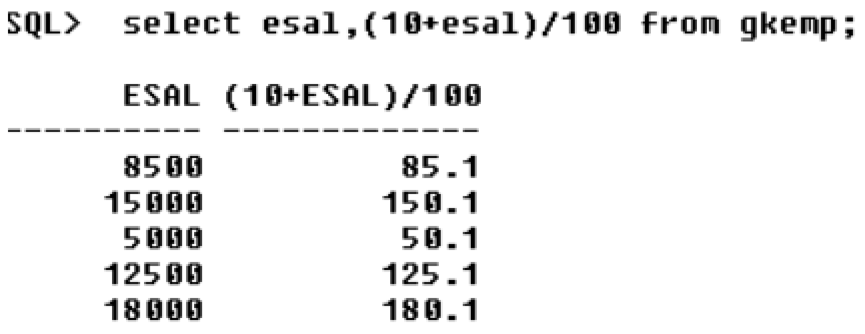
```
SQL> select * from gkemp;

     EMPID ENAME                      ESAL
---------- -------------------- ----------
       101 Nisarga                    8500
       102 Varsha G Kalyan           15000
       103 Eenchara                   5000
       104 Mohith G Kalyan           12500
       105 Kavitha                   18000
```



```
SQL> select esal,esal+2500 from gkemp;

      ESAL  ESAL+2500
---------- ----------
      8500      11000
     15000      17500
      5000       7500
     12500      15000
     18000      20500

SQL> select esal,esal-500 from gkemp;

      ESAL   ESAL-500
---------- ----------
      8500       8000
     15000      14500
      5000       4500
     12500      12000
     18000      17500

SQL> select esal, esal/100 from gkemp;

      ESAL   ESAL/100
---------- ----------
      8500         85
     15000        150
      5000         50
     12500        125
     18000        180
```



```
SQL> select esal, esal*10 from gkemp;

      ESAL    ESAL*10
---------- ----------
      8500      85000
     15000     150000
      5000      50000
     12500     125000
     18000     180000
```

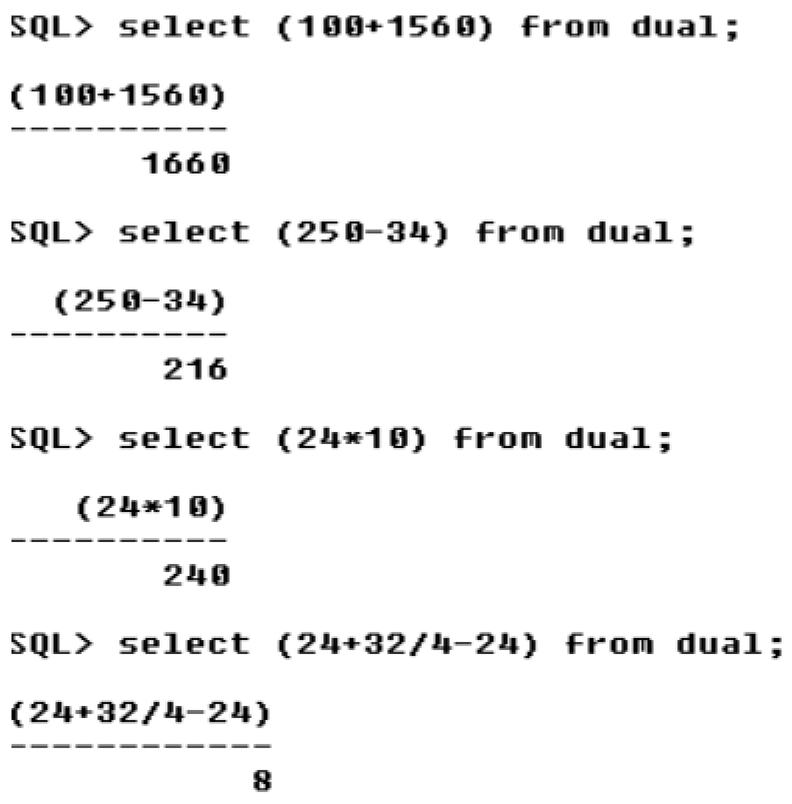


```
SQL>  select esal,(10+esal)/100 from gkemp;

      ESAL (10+ESAL)/100
---------- -------------
      8500          85.1
     15000         150.1
      5000          50.1
     12500         125.1
     18000         180.1
```

Special Note

The DUAL table is a special one-row, one-column table present by default in

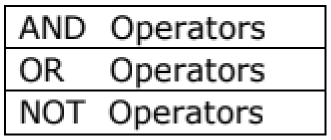
Oracle and other database installations. Dual is a dummy table.



```
SQL> select (100+1560) from dual;

(100+1560)
----------
      1660

SQL> select (250-34) from dual;

  (250-34)
----------
       216

SQL> select (24*10) from dual;

   (24*10)
----------
       240

SQL> select (24+32/4-24) from dual;

(24+32/4-24)
------------
           8
```

Logical Operators

Logical operators that can be used in SQL sentence are:



AND	Operators
OR	Operators
NOT	Operators

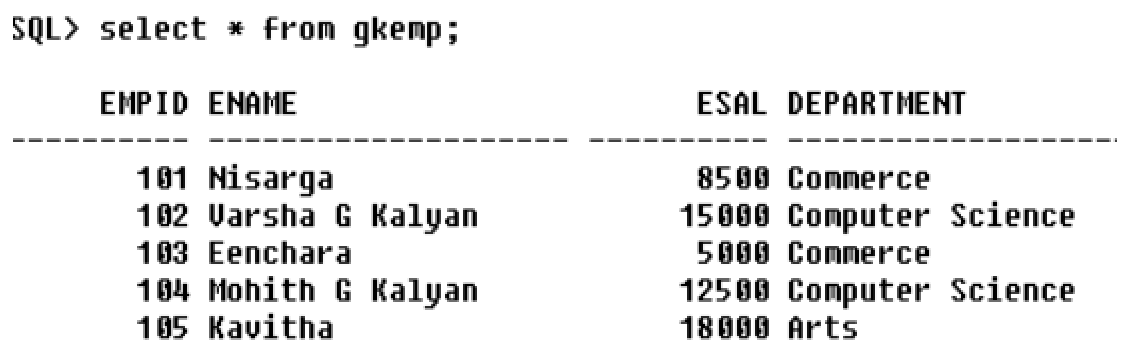
Operators Description

OR :-For the row to be selected at least one of the conditions must be true.

AND :-For a row to be selected all the specified conditions must be true.

NOT :-For a row to be selected the specified condition must be false.

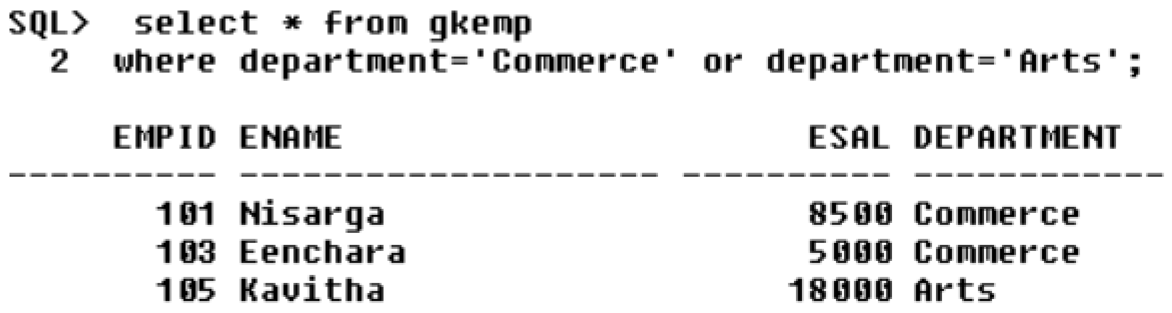
Consider the below employee table(gkemp)



```
SQL> select * from gkemp;

     EMPID ENAME                      ESAL DEPARTMENT
---------- -------------------- ---------- ------------------
       101 Nisarga                    8500 Commerce
       102 Varsha G Kalyan           15000 Computer Science
       103 Eenchara                   5000 Commerce
       104 Mohith G Kalyan           12500 Computer Science
       105 Kavitha                   18000 Arts
```

For example: if you want to find the names of employees who are working either

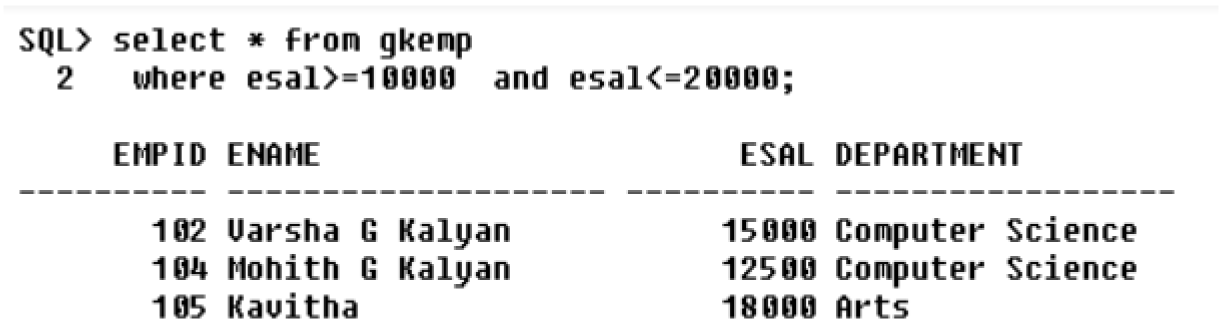
in Commerce or Arts department, the query would be like,



```
SQL>  select * from gkemp
  2  where department='Commerce' or department='Arts';

     EMPID ENAME                      ESAL DEPARTMENT
---------- -------------------- ---------- -----------
       101 Nisarga                    8500 Commerce
       103 Eenchara                   5000 Commerce
       105 Kavitha                   18000 Arts
```

For example: To find the names of the employee whose salary between10000 to

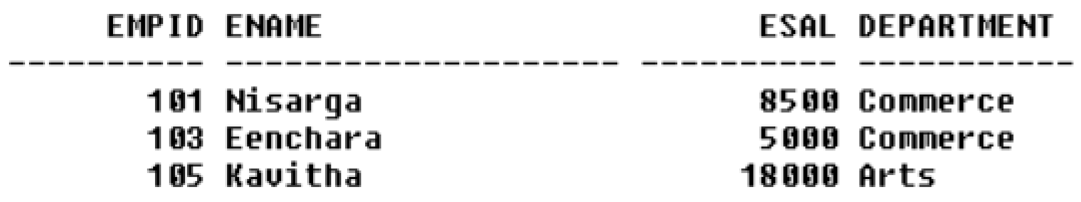
20000, the query would be like,



```
SQL> select * from gkemp
  2   where esal>=10000  and esal<=20000;

     EMPID ENAME                      ESAL DEPARTMENT
---------- -------------------- ---------- ------------------
       102 Varsha G Kalyan           15000 Computer Science
       104 Mohith G Kalyan           12500 Computer Science
       105 Kavitha                   18000 Arts
```

For example: If you want to find out the names of the employee who do not

belong to computer science department, the query would be like,



```
     EMPID ENAME                      ESAL DEPARTMENT
---------- -------------------- ---------- -----------
       101 Nisarga                    8500 Commerce
       103 Eenchara                   5000 Commerce
       105 Kavitha                   18000 Arts
```

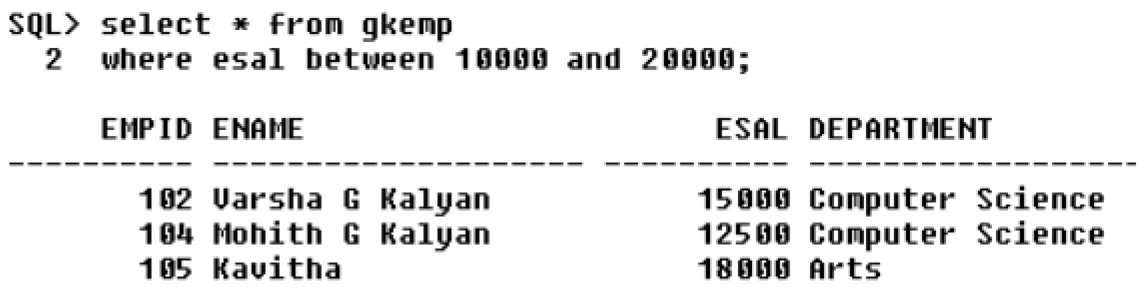
Range Searching (BETWEEN)

In order to select the data that is within a range of values, the BETWEEN operator is used. The BETWEEN operator allows the selection of rows that contain values within a specified lower and upper limit. The range coded after the word BETWEEN is inclusive.

The lower value must be coded first. The two values in between the range must be linked with the keyword AND. The BETWEEN operator can be used with both character and numeric data types. However, the data types cannot be mixed.

For example: Find the names of the employee whose salary between10000 and

20000, the query would be like,



```
SQL> select * from gkemp
  2  where esal between 10000 and 20000;

     EMPID ENAME                      ESAL DEPARTMENT
---------- -------------------- ---------- ------------------
       102 Varsha G Kalyan           15000 Computer Science
       104 Mohith G Kalyan           12500 Computer Science
       105 Kavitha                   18000 Arts
```

Pattern Matching (LIKE, IN, NOT IN)

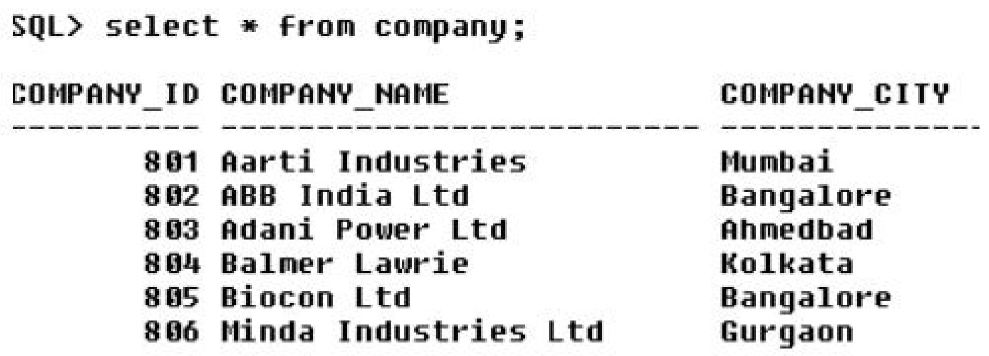
LIKE

The LIKE predicate allows comparison of one string value with another string value, which is not identical. this is achieved by using wild characters. Two wild characters that are available are:

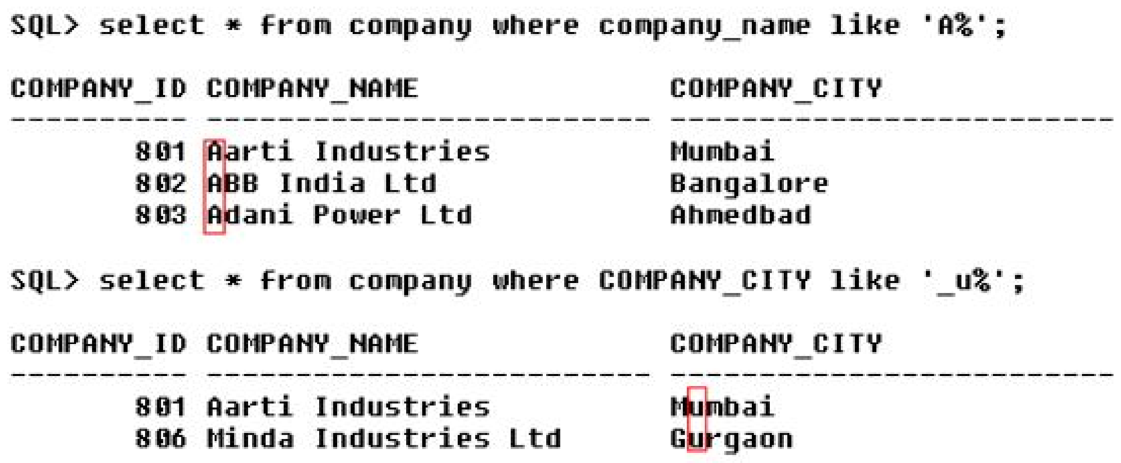
For character data types:

% allows to match any string of any length (including zero length).

_ allows to match on a single character.



```
SQL> select * from company;

COMPANY_ID COMPANY_NAME                COMPANY_CITY
---------- ------------------------- --------------
       801 Aarti Industries          Mumbai
       802 ABB India Ltd             Bangalore
       803 Adani Power Ltd           Ahmedbad
       804 Balmer Lawrie             Kolkata
       805 Biocon Ltd                Bangalore
       806 Minda Industries Ltd      Gurgaon
```



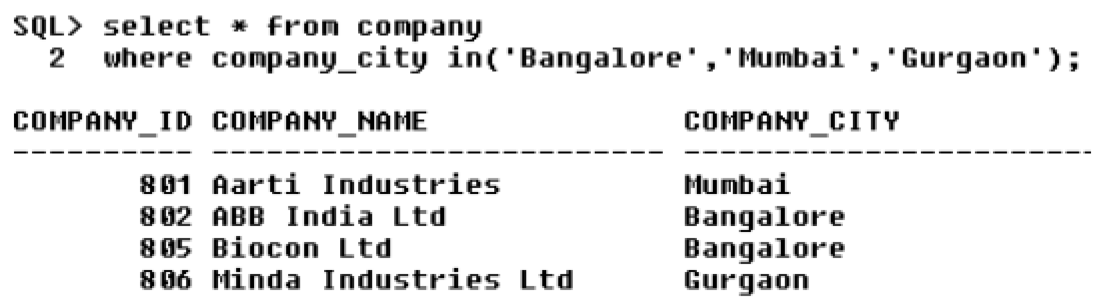
```
SQL> select * from company where company_name like 'A%';

COMPANY_ID COMPANY_NAME                COMPANY_CITY
---------- ------------------------- -------------------------
       801 Aarti Industries          Mumbai
       802 ABB India Ltd             Bangalore
       803 Adani Power Ltd           Ahmedbad

SQL> select * from company where COMPANY_CITY like '_u%';

COMPANY_ID COMPANY_NAME                COMPANY_CITY
---------- ------------------------- -------------------------
       801 Aarti Industries          Mumbai
       806 Minda Industries Ltd      Gurgaon
```

IN

The IN operator is used when you want to compare a column with more than one value. It is similar to an OR condition.

For example: If you want to find the names of company located in the city

Bangalore, Mumbai, Gurgaon, the query would be like,



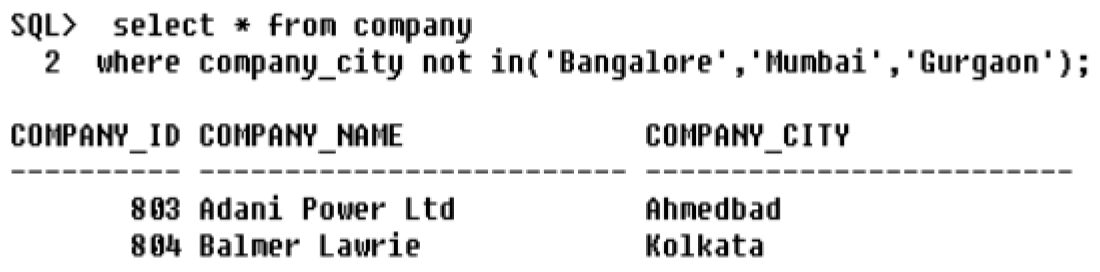
```
SQL> select * from company
  2  where company_city in('Bangalore','Mumbai','Gurgaon');

COMPANY_ID COMPANY_NAME                COMPANY_CITY
---------- ------------------------- -----------------------
       801 Aarti Industries          Mumbai
       802 ABB India Ltd             Bangalore
       805 Biocon Ltd                Bangalore
       806 Minda Industries Ltd      Gurgaon
```

NOT IN

The NOT IN operator is opposite to IN.

For example: If you want to find the names of company located in the other city

of Bangalore, Mumbai, Gurgaon, the query would be like,



```
SQL>  select * from company
  2  where company_city not in('Bangalore','Mumbai','Gurgaon');

COMPANY_ID COMPANY_NAME              COMPANY_CITY
---------- ------------------------- -------------------------
       803 Adani Power Ltd           Ahmedbad
       804 Balmer Lawrie             Kolkata
```

Column Aliases(Renaming Columns) in Oracle:

Sometimes you want to change the column headers in the report. For this you can use column aliases in oracle. Oracle has provided excellent object oriented techniques as its robust database. It always good to practice and implement column aliases since it will make your code readable while using this columns.

to add column aliases to your sql queries.

Give a column alias name separated by space after the column name.

Select DOB DateofBirth from gkstudent

In the above query, the word in bold is column aliases.

ORACLE FUNCTIONS

Oracle functions serve the purpose of manipulating data items and returning a result. Functions are also capable of accepting user-supplied variables or constants and operating on them. Such variables or constants are called arguments. Any number of arguments(or no arguments at all) can be passed to a function in the following format.

Function_Name(arguments1,arguments2.....)

Oracle functions can be clubbed together depending upon whether they operate on a single row or a group of rows retrieved from a table. Accordingly, functions can be classified as follows:

- Group Functions(Aggregate Functions): Function that act on a set of values are called group functions.

- Scalar Functions(Single Row Functions): Function that act on only one value at a time are called scalar functions.

- String Functions: for string data type
- Numeric functions: for Number data type
- Conversion function: for conversion of one data type to another.
- Date conversions: for date data type.

a) SQL Aggregate / Group Functions

Group functions return results based on groups of rows, rather than on single rows. returns the number of rows in the query.SQL aggregate functions return a single value, calculated from values in a column.

Useful aggregate functions:

a) COUNT() - Returns the number of rows

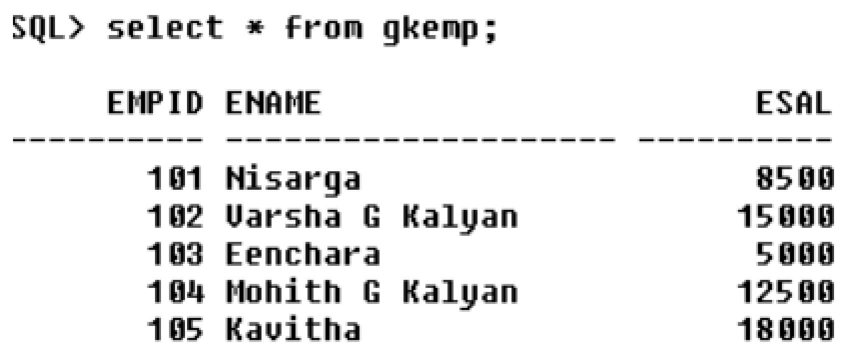
b) AVG() - Returns the average value

c) MAX() - Returns the largest value

d) MIN() - Returns the smallest value

e) SUM() - Returns the total sum

Consider the below employee table(gkemp)

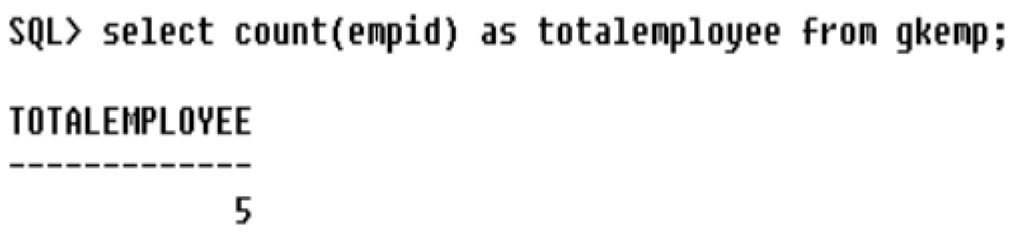


```
SQL> select * from gkemp;

     EMPID ENAME                      ESAL
---------- -------------------- ----------
       101 Nisarga                    8500
       102 Varsha G Kalyan           15000
       103 Eenchara                   5000
       104 Mohith G Kalyan           12500
       105 Kavitha                   18000
```

a) COUNT()

The COUNT() function counts number of values present in the column

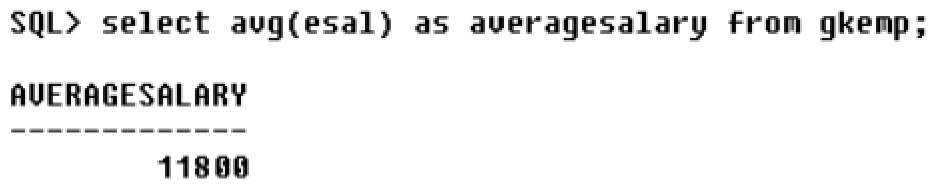
excluding Null values.



```
SQL> select count(empid) as totalemployee from gkemp;

TOTALEMPLOYEE
-------------
            5
```

b) AVG()

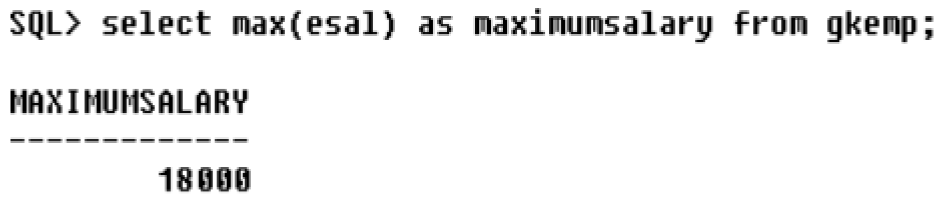
The AVG() function returns the average value of a column specified.



```
SQL> select avg(esal) as averagesalary from gkemp;

AVERAGESALARY
-------------
        11800
```

c) MAX()

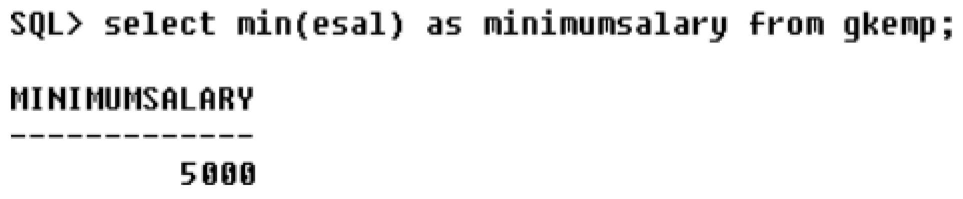
The MAX() function returns the highest value of a particular column.



```
SQL> select max(esal) as maximumsalary from gkemp;

MAXIMUMSALARY
-------------
        18000
```

d) MIN()

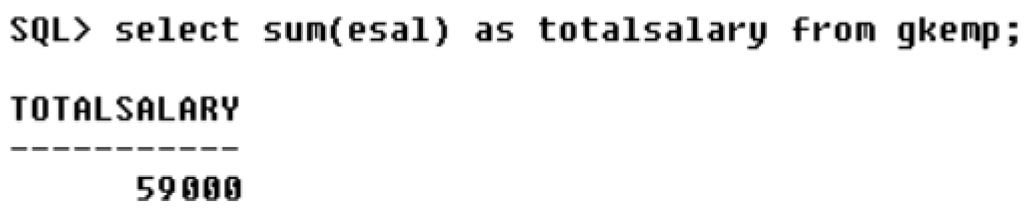
The MIN() function returns the smallest value of a particular column.



```
SQL> select min(esal) as minimumsalary from gkemp;

MINIMUMSALARY
-------------
         5000
```

e) SUM()

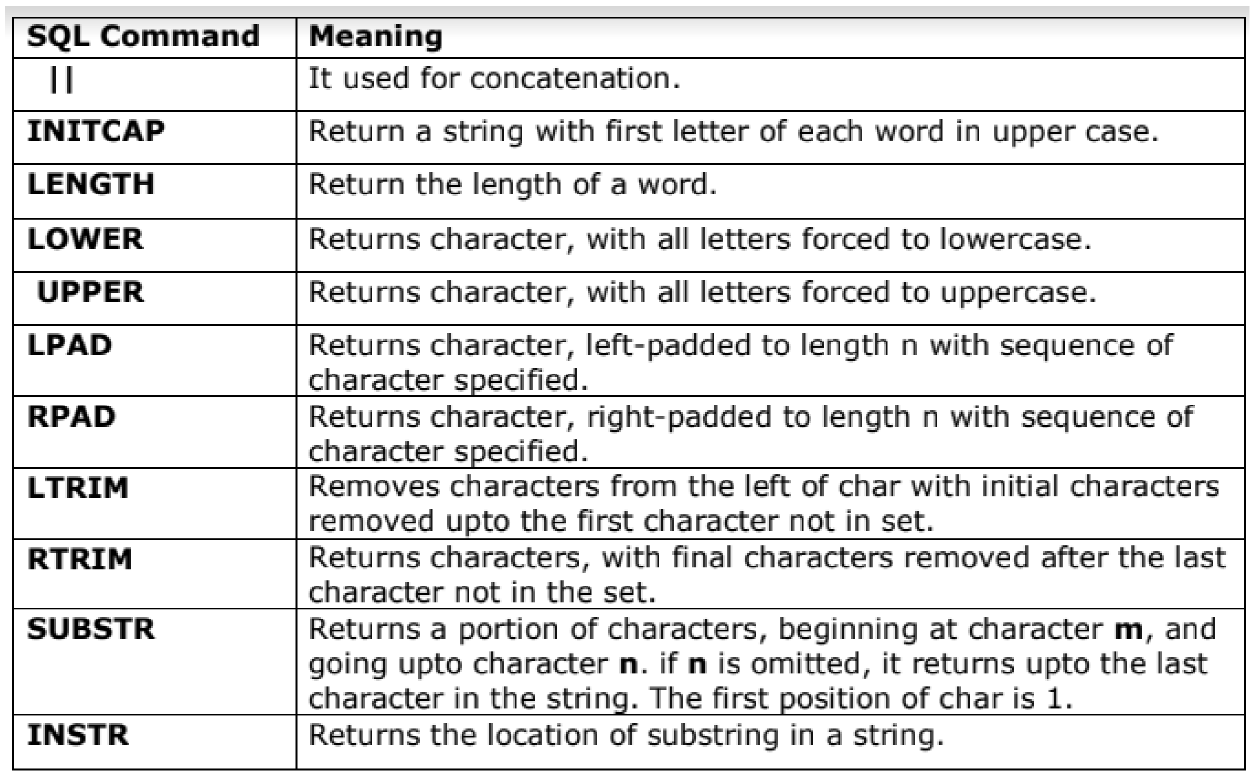
The SUM() function returns the sum of column values.



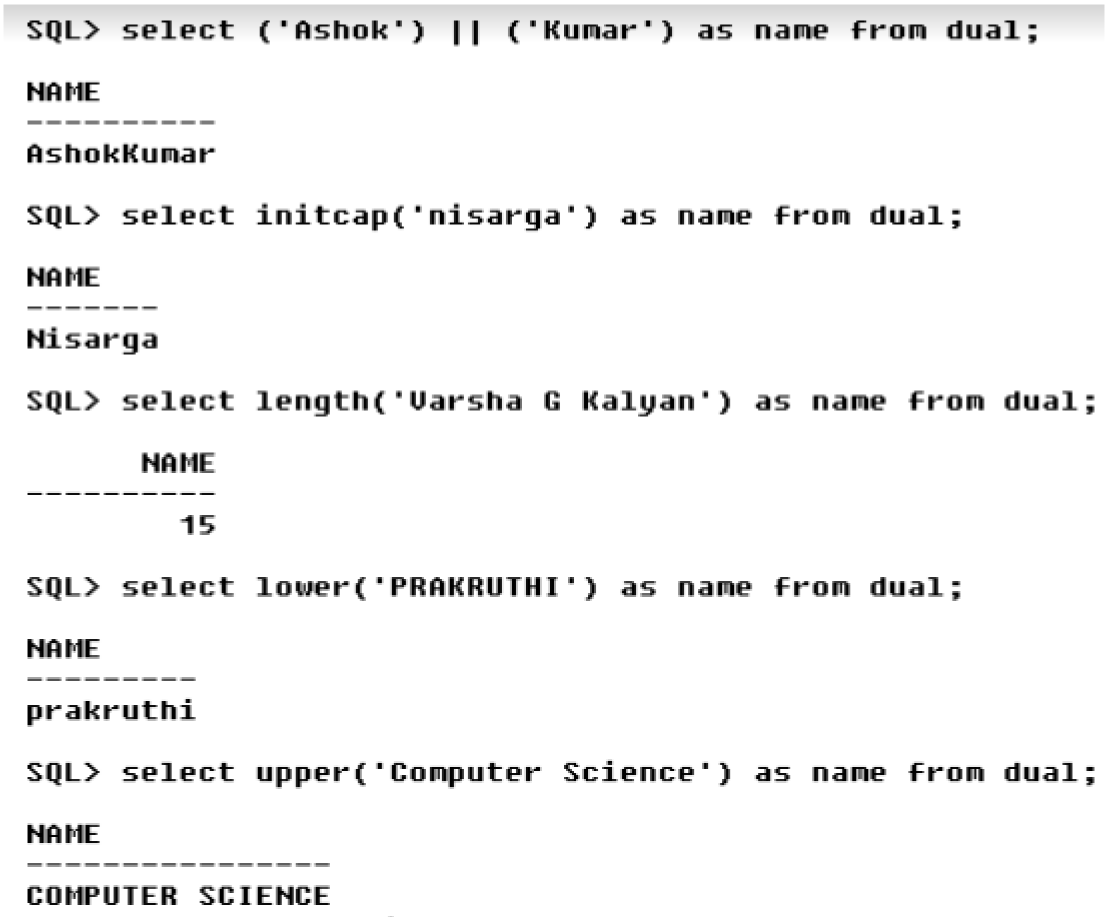
```
SQL> select sum(esal) as totalsalary from gkemp;

TOTALSALARY
-----------
      59000
```

b) SQL String Functions

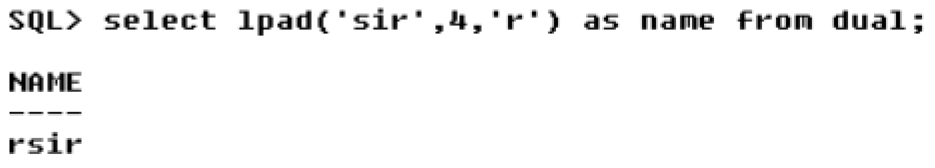
SQL string functions are used primarily for string manipulation. The following table details the important string functions:



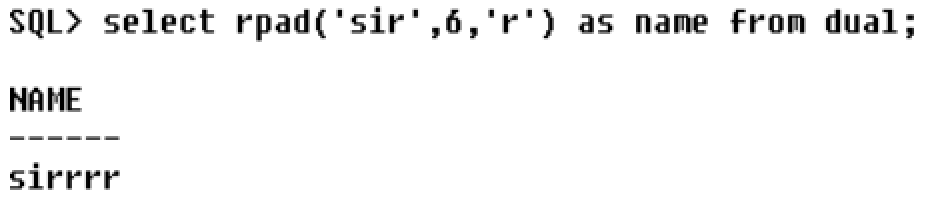
SQL Command	Meaning
||	It used for concatenation.
INITCAP	Return a string with first letter of each word in upper case.
LENGTH	Return the length of a word.
LOWER	Returns character, with all letters forced to lowercase.
UPPER	Returns character, with all letters forced to uppercase.
LPAD	Returns character, left-padded to length n with sequence of character specified.
RPAD	Returns character, right-padded to length n with sequence of character specified.
LTRIM	Removes characters from the left of char with initial characters removed upto the first character not in set.
RTRIM	Returns characters, with final characters removed after the last character not in the set.
SUBSTR	Returns a portion of characters, beginning at character **m**, and going upto character **n**. if **n** is omitted, it returns upto the last character in the string. The first position of char is 1.
INSTR	Returns the location of substring in a string.



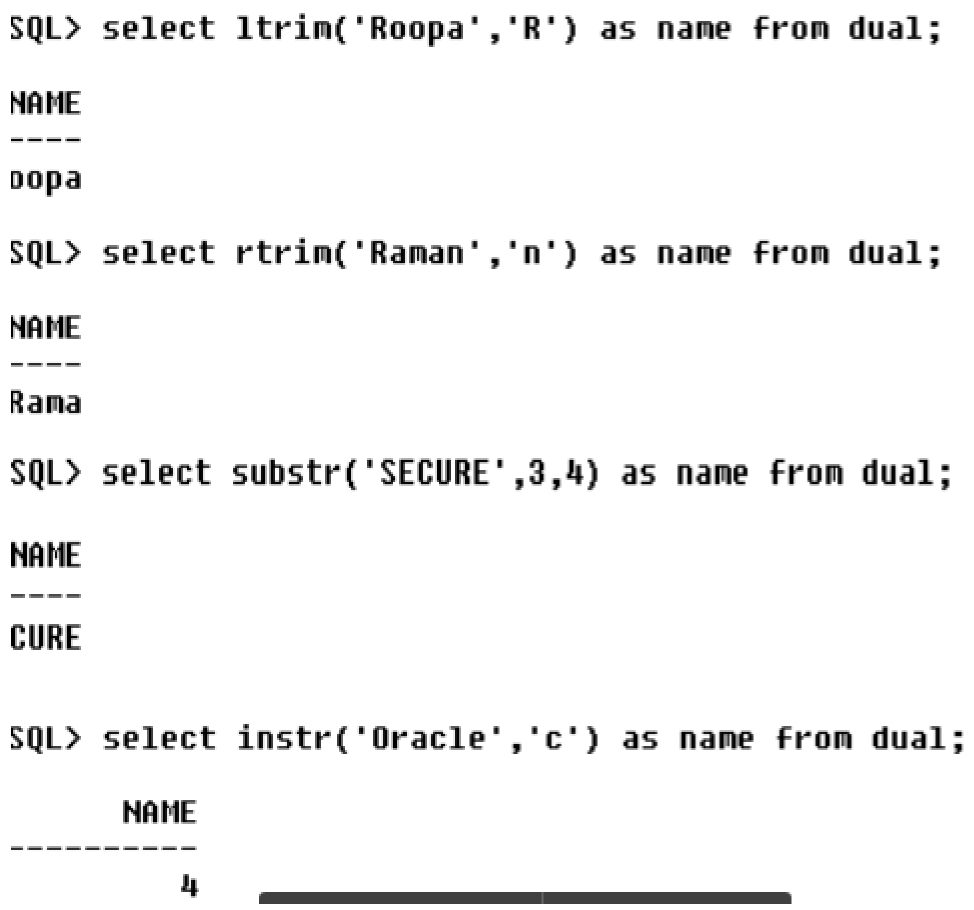
```
SQL> select ('Ashok') || ('Kumar') as name from dual;

NAME
----------
AshokKumar

SQL> select initcap('nisarga') as name from dual;

NAME
-------
Nisarga

SQL> select length('Varsha G Kalyan') as name from dual;

      NAME
----------
        15

SQL> select lower('PRAKRUTHI') as name from dual;

NAME
---------
prakruthi

SQL> select upper('Computer Science') as name from dual;

NAME
----------------
COMPUTER SCIENCE
```



```
SQL> select lpad('sir',4,'r') as name from dual;

NAME
----
rsir
```



```
SQL> select rpad('sir',6,'r') as name from dual;

NAME
------
sirrrr
```

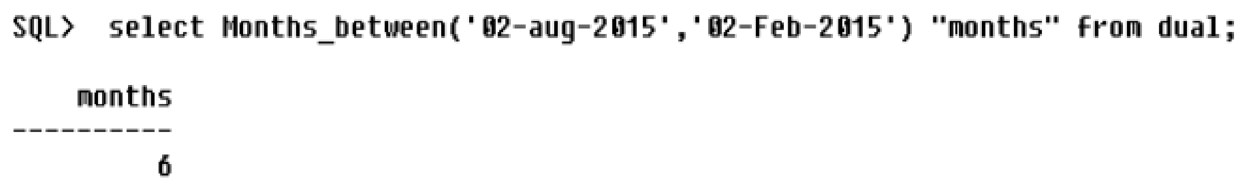


```
SQL> select ltrim('Roopa','R') as name from dual;

NAME
----
oopa

SQL> select rtrim('Raman','n') as name from dual;

NAME
----
Rama

SQL> select substr('SECURE',3,4) as name from dual;

NAME
----
CURE

SQL> select instr('Oracle','c') as name from dual;

      NAME
----------
         4
```

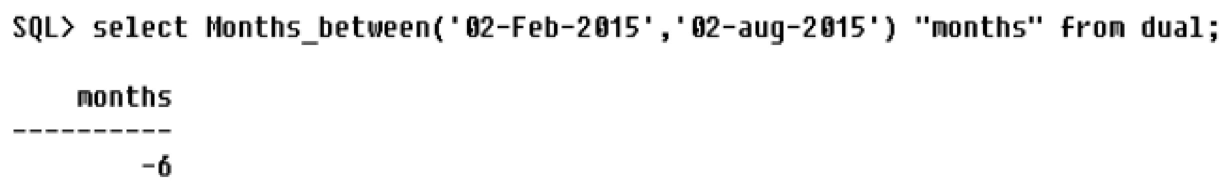
Date Conversion Functions

SQL Command	***Meaning***
SYSDATE	It shows the system date.
ADD_MONTHS(d, n)	Returns date after adding the number of months specified in the function.
LAST_DAY(d)	Returns the date of the month specified with the function.
MONTHS_BETWEEN(d1,d2)	Returns number of months between d1 and d2.
NEXT-DAY(date, char)	Returns the date of the first weekday named by char that is after the date named by date. char must be a day of the week
ROUND(date, [format])	Returns a date rounded to a specific unit of measure. If the second parameter is omitted, the ROUND function will round the date to the nearest day.
Conversion Functions	
TO_DATE(<char value>[,<format>])	Converts a character field to a date field
TO-CHAR(<date value>[,<format>])	Converts a date field to a character field

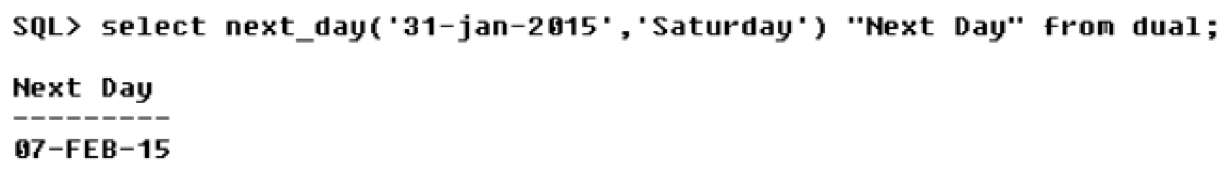
```
SQL> select sysdate "date" from dual;

date
---------
18-JAN-15

SQL> select add_months(sysdate,4) "Add Months" from dual;

Add Month
---------
18-MAY-15

SQL> select sysdate, last_day(sysdate) "LastDay" from dual;

SYSDATE   LastDay
--------- ---------
18-JAN-15 31-JAN-15
```



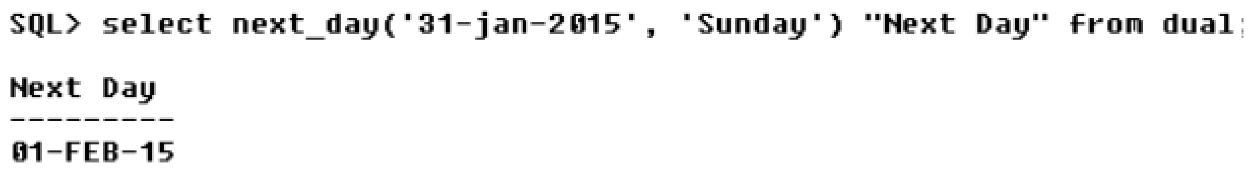
```
SQL>  select Months_between('02-aug-2015','02-Feb-2015') "months" from dual;

    months
----------
         6
```



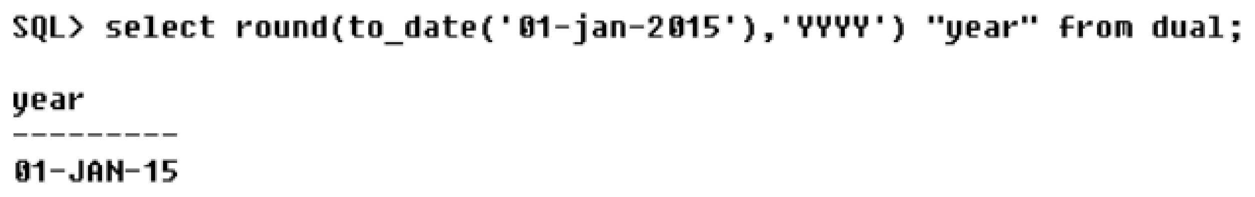
```
SQL> select Months_between('02-Feb-2015','02-aug-2015') "months" from dual;

    months
----------
        -6
```



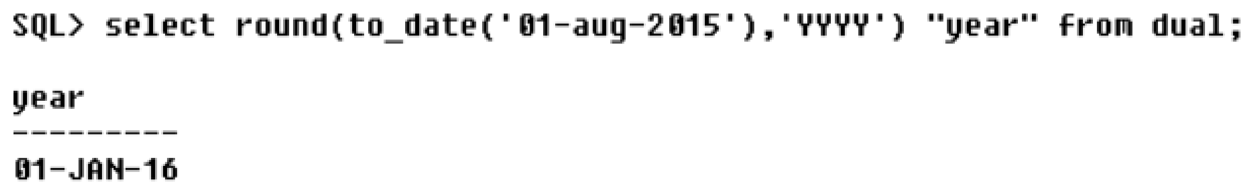
```
SQL> select next_day('31-jan-2015','Saturday') "Next Day" from dual;

Next Day
---------
07-FEB-15
```



```
SQL> select next_day('31-jan-2015', 'Sunday') "Next Day" from dual
Next Day
---------
01-FEB-15
```



```
SQL> select round(to_date('01-jan-2015'),'YYYY') "year" from dual;

year
---------
01-JAN-15
```

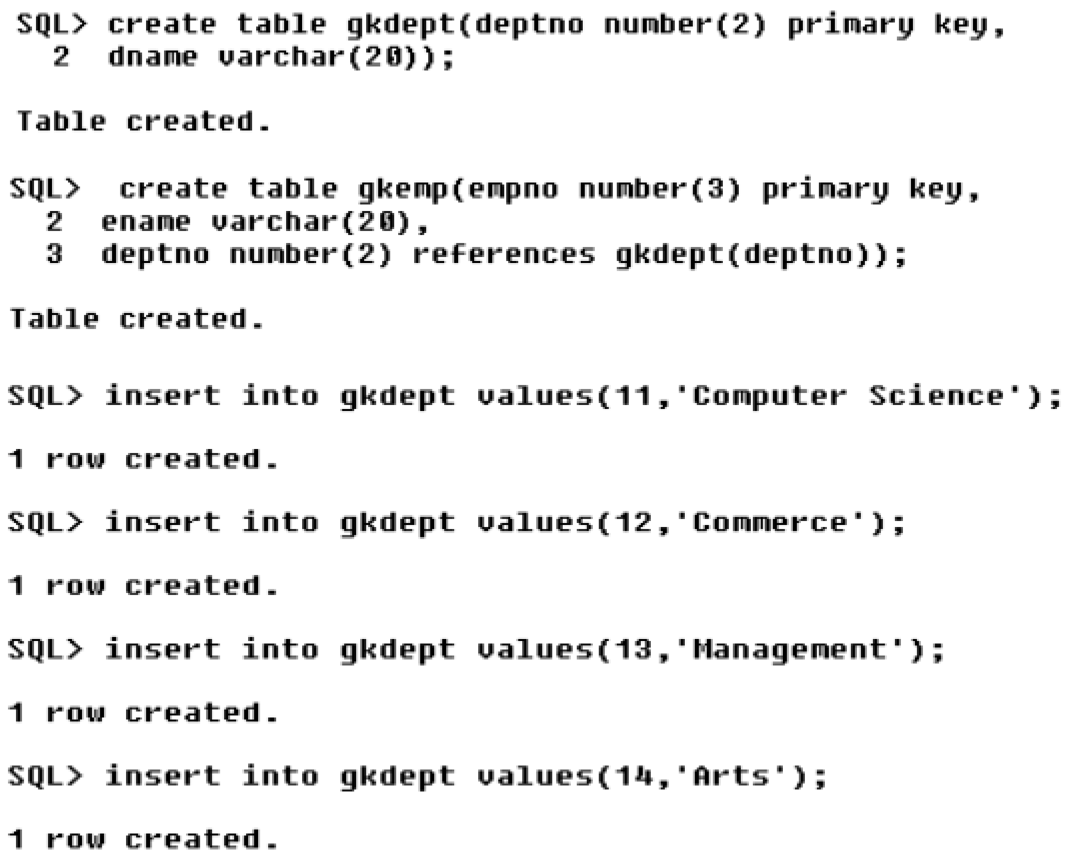


```
SQL> select round(to_date('01-aug-2015'),'YYYY') "year" from dual;

year
---------
01-JAN-16
```

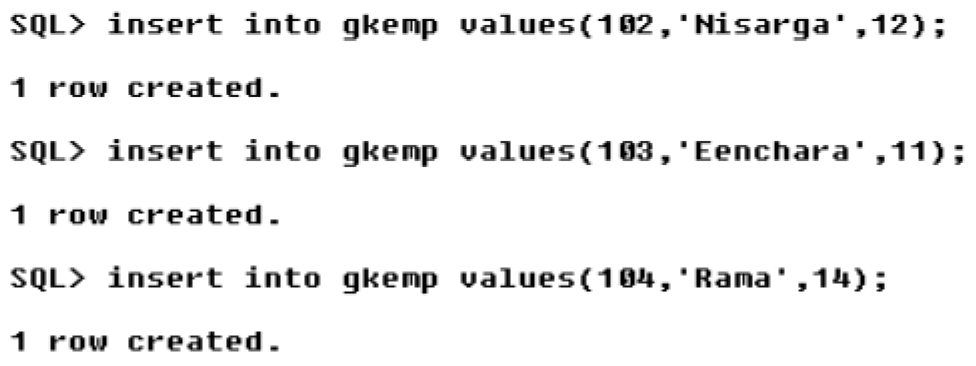
SET OPERATORS and JOINS

Set Operators

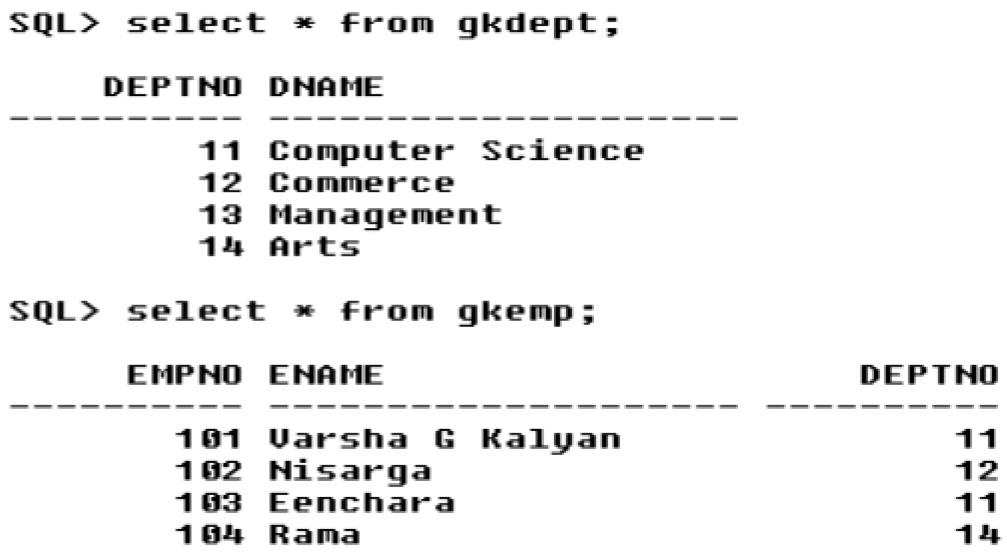
Consider the below tables for set operators examples



```
SQL> create table gkdept(deptno number(2) primary key,
  2  dname varchar(20));

Table created.

SQL>  create table gkemp(empno number(3) primary key,
  2  ename varchar(20),
  3  deptno number(2) references gkdept(deptno));

Table created.

SQL> insert into gkdept values(11,'Computer Science');

1 row created.

SQL> insert into gkdept values(12,'Commerce');

1 row created.

SQL> insert into gkdept values(13,'Management');

1 row created.

SQL> insert into gkdept values(14,'Arts');

1 row created.
```



```
SQL> insert into gkemp values(102,'Nisarga',12);

1 row created.

SQL> insert into gkemp values(103,'Eenchara',11);

1 row created.

SQL> insert into gkemp values(104,'Rama',14);

1 row created.
```



```
SQL> select * from gkdept;

    DEPTNO DNAME
---------- --------------------
        11 Computer Science
        12 Commerce
        13 Management
        14 Arts

SQL> select * from gkemp;

     EMPNO ENAME                    DEPTNO
---------- -------------------- ----------
       101 Varsha G Kalyan              11
       102 Nisarga                      12
       103 Eenchara                     11
       104 Rama                         14
```

Set operators combine the result of two quires into single one. The different set operators are:

UNION

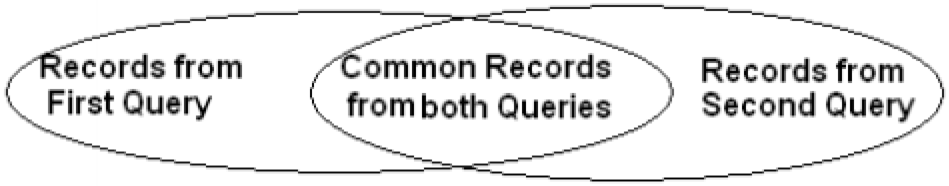
UNION ALL

INTERSECT

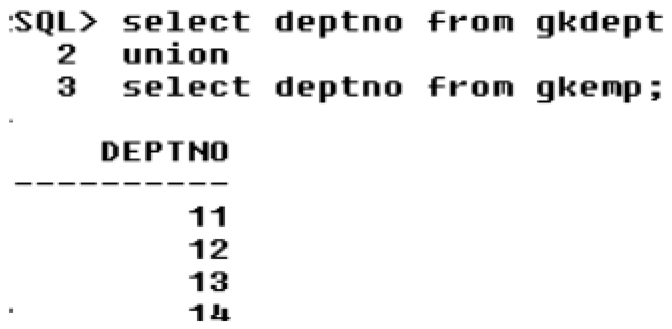
MINUS

Union Clause

UNION is used to combine the result of two or more SELECT statements. However it will eliminate duplicate rows from its result set. In case of UNION, number of columns in all the query must be same and datatype must be same in both the tables.



<u>Union Example</u>

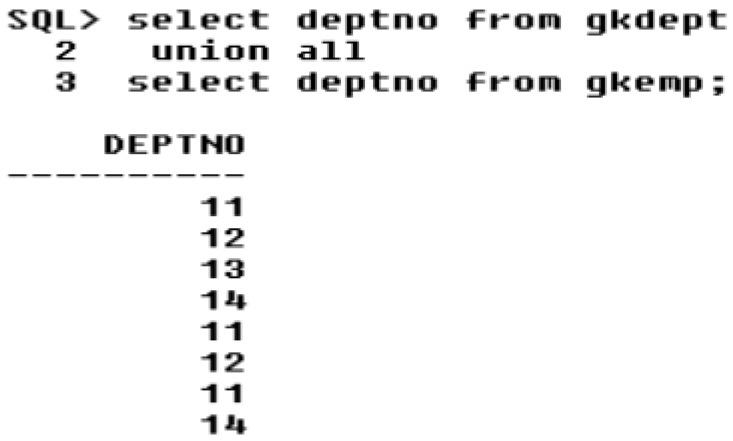


```
SQL> select deptno from gkdept
  2  union
  3  select deptno from gkemp;

    DEPTNO
----------
        11
        12
        13
        14
```

Union ALL Clause

Same as UNION but it shows the duplicate rows

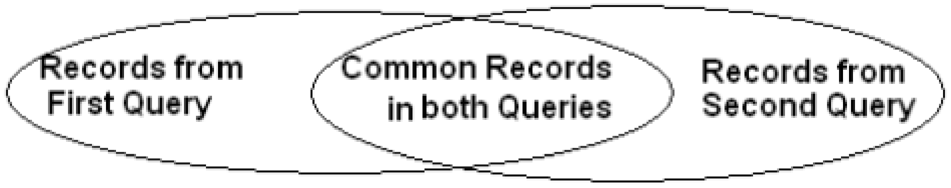
<u>Union All Example</u>

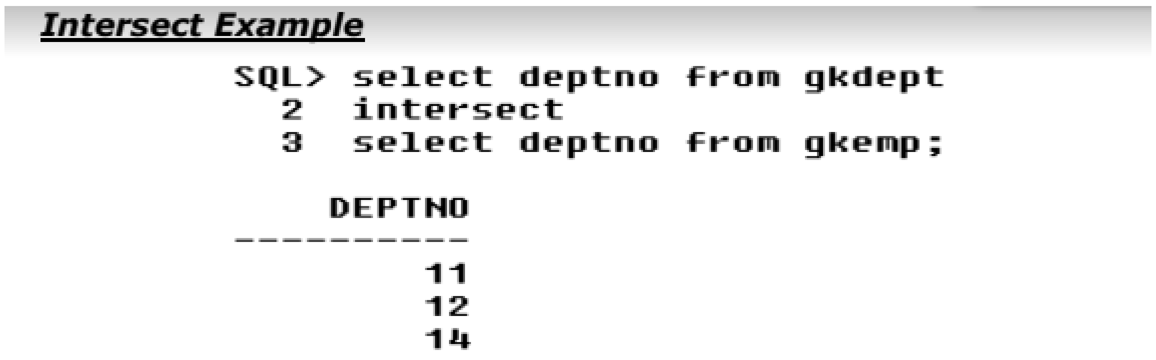


```
SQL> select deptno from gkdept
  2   union all
  3  select deptno from gkemp;

    DEPTNO
----------
        11
        12
        13
        14
        11
        12
        11
        14
```

Intersect Clause

Intersect is used to combine two SELECT statements, but it only returns the records which are common from both SELECT statement. In case of intersect the number of columns in all the query and datatype must be same.

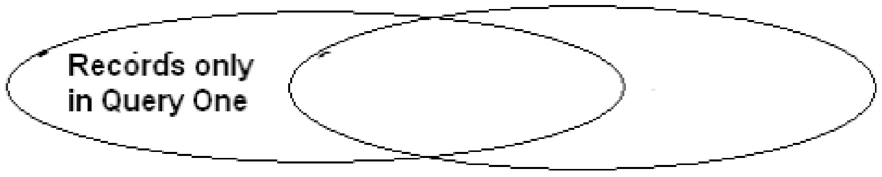


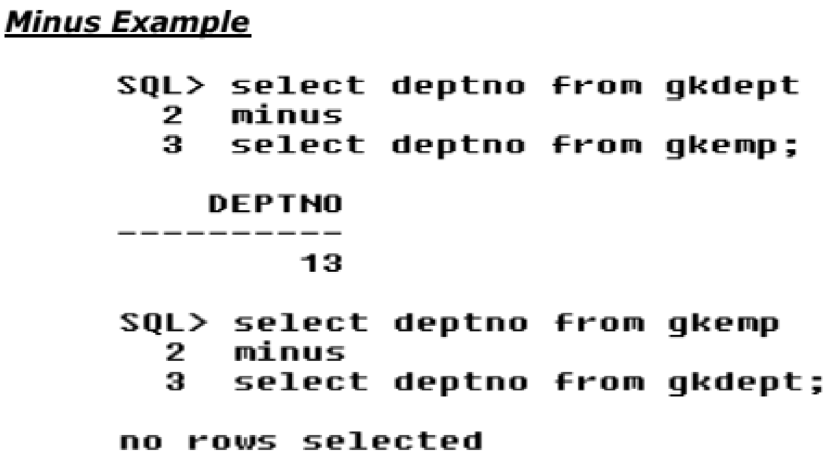

Intersect Example

```
SQL> select deptno from gkdept
  2  intersect
  3  select deptno from gkemp;

    DEPTNO
----------
        11
        12
        14
```

Minus Clause

Minus combines result of two SELECT statement and return only those result which belongs to the first set of result.




Minus Example

```
SQL> select deptno from gkdept
  2  minus
  3  select deptno from gkemp;

    DEPTNO
----------
        13

SQL> select deptno from gkemp
  2  minus
  3  select deptno from gkdept;

no rows selected
```

JOINS

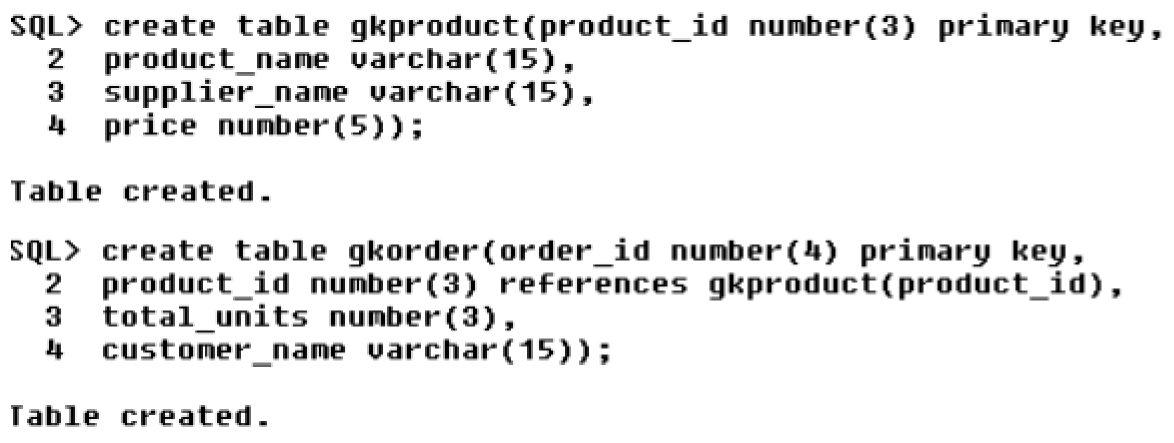
The SQL Joins clause is used to combine records from two or more tables in a database. A JOIN is a means for combining fields from two tables by using values common to each. Here, it is noticeable that the join is performed in the WHERE clause. Several operators can be used to join tables, such as =, <, >, <>, <=, >=, !=, BETWEEN, LIKE, and NOT; they can all be used to join tables. However, the most common

operator is the equal symbol. SQL Join Types:

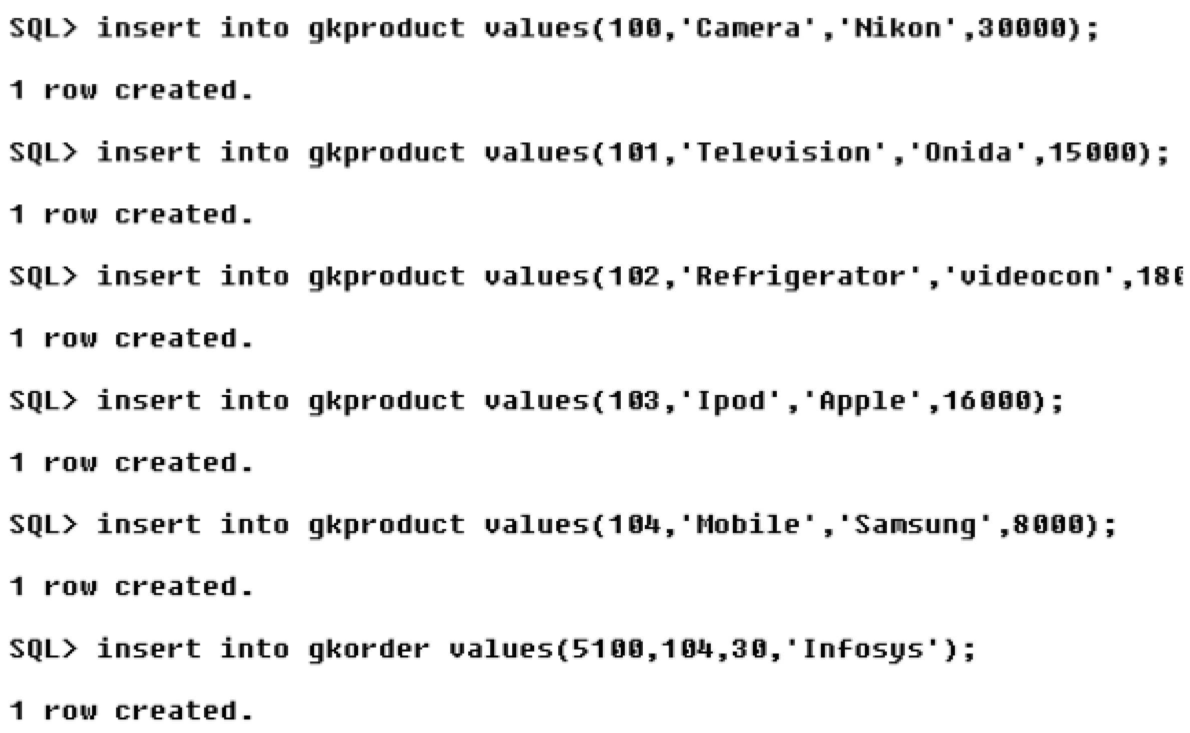
There are different types of joins available in SQL:

- INNER
- OUTER(LEFT,RIGHT,FULL)
- CROSS

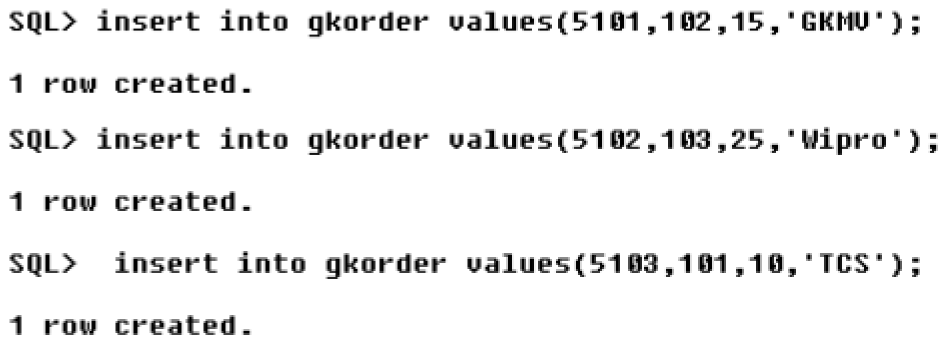
Consider the below tables for Join Operations examples



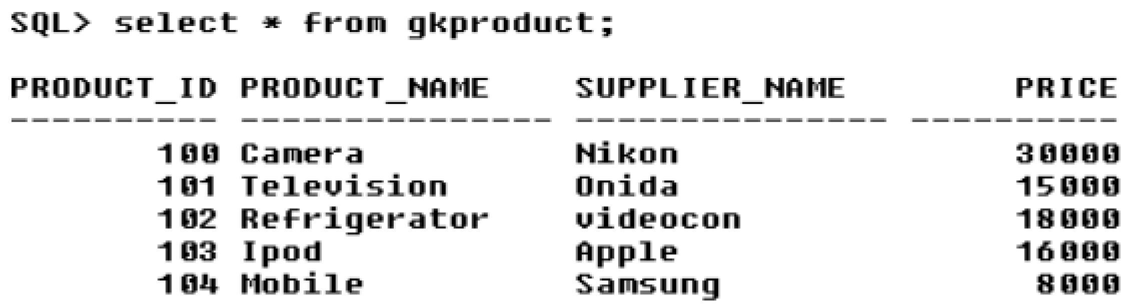
```
SQL> create table gkproduct(product_id number(3) primary key,
  2  product_name varchar(15),
  3  supplier_name varchar(15),
  4  price number(5));

Table created.

SQL> create table gkorder(order_id number(4) primary key,
  2  product_id number(3) references gkproduct(product_id),
  3  total_units number(3),
  4  customer_name varchar(15));

Table created.
```



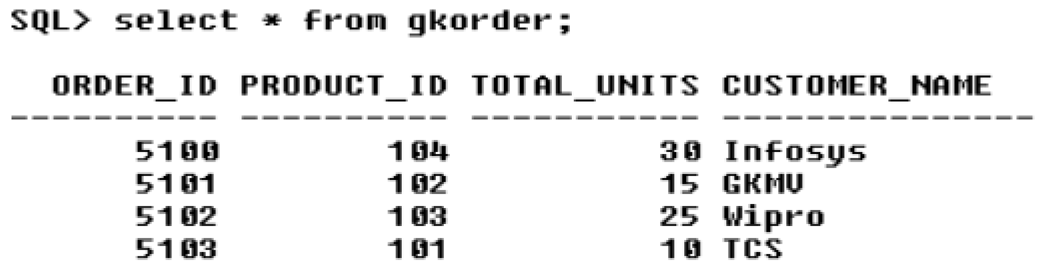
```
SQL> insert into gkproduct values(100,'Camera','Nikon',30000);

1 row created.

SQL> insert into gkproduct values(101,'Television','Onida',15000);

1 row created.

SQL> insert into gkproduct values(102,'Refrigerator','videocon',18

1 row created.

SQL> insert into gkproduct values(103,'Ipod','Apple',16000);

1 row created.

SQL> insert into gkproduct values(104,'Mobile','Samsung',8000);

1 row created.

SQL> insert into gkorder values(5100,104,30,'Infosys');

1 row created.
```



```
SQL> insert into gkorder values(5101,102,15,'GKMV');

1 row created.
SQL> insert into gkorder values(5102,103,25,'Wipro');

1 row created.

SQL>  insert into gkorder values(5103,101,10,'TCS');

1 row created.
```



```
SQL> select * from gkproduct;

PRODUCT_ID PRODUCT_NAME    SUPPLIER_NAME        PRICE
---------- --------------- --------------- ----------
       100 Camera          Nikon                30000
       101 Television      Onida                15000
       102 Refrigerator    videocon             18000
       103 Ipod            Apple                16000
       104 Mobile          Samsung               8000
```

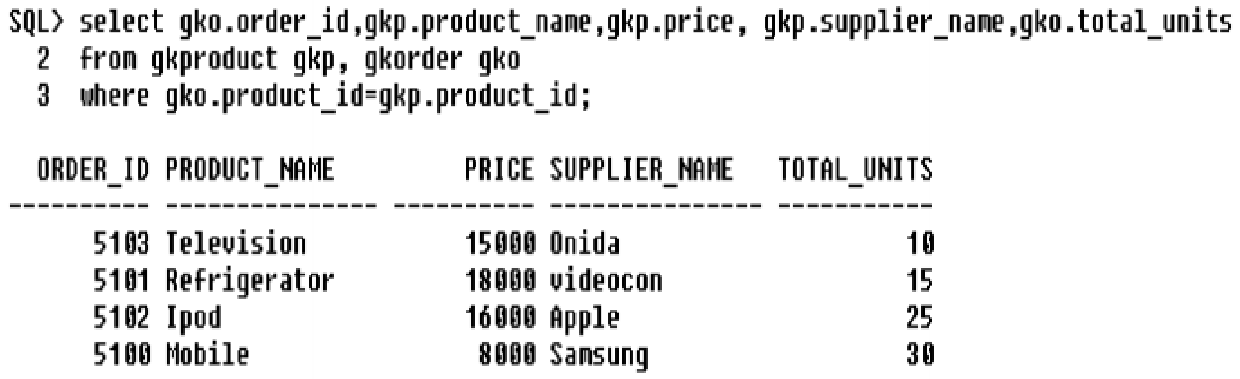


```
SQL> select * from gkorder;

  ORDER_ID PRODUCT_ID TOTAL_UNITS CUSTOMER_NAME
---------- ---------- ----------- ---------------
      5100        104          30 Infosys
      5101        102          15 GKMV
      5102        103          25 Wipro
      5103        101          10 TCS
```

INNER Join

Inner join are also known as Equi Joins. They are the most common joins used in SQL. They are known as equi joins because it uses the equal sign as the comparison operator (=). The INNER join returns all rows from both tables where there is a match. Consider the above tables (gkproduct and gkorder),

For example: If you want to display the product information for each order the query will be as given below

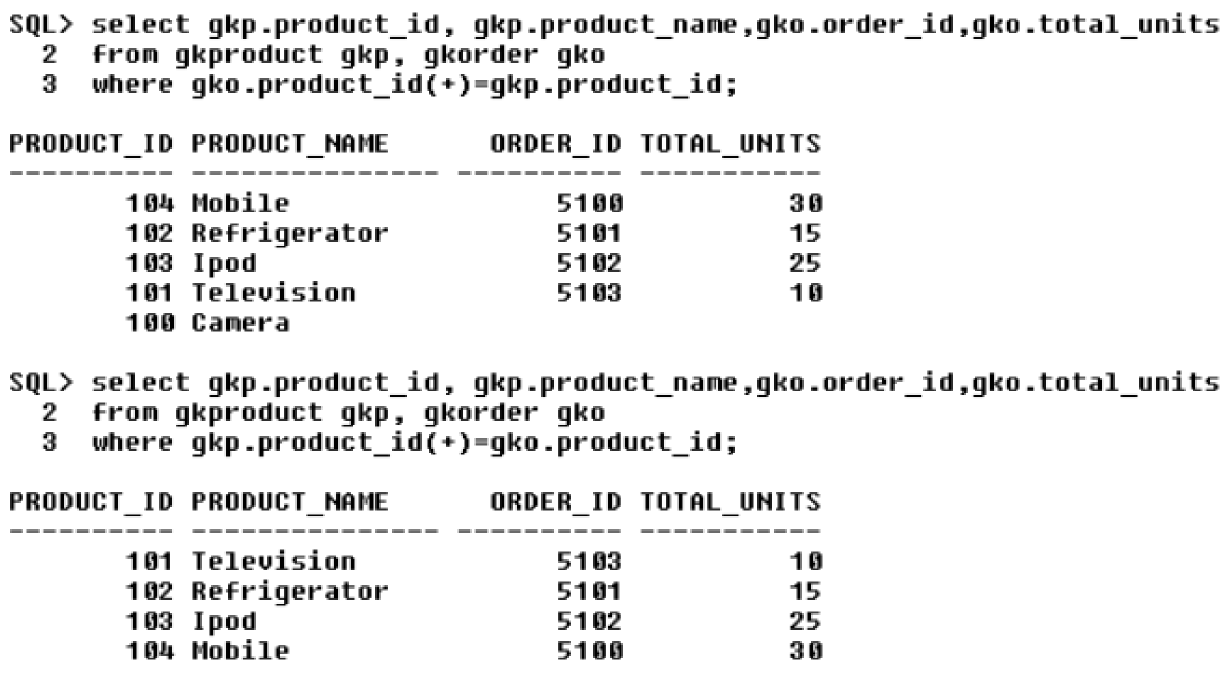


```
SQL> select gko.order_id,gkp.product_name,gkp.price, gkp.supplier_name,gko.total_units
  2  from gkproduct gkp, gkorder gko
  3  where gko.product_id=gkp.product_id;

  ORDER_ID PRODUCT_NAME         PRICE SUPPLIER_NAME   TOTAL_UNITS
---------- --------------- ---------- --------------- -----------
      5103 Television           15000 Onida                    10
      5101 Refrigerator         18000 videocon                 15
      5102 Ipod                 16000 Apple                    25
      5100 Mobile                8000 Samsung                  30
```

OUTER Join

OUTER join condition returns all rows from both tables which satisfy the join condition along with rows which do not satisfy the join condition from one of the tables. The sql outer join operator in Oracle is (+) and is used on one side of the join condition only.

For example: If you want to display all the product data along with order items data, with null values displayed for order items if a product has no order item, the sql query for outer join would be as shown below(ie First Query).



```
SQL> select gkp.product_id, gkp.product_name,gko.order_id,gko.total_units
  2  from gkproduct gkp, gkorder gko
  3  where gko.product_id(+)=gkp.product_id;

PRODUCT_ID PRODUCT_NAME      ORDER_ID TOTAL_UNITS
---------- --------------- ---------- -----------
       104 Mobile                5100          30
       102 Refrigerator          5101          15
       103 Ipod                  5102          25
       101 Television            5103          10
       100 Camera

SQL> select gkp.product_id, gkp.product_name,gko.order_id,gko.total_units
  2  from gkproduct gkp, gkorder gko
  3  where gkp.product_id(+)=gko.product_id;

PRODUCT_ID PRODUCT_NAME      ORDER_ID TOTAL_UNITS
---------- --------------- ---------- -----------
       101 Television            5103          10
       102 Refrigerator          5101          15
       103 Ipod                  5102          25
       104 Mobile                5100          30
```

NOTE: If the (+) operator is used in the left side of the join condition it is equivalent to left outer join. If used on the right side of the join condition it is equivalent to right outer join.

OUTER JOIN :

Outer Join retrieves Either, the matched rows from one table and all rows in the

other table Or, all rows in all tables (it doesn't matter whether or not there is a

match).

There are three kinds of Outer Join :

1.LEFT OUTER JOIN or LEFT JOIN

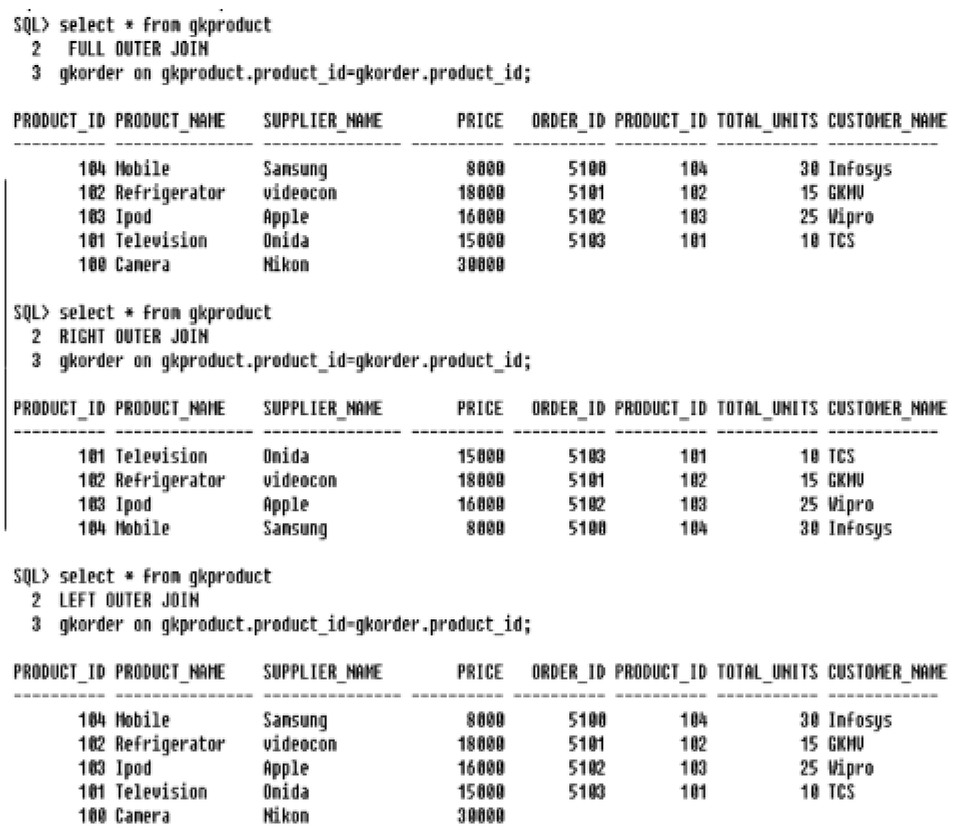
This join returns all the rows from the left table in conjunction with the matching rows from the right table. If there are no columns matching in the right table, it returns NULL values.

2. RIGHT OUTER JOIN or RIGHT JOIN

This join returns all the rows from the right table in conjunction with the matching rows from the left table. If there are no columns matching in the left table, it returns NULL values.

3.FULL OUTER JOIN or FULL JOIN

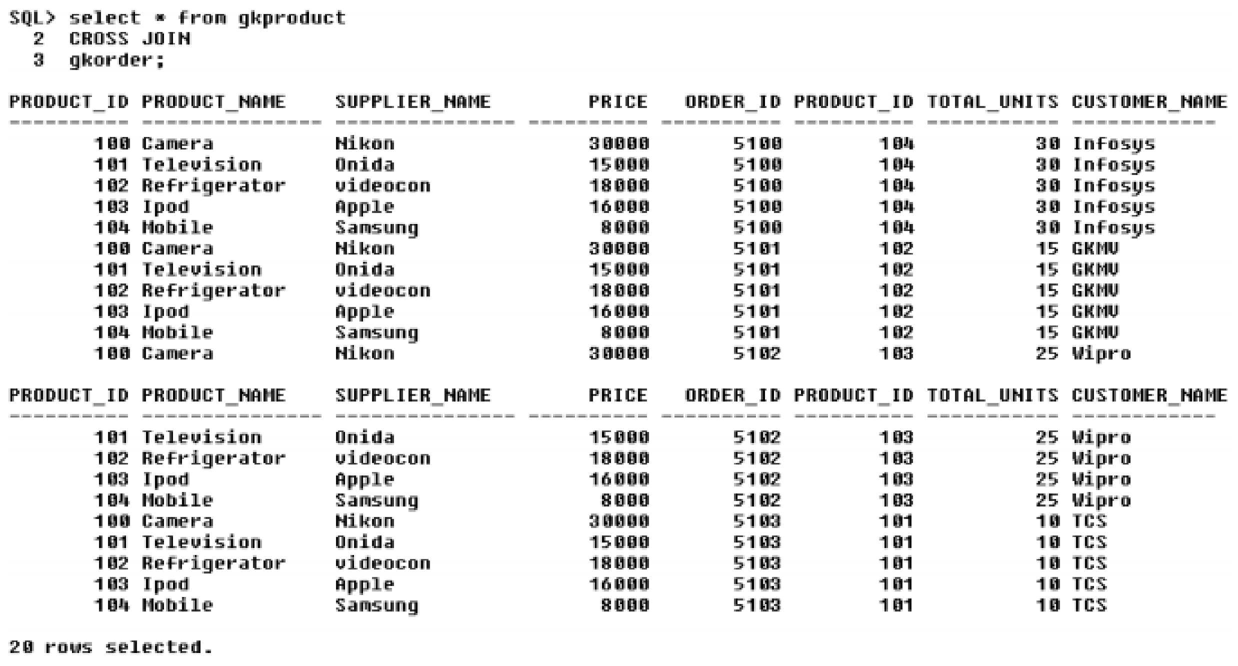
This join combines left outer join and right outer join. It returns row from either table when the conditions are met and returns null value when there is no match. In other words, OUTER JOIN is based on the fact that : ONLY the matching entries in ONE OF the tables (RIGHT or LEFT) or BOTH of the tables(FULL) SHOULD be listed.



```
SQL> select * from gkproduct
  2  FULL OUTER JOIN
  3  gkorder on gkproduct.product_id=gkorder.product_id;

PRODUCT_ID PRODUCT_NAME   SUPPLIER_NAME       PRICE   ORDER_ID PRODUCT_ID TOTAL_UNITS CUSTOMER_NAME
---------- -------------- -------------- ---------- ---------- ---------- ----------- -------------
       104 Mobile         Samsung              8000       5100        104          30 Infosys
       102 Refrigerator   videocon            18000       5101        102          15 GKMV
       103 Ipod           Apple               16000       5102        103          25 Wipro
       101 Television     Onida               15000       5103        101          10 TCS
       100 Camera         Nikon               30000

SQL> select * from gkproduct
  2  RIGHT OUTER JOIN
  3  gkorder on gkproduct.product_id=gkorder.product_id;

PRODUCT_ID PRODUCT_NAME   SUPPLIER_NAME       PRICE   ORDER_ID PRODUCT_ID TOTAL_UNITS CUSTOMER_NAME
---------- -------------- -------------- ---------- ---------- ---------- ----------- -------------
       101 Television     Onida               15000       5103        101          10 TCS
       102 Refrigerator   videocon            18000       5101        102          15 GKMV
       103 Ipod           Apple               16000       5102        103          25 Wipro
       104 Mobile         Samsung              8000       5100        104          30 Infosys

SQL> select * from gkproduct
  2  LEFT OUTER JOIN
  3  gkorder on gkproduct.product_id=gkorder.product_id;

PRODUCT_ID PRODUCT_NAME   SUPPLIER_NAME       PRICE   ORDER_ID PRODUCT_ID TOTAL_UNITS CUSTOMER_NAME
---------- -------------- -------------- ---------- ---------- ---------- ----------- -------------
       104 Mobile         Samsung              8000       5100        104          30 Infosys
       102 Refrigerator   videocon            18000       5101        102          15 GKMV
       103 Ipod           Apple               16000       5102        103          25 Wipro
       101 Television     Onida               15000       5103        101          10 TCS
       100 Camera         Nikon               30000
```

CROSS Join

It is the Cartesian product of the two tables involved. It will return a table with consists of records which combines each row from the first table with each row of the second table. The result of a CROSS JOIN will not make sense in most of the situations. Moreover, we won't need this at all (or needs the least, to be precise).



```
SQL> select * from gkproduct
  2  CROSS JOIN
  3  gkorder;

PRODUCT_ID PRODUCT_NAME    SUPPLIER_NAME        PRICE   ORDER_ID PRODUCT_ID TOTAL_UNITS CUSTOMER_NAME
---------- --------------- --------------- ---------- ---------- ---------- ----------- -----------
       100 Camera          Nikon                30000       5100        104          30 Infosys
       101 Television      Onida                15000       5100        104          30 Infosys
       102 Refrigerator    videocon             18000       5100        104          30 Infosys
       103 Ipod            Apple                16000       5100        104          30 Infosys
       104 Mobile          Samsung               8000       5100        104          30 Infosys
       100 Camera          Nikon                30000       5101        102          15 GKMV
       101 Television      Onida                15000       5101        102          15 GKMV
       102 Refrigerator    videocon             18000       5101        102          15 GKMV
       103 Ipod            Apple                16000       5101        102          15 GKMV
       104 Mobile          Samsung               8000       5101        102          15 GKMV
       100 Camera          Nikon                30000       5102        103          25 Wipro

PRODUCT_ID PRODUCT_NAME    SUPPLIER_NAME        PRICE   ORDER_ID PRODUCT_ID TOTAL_UNITS CUSTOMER_NAME
---------- --------------- --------------- ---------- ---------- ---------- ----------- -----------
       101 Television      Onida                15000       5102        103          25 Wipro
       102 Refrigerator    videocon             18000       5102        103          25 Wipro
       103 Ipod            Apple                16000       5102        103          25 Wipro
       104 Mobile          Samsung               8000       5102        103          25 Wipro
       100 Camera          Nikon                30000       5103        101          10 TCS
       101 Television      Onida                15000       5103        101          10 TCS
       102 Refrigerator    videocon             18000       5103        101          10 TCS
       103 Ipod            Apple                16000       5103        101          10 TCS
       104 Mobile          Samsung               8000       5103        101          10 TCS

20 rows selected.
```

INNER JOIN: returns rows when there is a match in both tables.

LEFT JOIN: returns all rows from the left table, even if there are no matches in the right table.

RIGHT JOIN: returns all rows from the right table, even if there are no matches in the left table.

FULL JOIN: returns rows when there is a match in one of the tables. SELF JOIN: is used to join a table to itself as if the table were two tables, temporarily renaming at least one table in the SQL statement.

CARTESIAN JOIN: returns the Cartesian product of the sets of records from the two or more joined tables.

Unit - V

Introduction to PL/SQL

PL/SQL is a combination of SQL along with the procedural features of programming languages. It was developed by Oracle Corporation in the early 90's to enhance the capabilities of SQL. PL/SQL is one of three key programming languages embedded in the Oracle Database, along with SQL itself and Java. This tutorial will give you great understanding on PL/SQL to proceed with Oracle database and other advanced RDBMS concepts.

The PL/SQL programming language was developed by Oracle Corporation in the late 1980s as procedural extension language for SQL and the Oracle relational database. Following are certain notable facts about PL/SQL −

- PL/SQL is a completely portable, high-performance transaction-processing language.
- PL/SQL provides a built-in, interpreted and OS independent programming environment.
- PL/SQL can also directly be called from the command-line **SQL*Plus interface**.
- Direct call can also be made from external programming language calls to database.
- PL/SQL's general syntax is based on that of ADA and Pascal programming language.
- Apart from Oracle, PL/SQL is available in **TimesTen in-memory database** and **IBM DB2**.

Features of PL/SQL

PL/SQL has the following features −

- PL/SQL is tightly integrated with SQL.
- It offers extensive error checking.
- It offers numerous data types.
- It offers a variety of programming structures.
- It supports structured programming through functions and procedures.
- It supports object-oriented programming.
- It supports the development of web applications and server pages.

Advantages of PL/SQL

PL/SQL has the following advantages −

- SQL is the standard database language and PL/SQL is strongly integrated with SQL. PL/SQL supports both static and dynamic SQL. Static SQL supports DML operations and transaction control from PL/SQL block. In Dynamic SQL, SQL allows embedding DDL statements in PL/SQL blocks.
- PL/SQL allows sending an entire block of statements to the database at one time. This reduces network traffic and provides high performance for the applications.

- PL/SQL gives high productivity to programmers as it can query, transform, and update data in a database.
- PL/SQL saves time on design and debugging by strong features, such as exception handling, encapsulation, data hiding, and object-oriented data types.
- Applications written in PL/SQL are fully portable.
- PL/SQL provides high security level.
- PL/SQL provides access to predefined SQL packages.
- PL/SQL provides support for Object-Oriented Programming.
- PL/SQL provides support for developing Web Applications and Server Pages.

PL/SQL Block Structure

In this chapter, we will discuss the Basic Syntax of PL/SQL which is a **block-structured** language; this means that the PL/SQL programs are divided and written in logical blocks of code. Each block consists of three sub-parts –

S.No	Sections & Description
1	**Declarations** This section starts with the keyword **DECLARE**. It is an optional section and defines all variables, cursors, subprograms, and other elements to be used in the program.
2	**Executable Commands** This section is enclosed between the keywords **BEGIN** and **END** and it is a mandatory section. It consists of the executable PL/SQL statements of the program. It should have at least one executable line of code, which may be just a **NULL command** to indicate that nothing should be executed.
3	**Exception Handling** This section starts with the keyword **EXCEPTION**. This optional section contains **exception(s)** that handle errors in the program.

Every PL/SQL statement ends with a semicolon (;). PL/SQL blocks can be nested within other PL/SQL blocks using **BEGIN** and **END**. Following is the basic structure of a PL/SQL block –

```
DECLARE
  <declarations section>
BEGIN
```

```
  <executable command(s)>
EXCEPTION
  <exception handling>
END;
```

The 'Hello World' Example

```
DECLARE
   message  varchar2(20):= 'Hello, World!';
BEGIN
   dbms_output.put_line(message);
END;
/
```

The **end;** line signals the end of the PL/SQL block. To run the code from the SQL command line, you may need to type / at the beginning of the first blank line after the last line of the code. When the above code is executed at the SQL prompt, it produces the following result −

```
Hello World

PL/SQL procedure successfully completed.
```

The PL/SQL Identifiers

PL/SQL identifiers are constants, variables, exceptions, procedures, cursors, and reserved words. The identifiers consist of a letter optionally followed by more letters, numerals, dollar signs, underscores, and number signs and should not exceed 30 characters.

By default, **identifiers are not case-sensitive**. So you can use **integer** or **INTEGER** to represent a numeric value. You cannot use a reserved keyword as an identifier.

The PL/SQL Delimiters

A delimiter is a symbol with a special meaning. Following is the list of delimiters in PL/SQL −

Delimiter	Description
+, -, *, /	Addition, subtraction/negation, multiplication, division
%	Attribute indicator
'	Character string delimiter
.	Component selector

(,)	Expression or list delimiter
:	Host variable indicator
,	Item separator
"	Quoted identifier delimiter
=	Relational operator
@	Remote access indicator
;	Statement terminator
:=	Assignment operator
=>	Association operator
||	Concatenation operator
**	Exponentiation operator
<<, >>	Label delimiter (begin and end)
/*, */	Multi-line comment delimiter (begin and end)
--	Single-line comment indicator
..	Range operator

<, >, <=, >=	Relational operators
<>, '=, ~=, ^=	Different versions of NOT EQUAL

The PL/SQL Comments

Program comments are explanatory statements that can be included in the PL/SQL code that you write and helps anyone reading its source code. All programming languages allow some form of comments.

The PL/SQL supports single-line and multi-line comments. All characters available inside any comment are ignored by the PL/SQL compiler. The PL/SQL single-line comments start with the delimiter -- (double hyphen) and multi-line comments are enclosed by /* and */.

```
DECLARE
   -- variable declaration
   message  varchar2(20):= 'Hello, World!';
BEGIN
   /*
   *  PL/SQL executable statement(s)
   */
   dbms_output.put_line(message);
END;
/
```

When the above code is executed at the SQL prompt, it produces the following result –

```
Hello World

PL/SQL procedure successfully completed.
```

PL/SQL Program Units

A PL/SQL unit is any one of the following –

- PL/SQL block
- Function
- Package
- Package body
- Procedure
- Trigger
- Type
- Type body

Data Types

we will discuss the Data Types in PL/SQL. The PL/SQL variables, constants and parameters must have a valid data type, which specifies a storage format, constraints, and a valid range of values. We will focus on the **SCALAR** and the **LOB** data types in this chapter. The other two data types will be covered in other chapters.

S.No	Category & Description
1	**Scalar** Single values with no internal components, such as a **NUMBER, DATE,** or **BOOLEAN**.
2	**Large Object (LOB)** Pointers to large objects that are stored separately from other data items, such as text, graphic images, video clips, and sound waveforms.
3	**Composite** Data items that have internal components that can be accessed individually. For example, collections and records.
4	**Reference** Pointers to other data items.

PL/SQL Scalar Data Types and Subtypes

PL/SQL Scalar Data Types and Subtypes come under the following categories −

S.No	Date Type & Description
1	**Numeric** Numeric values on which arithmetic operations are performed.
2	**Character** Alphanumeric values that represent single characters or strings of characters.

3	**Boolean** Logical values on which logical operations are performed.
4	**Datetime** Dates and times.

PL/SQL provides subtypes of data types. For example, the data type NUMBER has a subtype called INTEGER. You can use the subtypes in your PL/SQL program to make the data types compatible with data types in other programs while embedding the PL/SQL code in another program, such as a Java program.

PL/SQL Numeric Data Types and Subtypes

Following table lists out the PL/SQL pre-defined numeric data types and their sub-types −

S.No	Data Type & Description
1	**PLS_INTEGER** Signed integer in range -2,147,483,648 through 2,147,483,647, represented in 32 bits
2	**BINARY_INTEGER** Signed integer in range -2,147,483,648 through 2,147,483,647, represented in 32 bits
3	**BINARY_FLOAT** Single-precision IEEE 754-format floating-point number
4	**BINARY_DOUBLE** Double-precision IEEE 754-format floating-point number
5	**NUMBER(prec, scale)** Fixed-point or floating-point number with absolute value in range 1E-130 to (but not including) 1.0E126. A NUMBER variable can also represent 0

6	**DEC(prec, scale)** ANSI specific fixed-point type with maximum precision of 38 decimal digits
7	**DECIMAL(prec, scale)** IBM specific fixed-point type with maximum precision of 38 decimal digits
8	**NUMERIC(pre, secale)** Floating type with maximum precision of 38 decimal digits
9	**DOUBLE PRECISION** ANSI specific floating-point type with maximum precision of 126 binary digits (approximately 38 decimal digits)
10	**FLOAT** ANSI and IBM specific floating-point type with maximum precision of 126 binary digits (approximately 38 decimal digits)
11	**INT** ANSI specific integer type with maximum precision of 38 decimal digits
12	**INTEGER** ANSI and IBM specific integer type with maximum precision of 38 decimal digits
13	**SMALLINT** ANSI and IBM specific integer type with maximum precision of 38 decimal digits
14	**REAL** Floating-point type with maximum precision of 63 binary digits (approximately 18 decimal digits)

Following is a valid declaration −

```
DECLARE
   num1 INTEGER;
   num2 REAL;
   num3 DOUBLE PRECISION;
BEGIN
```

```
   null;
END;
/
```

When the above code is compiled and executed, it produces the following result −

```
PL/SQL procedure successfully completed
```

PL/SQL Character Data Types and Subtypes

Following is the detail of PL/SQL pre-defined character data types and their sub-types −

S.No	Data Type & Description
1	**CHAR** Fixed-length character string with maximum size of 32,767 bytes
2	**VARCHAR2** Variable-length character string with maximum size of 32,767 bytes
3	**RAW** Variable-length binary or byte string with maximum size of 32,767 bytes, not interpreted by PL/SQL
4	**NCHAR** Fixed-length national character string with maximum size of 32,767 bytes
5	**NVARCHAR2** Variable-length national character string with maximum size of 32,767 bytes
6	**LONG** Variable-length character string with maximum size of 32,760 bytes
7	**LONG RAW** Variable-length binary or byte string with maximum size of 32,760 bytes, not interpreted by PL/SQL

8	**ROWID** Physical row identifier, the address of a row in an ordinary table
9	**UROWID** Universal row identifier (physical, logical, or foreign row identifier)

PL/SQL Boolean Data Types

The **BOOLEAN** data type stores logical values that are used in logical operations. The logical values are the Boolean values **TRUE** and **FALSE** and the value **NULL**.

However, SQL has no data type equivalent to BOOLEAN. Therefore, Boolean values cannot be used in −

- SQL statements
- Built-in SQL functions (such as **TO_CHAR**)
- PL/SQL functions invoked from SQL statements

PL/SQL Datetime and Interval Types

The **DATE** datatype is used to store fixed-length datetimes, which include the time of day in seconds since midnight. Valid dates range from January 1, 4712 BC to December 31, 9999 AD.

The default date format is set by the Oracle initialization parameter NLS_DATE_FORMAT. For example, the default might be 'DD-MON-YY', which includes a two-digit number for the day of the month, an abbreviation of the month name, and the last two digits of the year. For example, 01-OCT-12.

Each DATE includes the century, year, month, day, hour, minute, and second. The following table shows the valid values for each field −

Field Name	**Valid Datetime Values**	**Valid Interval Values**
YEAR	-4712 to 9999 (excluding year 0)	Any nonzero integer
MONTH	01 to 12	0 to 11
DAY	01 to 31 (limited by the values of MONTH and YEAR, according to the rules of the calendar for the locale)	Any nonzero integer

HOUR	00 to 23	0 to 23
MINUTE	00 to 59	0 to 59
SECOND	00 to 59.9(n), where 9(n) is the precision of time fractional seconds	0 to 59.9(n), where 9(n) is the precision of interval fractional seconds
TIMEZONE_HOUR	-12 to 14 (range accommodates daylight savings time changes)	Not applicable
TIMEZONE_MINUTE	00 to 59	Not applicable
TIMEZONE_REGION	Found in the dynamic performance view V$TIMEZONE_NAMES	Not applicable
TIMEZONE_ABBR	Found in the dynamic performance view V$TIMEZONE_NAMES	Not applicable

PL/SQL Large Object (LOB) Data Types

Large Object (LOB) data types refer to large data items such as text, graphic images, video clips, and sound waveforms. LOB data types allow efficient, random, piecewise access to this data. Following are the predefined PL/SQL LOB data types −

Data Type	**Description**	**Size**
BFILE	Used to store large binary objects in operating system files outside the database.	System-dependent. Cannot exceed 4 gigabytes (GB).
BLOB	Used to store large binary objects in the database.	8 to 128 terabytes (TB)

CLOB	Used to store large blocks of character data in the database.	8 to 128 TB
NCLOB	Used to store large blocks of NCHAR data in the database.	

NULLs in PL/SQL

PL/SQL NULL values represent **missing** or **unknown data** and they are not an integer, a character, or any other specific data type. Note that **NULL** is not the same as an empty data string or the null character value **'\0'**. A null can be assigned but it cannot be equated with anything, including itself.

PL/SQL Variables

we will discuss Variables in Pl/SQL. A variable is nothing but a name given to a storage area that our programs can manipulate. Each variable in PL/SQL has a specific data type, which determines the size and the layout of the variable's memory; the range of values that can be stored within that memory and the set of operations that can be applied to the variable.

The name of a PL/SQL variable consists of a letter optionally followed by more letters, numerals, dollar signs, underscores, and number signs and should not exceed 30 characters. By default, variable names are not case-sensitive. You cannot use a reserved PL/SQL keyword as a variable name.

PL/SQL programming language allows to define various types of variables, such as date time data types, records, collections, etc. which we will cover in subsequent chapters. For this chapter, let us study only basic variable types.

Variable Declaration in PL/SQL

PL/SQL variables must be declared in the declaration section or in a package as a global variable. When you declare a variable, PL/SQL allocates memory for the variable's value and the storage location is identified by the variable name.

The syntax for declaring a variable is −

```
variable_name [CONSTANT] datatype [NOT NULL] [:= | DEFAULT initial_value]
```

Where, *variable_name* is a valid identifier in PL/SQL, *datatype* must be a valid PL/SQL data type or any user defined data type which we already have discussed in the last chapter. Some valid variable declarations along with their definition are shown below −

```
sales number(10, 2);
pi CONSTANT double precision := 3.1415;
name varchar2(25);
address varchar2(100);
```

When you provide a size, scale or precision limit with the data type, it is called a **constrained declaration**. Constrained declarations require less memory than unconstrained declarations. For example −

```
sales number(10, 2);
name varchar2(25);
address varchar2(100);
```

Initializing Variables in PL/SQL

Whenever you declare a variable, PL/SQL assigns it a default value of NULL. If you want to initialize a variable with a value other than the NULL value, you can do so during the declaration, using either of the following −

- The **DEFAULT** keyword
- The **assignment** operator

For example −

```
counter binary_integer := 0;
greetings varchar2(20) DEFAULT 'Have a Good Day';
```

You can also specify that a variable should not have a **NULL** value using the **NOT NULL** constraint. If you use the NOT NULL constraint, you must explicitly assign an initial value for that variable.

It is a good programming practice to initialize variables properly otherwise, sometimes programs would produce unexpected results. Try the following example which makes use of various types of variables −

```
DECLARE
  a integer := 10;
  b integer := 20;
  c integer;
  f real;
BEGIN
  c := a + b;
  dbms_output.put_line('Value of c: ' || c);
  f := 70.0/3.0;
  dbms_output.put_line('Value of f: ' || f);
END;
/
```

When the above code is executed, it produces the following result −

```
Value of c: 30
Value of f: 23.333333333333333333

PL/SQL procedure successfully completed.
```

Variable Scope in PL/SQL

PL/SQL allows the nesting of blocks, i.e., each program block may contain another inner block. If a variable is declared within an inner block, it is not accessible to the outer block. However, if a variable is declared and accessible to an outer block, it is also accessible to all nested inner blocks. There are two types of variable scope −

- **Local variables** − Variables declared in an inner block and not accessible to outer blocks.
- **Global variables** − Variables declared in the outermost block or a package.

Following example shows the usage of **Local** and **Global** variables in its simple form −

```
DECLARE
  -- Global variables
  num1 number := 95;
  num2 number := 85;
BEGIN
  dbms_output.put_line('Outer Variable num1: ' || num1);
  dbms_output.put_line('Outer Variable num2: ' || num2);
  DECLARE
    -- Local variables
    num1 number := 195;
    num2 number := 185;
  BEGIN
    dbms_output.put_line('Inner Variable num1: ' || num1);
    dbms_output.put_line('Inner Variable num2: ' || num2);
  END;
END;
/
```

When the above code is executed, it produces the following result −

```
Outer Variable num1: 95
Outer Variable num2: 85
Inner Variable num1: 195
Inner Variable num2: 185

PL/SQL procedure successfully completed.
```

Assigning SQL Query Results to PL/SQL Variables

You can use the **SELECT INTO** statement of SQL to assign values to PL/SQL variables. For each item in the **SELECT list**, there must be a corresponding, type-compatible variable in the **INTO list**. The following example illustrates the concept. Let us create a table named CUSTOMERS −

(**For SQL statements, please refer to the** SQL tutorial)

```
CREATE TABLE CUSTOMERS(
  ID   INT NOT NULL,
  NAME VARCHAR (20) NOT NULL,
  AGE INT NOT NULL,
  ADDRESS CHAR (25),
  SALARY   DECIMAL (18, 2),
  PRIMARY KEY (ID)
);

Table Created
```

Let us now insert some values in the table −

```
INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (1, 'Ramesh', 32, 'Ahmedabad', 2000.00 );

INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (2, 'Khilan', 25, 'Delhi', 1500.00 );

INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (3, 'kaushik', 23, 'Kota', 2000.00 );

INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (4, 'Chaitali', 25, 'Mumbai', 6500.00 );

INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (5, 'Hardik', 27, 'Bhopal', 8500.00 );

INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (6, 'Komal', 22, 'MP', 4500.00 );
```

The following program assigns values from the above table to PL/SQL variables using the **SELECT INTO clause** of SQL −

```
DECLARE
   c_id customers.id%type := 1;
   c_name  customers.name%type;
   c_addr customers.address%type;
   c_sal  customers.salary%type;
BEGIN
   SELECT name, address, salary INTO c_name, c_addr, c_sal
   FROM customers
   WHERE id = c_id;
   dbms_output.put_line
   ('Customer ' ||c_name || ' from ' || c_addr || ' earns ' || c_sal);
END;
/
```

When the above code is executed, it produces the following result −

```
Customer Ramesh from Ahmedabad earns 2000

PL/SQL procedure completed successfully
```

PL/SQL Constants and Literals

we will discuss **constants** and **literals** in PL/SQL. A constant holds a value that once declared, does not change in the program. A constant declaration specifies its name, data type, and value, and allocates storage for it. The declaration can also impose the **NOT NULL constraint**.

Declaring a Constant

A constant is declared using the **CONSTANT** keyword. It requires an initial value and does not allow that value to be changed. For example −

```
PI CONSTANT NUMBER := 3.141592654;
DECLARE
  -- constant declaration
  pi constant number := 3.141592654;
  -- other declarations
  radius number(5,2);
  dia number(5,2);
  circumference number(7, 2);
  area number (10, 2);
BEGIN
  -- processing
  radius := 9.5;
  dia := radius * 2;
  circumference := 2.0 * pi * radius;
  area := pi * radius * radius;
  -- output
  dbms_output.put_line('Radius: ' || radius);
  dbms_output.put_line('Diameter: ' || dia);
  dbms_output.put_line('Circumference: ' || circumference);
  dbms_output.put_line('Area: ' || area);
END;
/
```

When the above code is executed at the SQL prompt, it produces the following result −

```
Radius: 9.5
Diameter: 19
Circumference: 59.69
Area: 283.53

Pl/SQL procedure successfully completed.
```

The PL/SQL Literals

A literal is an explicit numeric, character, string, or Boolean value not represented by an identifier. For example, TRUE, 786, NULL, 'tutorialspoint' are all literals of type Boolean, number, or string. PL/SQL, literals are case-sensitive. PL/SQL supports the following kinds of literals –

- Numeric Literals
- Character Literals
- String Literals
- BOOLEAN Literals
- Date and Time Literals

The following table provides examples from all these categories of literal values.

S.No	Literal Type & Example
1	**Numeric Literals** 050 78 -14 0 +32767 6.6667 0.0 -12.0 3.14159 +7800.00 6E5 1.0E-8 3.14159e0 -1E38 -9.5e-3
2	**Character Literals** 'A' '%' '9' ' ' 'z' '('
3	**String Literals** 'Hello, world!' 'Tutorials Point' '19-NOV-12'
4	**BOOLEAN Literals** TRUE, FALSE, and NULL.

5	**Date and Time Literals**
	DATE '1978-12-25';
	TIMESTAMP '2012-10-29 12:01:01';

To embed single quotes within a string literal, place two single quotes next to each other as shown in the following program –

```
DECLARE
  message  varchar2(30):= 'That''s tutorialspoint.com!';
BEGIN
  dbms_output.put_line(message);
END;
/
```

When the above code is executed at the SQL prompt, it produces the following result –

```
That's tutorialspoint.com!

PL/SQL procedure successfully completed.
```

PL/SQL Operators

we will discuss operators in PL/SQL. An operator is a symbol that tells the compiler to perform specific mathematical or logical manipulation. PL/SQL language is rich in built-in operators and provides the following types of operators –

- Arithmetic operators
- Relational operators
- Comparison operators
- Logical operators
- String operators

Here, we will understand the arithmetic, relational, comparison and logical operators one by one. The String operators will be discussed in a later chapter – **PL/SQL - Strings**.

Arithmetic Operators

Following table shows all the arithmetic operators supported by PL/SQL. Let us assume **variable A** holds 10 and **variable B** holds 5, then –

Show Examples

Operator	Description	Example
+	Adds two operands	A + B will give 15
-	Subtracts second operand from the first	A - B will give 5

*	Multiplies both operands	A * B will give 50
/	Divides numerator by de-numerator	A / B will give 2
**	Exponentiation operator, raises one operand to the power of other	A ** B will give 100000

Relational Operators

Relational operators compare two expressions or values and return a Boolean result. Following table shows all the relational operators supported by PL/SQL. Let us assume **variable A** holds 10 and **variable B** holds 20, then –

Show Examples

Operator	Description	Example
=	Checks if the values of two operands are equal or not, if yes then condition becomes true.	(A = B) is not true.
!= <> ~=	Checks if the values of two operands are equal or not, if values are not equal then condition becomes true.	(A != B) is true.
>	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.	(A > B) is not true.
<	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.	(A < B) is true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.	(A >= B) is not true.
<=	Checks if the value of left operand is less than or equal to the value	(A <= B) is

	of right operand, if yes then condition becomes true.	true

Comparison Operators

Comparison operators are used for comparing one expression to another. The result is always either **TRUE, FALSE** or **NULL**.

Show Examples

Operator	**Description**	**Example**
LIKE	The LIKE operator compares a character, string, or CLOB value to a pattern and returns TRUE if the value matches the pattern and FALSE if it does not.	If 'Zara Ali' like 'Z% A_i' returns a Boolean true, whereas, 'Nuha Ali' like 'Z% A_i' returns a Boolean false.
BETWEEN	The BETWEEN operator tests whether a value lies in a specified range. x BETWEEN a AND b means that x >= a and x <= b.	If x = 10 then, x between 5 and 20 returns true, x between 5 and 10 returns true, but x between 11 and 20 returns false.
IN	The IN operator tests set membership. x IN (set) means that x is equal to any member of set.	If x = 'm' then, x in ('a', 'b', 'c') returns Boolean false but x in ('m', 'n', 'o') returns Boolean true.
IS NULL	The IS NULL operator returns the BOOLEAN value TRUE if its operand is NULL or FALSE if it is not NULL. Comparisons involving NULL values always yield NULL.	If x = 'm', then 'x is null' returns Boolean false.

Logical Operators

Following table shows the Logical operators supported by PL/SQL. All these operators work on Boolean operands and produce Boolean results. Let us assume **variable A** holds true and **variable B** holds false, then –

Show Examples

Operator	Description	Examples
and	Called the logical AND operator. If both the operands are true then condition becomes true.	(A and B) is false.
or	Called the logical OR Operator. If any of the two operands is true then condition becomes true.	(A or B) is true.
not	Called the logical NOT Operator. Used to reverse the logical state of its operand. If a condition is true then Logical NOT operator will make it false.	not (A and B) is true.

PL/SQL Operator Precedence

Operator precedence determines the grouping of terms in an expression. This affects how an expression is evaluated. Certain operators have higher precedence than others; for example, the multiplication operator has higher precedence than the addition operator.

For example, **x = 7 + 3 * 2**; here, **x** is assigned **13**, not 20 because operator * has higher precedence than +, so it first gets multiplied with **3*2** and then adds into **7**.

Here, operators with the highest precedence appear at the top of the table, those with the lowest appear at the bottom. Within an expression, higher precedence operators will be evaluated first.

The precedence of operators goes as follows: =, <, >, <=, >=, <>, !=, ~=, ^=, IS NULL, LIKE, BETWEEN, IN.

Show Examples

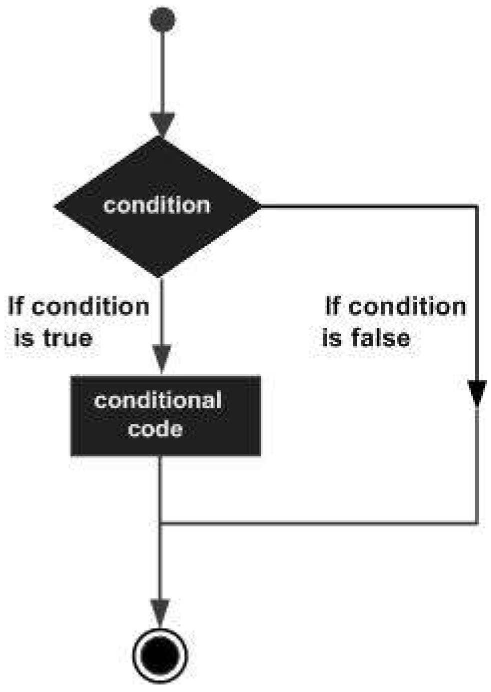
Operator	Operation
**	exponentiation
+, -	identity, negation
*, /	multiplication, division
+, -, ||	addition, subtraction, concatenation

comparison	
NOT	logical negation
AND	conjunction
OR	inclusion

PL/SQL conditions

we will discuss conditions in PL/SQL. Decision-making structures require that the programmer specify one or more conditions to be evaluated or tested by the program, along with a statement or statements to be executed if the condition is determined to be true, and optionally, other statements to be executed if the condition is determined to be false.

Following is the general form of a typical conditional (i.e., decision making) structure found in most of the programming languages –



PL/SQL programming language provides following types of decision-making statements. Click the following links to check their detail.

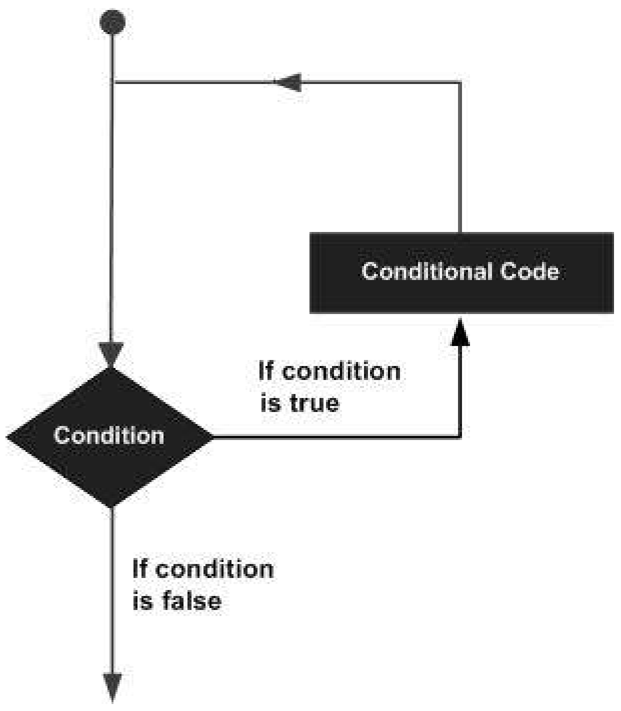
S.No	Statement & Description
1	IF - THEN statement The **IF statement** associates a condition with a sequence of statements enclosed by the keywords **THEN** and **END IF**. If the condition is true, the statements get executed and if the condition is false or NULL then the IF statement does nothing.
2	IF-THEN-ELSE statement **IF statement** adds the keyword **ELSE** followed by an alternative sequence of statement. If the condition is false or NULL, then only the alternative sequence of statements get executed. It ensures that either of the sequence of statements is executed.
3	IF-THEN-ELSIF statement It allows you to choose between several alternatives.
4	Case statement Like the IF statement, the **CASE statement** selects one sequence of statements to execute. However, to select the sequence, the CASE statement uses a selector rather than multiple Boolean expressions. A selector is an expression whose value is used to select one of several alternatives.
5	Searched CASE statement The searched CASE statement **has no selector**, and it's WHEN clauses contain search conditions that yield Boolean values.
6	nested IF-THEN-ELSE You can use one **IF-THEN** or **IF-THEN-ELSIF** statement inside another **IF-THEN** or **IF-THEN-ELSIF** statement(s).

PL/SQL Loops

we will discuss Loops in PL/SQL. There may be a situation when you need to execute a block of code several number of times. In general, statements are executed sequentially: The first statement in a function is executed first, followed by the second, and so on.

Programming languages provide various control structures that allow for more complicated execution paths.

A loop statement allows us to execute a statement or group of statements multiple times and following is the general form of a loop statement in most of the programming languages −



PL/SQL provides the following types of loop to handle the looping requirements. Click the following links to check their detail.

S.No	Loop Type & Description
1	PL/SQL Basic LOOP In this loop structure, sequence of statements is enclosed between the LOOP and the END LOOP statements. At each iteration, the sequence of statements is executed and then control resumes at the top of the loop.
2	PL/SQL WHILE LOOP Repeats a statement or group of statements while a given condition is true. It tests the

	condition before executing the loop body.
3	PL/SQL FOR LOOP Execute a sequence of statements multiple times and abbreviates the code that manages the loop variable.
4	Nested loops in PL/SQL You can use one or more loop inside any another basic loop, while, or for loop.

Labeling a PL/SQL Loop

PL/SQL loops can be labeled. The label should be enclosed by double angle brackets (<< and >>) and appear at the beginning of the LOOP statement. The label name can also appear at the end of the LOOP statement. You may use the label in the EXIT statement to exit from the loop.

The following program illustrates the concept –

```
DECLARE
  i number(1);
  j number(1);
BEGIN
  << outer_loop >>
  FOR i IN 1..3 LOOP
    << inner_loop >>
    FOR j IN 1..3 LOOP
      dbms_output.put_line('i is: '|| i || ' and j is: ' || j);
    END loop inner_loop;
  END loop outer_loop;
END;
/
```

When the above code is executed at the SQL prompt, it produces the following result –

```
i is: 1 and j is: 1
i is: 1 and j is: 2
i is: 1 and j is: 3
i is: 2 and j is: 1
i is: 2 and j is: 2
i is: 2 and j is: 3
i is: 3 and j is: 1
i is: 3 and j is: 2
i is: 3 and j is: 3

PL/SQL procedure successfully completed.
```

The Loop Control Statements

Loop control statements change execution from its normal sequence. When execution leaves a scope, all automatic objects that were created in that scope are destroyed.

PL/SQL supports the following control statements. Labeling loops also help in taking the control outside a loop. Click the following links to check their details.

S.No	Control Statement & Description
1	EXIT statement The Exit statement completes the loop and control passes to the statement immediately after the END LOOP.
2	CONTINUE statement Causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating.
3	GOTO statement Transfers control to the labeled statement. Though it is not advised to use the GOTO statement in your program.

PL/SQL Strings

The string in PL/SQL is actually a sequence of characters with an optional size specification. The characters could be numeric, letters, blank, special characters or a combination of all. PL/SQL offers three kinds of strings −

- **Fixed-length strings** − In such strings, programmers specify the length while declaring the string. The string is right-padded with spaces to the length so specified.
- **Variable-length strings** − In such strings, a maximum length up to 32,767, for the string is specified and no padding takes place.
- **Character large objects (CLOBs)** − These are variable-length strings that can be up to 128 terabytes.

PL/SQL strings could be either variables or literals. A string literal is enclosed within quotation marks. For example,

'This is a string literal.' Or 'hello world'

To include a single quote inside a string literal, you need to type two single quotes next to one another. For example,

```
'this isn''t what it looks like'
```

Declaring String Variables

Oracle database provides numerous string datatypes, such as CHAR, NCHAR, VARCHAR2, NVARCHAR2, CLOB, and NCLOB. The datatypes prefixed with an **'N'** are **'national character set'** datatypes, that store Unicode character data.

If you need to declare a variable-length string, you must provide the maximum length of that string. For example, the VARCHAR2 data type. The following example illustrates declaring and using some string variables −

```
DECLARE
  name varchar2(20);
  company varchar2(30);
  introduction clob;
  choice char(1);
BEGIN
  name := 'John Smith';
  company := 'Infotech';
  introduction := ' Hello! I''m John Smith from Infotech.';
  choice := 'y';
  IF choice = 'y' THEN
    dbms_output.put_line(name);
    dbms_output.put_line(company);
    dbms_output.put_line(introduction);
  END IF;
END;
/
```

When the above code is executed at the SQL prompt, it produces the following result −

```
John Smith
Infotech
Hello! I'm John Smith from Infotech.

PL/SQL procedure successfully completed
```

To declare a fixed-length string, use the CHAR datatype. Here you do not have to specify a maximum length for a fixed-length variable. If you leave off the length constraint, Oracle Database automatically uses a maximum length required. The following two declarations are identical −

```
red_flag CHAR(1) := 'Y';
 red_flag CHAR   := 'Y';
```

PL/SQL String Functions and Operators

PL/SQL offers the concatenation operator **(||)** for joining two strings. The following table provides the string functions provided by PL/SQL –

S.No	Function & Purpose
1	**ASCII(x);** Returns the ASCII value of the character x.
2	**CHR(x);** Returns the character with the ASCII value of x.
3	**CONCAT(x, y);** Concatenates the strings x and y and returns the appended string.
4	**INITCAP(x);** Converts the initial letter of each word in x to uppercase and returns that string.
5	**INSTR(x, find_string [, start] [, occurrence]);** Searches for **find_string** in x and returns the position at which it occurs.
6	**INSTRB(x);** Returns the location of a string within another string, but returns the value in bytes.
7	**LENGTH(x);** Returns the number of characters in x.
8	**LENGTHB(x);** Returns the length of a character string in bytes for single byte character set.
9	**LOWER(x);** Converts the letters in x to lowercase and returns that string.

10	**LPAD(x, width [, pad_string]) ;** Pads **x** with spaces to the left, to bring the total length of the string up to width characters.
11	**LTRIM(x [, trim_string]);** Trims characters from the left of **x**.
12	**NANVL(x, value);** Returns value if x matches the NaN special value (not a number), otherwise **x** is returned.
13	**NLS_INITCAP(x);** Same as the INITCAP function except that it can use a different sort method as specified by NLSSORT.
14	**NLS_LOWER(x) ;** Same as the LOWER function except that it can use a different sort method as specified by NLSSORT.
15	**NLS_UPPER(x);** Same as the UPPER function except that it can use a different sort method as specified by NLSSORT.
16	**NLSSORT(x);** Changes the method of sorting the characters. Must be specified before any NLS function; otherwise, the default sort will be used.
17	**NVL(x, value);** Returns value if **x** is null; otherwise, x is returned.
18	**NVL2(x, value1, value2);** Returns value1 if x is not null; if x is null, value2 is returned.
	REPLACE(x, search_string, replace_string);

19	Searches **x** for search_string and replaces it with replace_string.
20	**RPAD(x, width [, pad_string]);** Pads **x** to the right.
21	**RTRIM(x [, trim_string]);** Trims **x** from the right.
22	**SOUNDEX(x) ;** Returns a string containing the phonetic representation of **x**.
23	**SUBSTR(x, start [, length]);** Returns a substring of **x** that begins at the position specified by start. An optional length for the substring may be supplied.
24	**SUBSTRB(x);** Same as SUBSTR except that the parameters are expressed in bytes instead of characters for the single-byte character systems.
25	**TRIM([trim_char FROM) x);** Trims characters from the left and right of **x**.
26	**UPPER(x);** Converts the letters in x to uppercase and returns that string.

Let us now work out on a few examples to understand the concept −

Example 1

```
DECLARE
  greetings varchar2(11) := 'hello world';
BEGIN
  dbms_output.put_line(UPPER(greetings));
  dbms_output.put_line(LOWER(greetings));

  dbms_output.put_line(INITCAP(greetings));

  /* retrieve the first character in the string */
  dbms_output.put_line ( SUBSTR (greetings, 1, 1));

  /* retrieve the last character in the string */
  dbms_output.put_line ( SUBSTR (greetings, -1, 1));

  /* retrieve five characters,
    starting from the seventh position. */
  dbms_output.put_line ( SUBSTR (greetings, 7, 5));

  /* retrieve the remainder of the string,
    starting from the second position. */
  dbms_output.put_line ( SUBSTR (greetings, 2));

  /* find the location of the first "e" */
  dbms_output.put_line ( INSTR (greetings, 'e'));
END;
/
```

When the above code is executed at the SQL prompt, it produces the following result −

```
HELLO WORLD
hello world
Hello World
h
d
World
ello World
2

PL/SQL procedure successfully completed.
```

Example 2

```
DECLARE
  greetings varchar2(30) := '......Hello World.....';
BEGIN
  dbms_output.put_line(RTRIM(greetings,'.'));
  dbms_output.put_line(LTRIM(greetings, '.'));
  dbms_output.put_line(TRIM( '.' from greetings));
END;
/
```

When the above code is executed at the SQL prompt, it produces the following result −

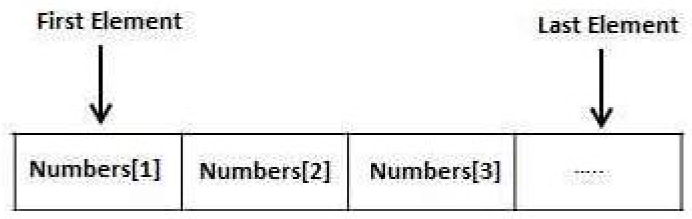
```
......Hello World
```

```
Hello World.....
Hello World

PL/SQL procedure successfully completed.
```

PL/SQL Arrays

we will discuss arrays in PL/SQL. The PL/SQL programming language provides a data structure called the **VARRAY**, which can store a fixed-size sequential collection of elements of the same type. A varray is used to store an ordered collection of data, however it is often better to think of an array as a collection of variables of the same type.

All varrays consist of contiguous memory locations. The lowest address corresponds to the first element and the highest address to the last element.



An array is a part of collection type data and it stands for variable-size arrays. We will study other collection types in a later chapter **'PL/SQL Collections'**.

Each element in a **varray** has an index associated with it. It also has a maximum size that can be changed dynamically.

Creating a Varray Type

A varray type is created with the **CREATE TYPE** statement. You must specify the maximum size and the type of elements stored in the varray.

The basic syntax for creating a VARRAY type at the schema level is −

```
CREATE OR REPLACE TYPE varray_type_name IS VARRAY(n) of <element_type>
```

Where,

- *varray_type_name* is a valid attribute name,
- *n* is the number of elements (maximum) in the varray,
- *element_type* is the data type of the elements of the array.

Maximum size of a varray can be changed using the **ALTER TYPE** statement.

For example,

```
CREATE Or REPLACE TYPE namearray AS VARRAY(3) OF VARCHAR2(10);
/
```

```
Type created.
```

The basic syntax for creating a VARRAY type within a PL/SQL block is −

TYPE varray_type_name IS VARRAY(n) of <element_type>

For example −

```
TYPE namearray IS VARRAY(5) OF VARCHAR2(10);
Type grades IS VARRAY(5) OF INTEGER;
```

Let us now work out on a few examples to understand the concept −

Example 1

The following program illustrates the use of varrays −

```
DECLARE
  type namesarray IS VARRAY(5) OF VARCHAR2(10);
  type grades IS VARRAY(5) OF INTEGER;
  names namesarray;
  marks grades;
  total integer;
BEGIN
  names := namesarray('Kavita', 'Pritam', 'Ayan', 'Rishav', 'Aziz');
  marks:= grades(98, 97, 78, 87, 92);
  total := names.count;
  dbms_output.put_line('Total '|| total || ' Students');
  FOR i in 1 .. total LOOP
    dbms_output.put_line('Student: ' || names(i) || '
    Marks: ' || marks(i));
  END LOOP;
END;
/
```

When the above code is executed at the SQL prompt, it produces the following result −

```
Total 5 Students
Student: Kavita  Marks: 98
Student: Pritam  Marks: 97
Student: Ayan  Marks: 78
Student: Rishav  Marks: 87
Student: Aziz  Marks: 92

PL/SQL procedure successfully completed.
```

Please note −

- In Oracle environment, the starting index for varrays is always 1.
- You can initialize the varray elements using the constructor method of the varray type, which has the same name as the varray.

- Varrays are one-dimensional arrays.
- A varray is automatically NULL when it is declared and must be initialized before its elements can be referenced.

Example 2

Elements of a varray could also be a %ROWTYPE of any database table or %TYPE of any database table field. The following example illustrates the concept.

We will use the CUSTOMERS table stored in our database as −

Select * from customers;

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
+----+----------+-----+-----------+----------+
```

Following example makes the use of **cursor**, which you will study in detail in a separate chapter.

```
DECLARE
  CURSOR c_customers is
  SELECT  name FROM customers;
  type c_list is varray (6) of customers.name%type;
  name_list c_list := c_list();
  counter integer :=0;
BEGIN
  FOR n IN c_customers LOOP
    counter := counter + 1;
    name_list.extend;
    name_list(counter)  := n.name;
    dbms_output.put_line('Customer('||counter ||'):'||name_list(counter));
  END LOOP;
END;
/
```

When the above code is executed at the SQL prompt, it produces the following result −

```
Customer(1): Ramesh
Customer(2): Khilan
Customer(3): kaushik
Customer(4): Chaitali
Customer(5): Hardik
Customer(6): Komal
```

```
PL/SQL procedure successfully completed.
```

PL/SQL PROCEDURES

we will discuss Procedures in PL/SQL. A **subprogram** is a program unit/module that performs a particular task. These subprograms are combined to form larger programs. This is basically called the 'Modular design'. A subprogram can be invoked by another subprogram or program which is called the **calling program**.

A subprogram can be created –

- At the schema level
- Inside a package
- Inside a PL/SQL block

At the schema level, subprogram is a **standalone subprogram**. It is created with the CREATE PROCEDURE or the CREATE FUNCTION statement. It is stored in the database and can be deleted with the DROP PROCEDURE or DROP FUNCTION statement.

A subprogram created inside a package is a **packaged subprogram**. It is stored in the database and can be deleted only when the package is deleted with the DROP PACKAGE statement. We will discuss packages in the chapter **'PL/SQL - Packages'**.

PL/SQL subprograms are named PL/SQL blocks that can be invoked with a set of parameters. PL/SQL provides two kinds of subprograms –

- **Functions** – These subprograms return a single value; mainly used to compute and return a value.
- **Procedures** – These subprograms do not return a value directly; mainly used to perform an action.

This chapter is going to cover important aspects of a **PL/SQL procedure**. We will discuss **PL/SQL function** in the next chapter.

Parts of a PL/SQL Subprogram

Each PL/SQL subprogram has a name, and may also have a parameter list. Like anonymous PL/SQL blocks, the named blocks will also have the following three parts –

S.No	Parts & Description
1	**Declarative Part** It is an optional part. However, the declarative part for a subprogram does not start with

	the DECLARE keyword. It contains declarations of types, cursors, constants, variables, exceptions, and nested subprograms. These items are local to the subprogram and cease to exist when the subprogram completes execution.
2	**Executable Part** This is a mandatory part and contains statements that perform the designated action.
3	**Exception-handling** This is again an optional part. It contains the code that handles run-time errors.

Creating a Procedure

A procedure is created with the **CREATE OR REPLACE PROCEDURE** statement. The simplified syntax for the CREATE OR REPLACE PROCEDURE statement is as follows −

```
CREATE [OR REPLACE] PROCEDURE procedure_name
[(parameter_name [IN | OUT | IN OUT] type [, ...])]
{IS | AS}
BEGIN
 < procedure_body >
END procedure_name;
```

Where,

- *procedure-name* specifies the name of the procedure.
- [OR REPLACE] option allows the modification of an existing procedure.
- The optional parameter list contains name, mode and types of the parameters. **IN** represents the value that will be passed from outside and OUT represents the parameter that will be used to return a value outside of the procedure.
- *procedure-body* contains the executable part.
- The AS keyword is used instead of the IS keyword for creating a standalone procedure.

Example

The following example creates a simple procedure that displays the string 'Hello World!' on the screen when executed.

```
CREATE OR REPLACE PROCEDURE greetings
AS
BEGIN
  dbms_output.put_line('Hello World!');
END;
/
```

When the above code is executed using the SQL prompt, it will produce the following result −

```
Procedure created.
```

Executing a Standalone Procedure

A standalone procedure can be called in two ways −

- Using the **EXECUTE** keyword
- Calling the name of the procedure from a PL/SQL block

The above procedure named **'greetings'** can be called with the EXECUTE keyword as −

```
EXECUTE greetings;
```

The above call will display −

```
Hello World

PL/SQL procedure successfully completed.
```

The procedure can also be called from another PL/SQL block −

```
BEGIN
  greetings;
END;
/
```

The above call will display −

```
Hello World

PL/SQL procedure successfully completed.
```

Deleting a Standalone Procedure

A standalone procedure is deleted with the **DROP PROCEDURE** statement. Syntax for deleting a procedure is −

```
DROP PROCEDURE procedure-name;
```

You can drop the greetings procedure by using the following statement −

```
DROP PROCEDURE greetings;
```

Parameter Modes in PL/SQL Subprograms

The following table lists out the parameter modes in PL/SQL subprograms −

S.No	Parameter Mode & Description

1	**IN** An IN parameter lets you pass a value to the subprogram. **It is a read-only parameter**. Inside the subprogram, an IN parameter acts like a constant. It cannot be assigned a value. You can pass a constant, literal, initialized variable, or expression as an IN parameter. You can also initialize it to a default value; however, in that case, it is omitted from the subprogram call. **It is the default mode of parameter passing. Parameters are passed by reference**.
2	**OUT** An OUT parameter returns a value to the calling program. Inside the subprogram, an OUT parameter acts like a variable. You can change its value and reference the value after assigning it. **The actual parameter must be variable and it is passed by value**.
3	**IN OUT** An **IN OUT** parameter passes an initial value to a subprogram and returns an updated value to the caller. It can be assigned a value and the value can be read. The actual parameter corresponding to an IN OUT formal parameter must be a variable, not a constant or an expression. Formal parameter must be assigned a value. **Actual parameter is passed by value.**

IN & OUT Mode Example 1

This program finds the minimum of two values. Here, the procedure takes two numbers using the IN mode and returns their minimum using the OUT parameters.

```
DECLARE
  a number;
  b number;
  c number;
PROCEDURE findMin(x IN number, y IN number, z OUT number) IS
BEGIN
  IF x < y THEN
    z:= x;
  ELSE
    z:= y;
  END IF;
END;
BEGIN
  a:= 23;
  b:= 45;
  findMin(a, b, c);
  dbms_output.put_line(' Minimum of (23, 45) : ' || c);
END;
```

```
/
```

When the above code is executed at the SQL prompt, it produces the following result −

```
Minimum of (23, 45) : 23

PL/SQL procedure successfully completed.
```

IN & OUT Mode Example 2

This procedure computes the square of value of a passed value. This example shows how we can use the same parameter to accept a value and then return another result.

```
DECLARE
  a number;
PROCEDURE squareNum(x IN OUT number) IS
BEGIN
 x := x * x;
END;
BEGIN
  a:= 23;
  squareNum(a);
  dbms_output.put_line(' Square of (23): ' || a);
END;
/
```

When the above code is executed at the SQL prompt, it produces the following result −

```
Square of (23): 529

PL/SQL procedure successfully completed.
```

Methods for Passing Parameters

Actual parameters can be passed in three ways −

- Positional notation
- Named notation
- Mixed notation

Positional Notation

In positional notation, you can call the procedure as −

```
findMin(a, b, c, d);
```

In positional notation, the first actual parameter is substituted for the first formal parameter; the second actual parameter is substituted for the second formal parameter, and so on. So, **a** is substituted for **x, b** is substituted for **y, c** is substituted for **z** and **d** is substituted for **m**.

Named Notation

In named notation, the actual parameter is associated with the formal parameter using the **arrow symbol (=>)**. The procedure call will be like the following −

```
findMin(x => a, y => b, z => c, m => d);
```

Mixed Notation

In mixed notation, you can mix both notations in procedure call; however, the positional notation should precede the named notation.

The following call is legal −

```
findMin(a, b, c, m => d);
```

However, this is not legal:

```
findMin(x => a, b, c, d);
```

PL/SQL FUNCTIONS

we will discuss the functions in PL/SQL. A function is same as a procedure except that it returns a value. Therefore, all the discussions of the previous chapter are true for functions too.

Creating a Function

A standalone function is created using the **CREATE FUNCTION** statement. The simplified syntax for the **CREATE OR REPLACE PROCEDURE** statement is as follows −

```
CREATE [OR REPLACE] FUNCTION function_name
[(parameter_name [IN | OUT | IN OUT] type [, ...])]
RETURN return_datatype
{IS | AS}
BEGIN
  < function_body >
END [function_name];
```

Where,

- *function-name* specifies the name of the function.
- [OR REPLACE] option allows the modification of an existing function.
- The optional parameter list contains name, mode and types of the parameters. IN represents the value that will be passed from outside and OUT represents the parameter that will be used to return a value outside of the procedure.
- The function must contain a **return** statement.
- The *RETURN* clause specifies the data type you are going to return from the function.
- *function-body* contains the executable part.

- The AS keyword is used instead of the IS keyword for creating a standalone function.

Example

The following example illustrates how to create and call a standalone function. This function returns the total number of CUSTOMERS in the customers table.

We will use the CUSTOMERS table, which we had created in the PL/SQL Variables chapter –

Select * from customers;

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
+----+----------+-----+-----------+----------+
```

```
CREATE OR REPLACE FUNCTION totalCustomers
RETURN number IS
  total number(2) := 0;
BEGIN
  SELECT count(*) into total
  FROM customers;

  RETURN total;
END;
/
```

When the above code is executed using the SQL prompt, it will produce the following result –

```
Function created.
```

Calling a Function

While creating a function, you give a definition of what the function has to do. To use a function, you will have to call that function to perform the defined task. When a program calls a function, the program control is transferred to the called function.

A called function performs the defined task and when its return statement is executed or when the **last end statement** is reached, it returns the program control back to the main program.

To call a function, you simply need to pass the required parameters along with the function name and if the function returns a value, then you can store the returned value. Following program calls the function **totalCustomers** from an anonymous block –

```
DECLARE
  c number(2);
BEGIN
  c := totalCustomers();
  dbms_output.put_line('Total no. of Customers: ' || c);
END;
/
```

When the above code is executed at the SQL prompt, it produces the following result −

```
Total no. of Customers: 6

PL/SQL procedure successfully completed.
```

Example

The following example demonstrates Declaring, Defining, and Invoking a Simple PL/SQL Function that computes and returns the maximum of two values.

```
DECLARE
  a number;
  b number;
  c number;
FUNCTION findMax(x IN number, y IN number)
RETURN number
IS
   z number;
BEGIN
  IF x > y THEN
    z:= x;
  ELSE
    Z:= y;
  END IF;
  RETURN z;
END;
BEGIN
  a:= 23;
  b:= 45;
  c := findMax(a, b);
  dbms_output.put_line(' Maximum of (23,45): ' || c);
END;
/
```

When the above code is executed at the SQL prompt, it produces the following result −

```
Maximum of (23,45): 45

PL/SQL procedure successfully completed.
```

PL/SQL Recursive Functions

We have seen that a program or subprogram may call another subprogram. When a subprogram calls itself, it is referred to as a recursive call and the process is known as **recursion**.

To illustrate the concept, let us calculate the factorial of a number. Factorial of a number n is defined as –

```
n! = n*(n-1)!
   = n*(n-1)*(n-2)!
      ...
   = n*(n-1)*(n-2)*(n-3)... 1
```

The following program calculates the factorial of a given number by calling itself recursively –

```
DECLARE
   num number;
   factorial number;

FUNCTION fact(x number)
RETURN number
IS
   f number;
BEGIN
   IF x=0 THEN
      f := 1;
   ELSE
      f := x * fact(x-1);
   END IF;
RETURN f;
END;

BEGIN
   num:= 6;
   factorial := fact(num);
   dbms_output.put_line(' Factorial '|| num || ' is ' || factorial);
END;
/
```

When the above code is executed at the SQL prompt, it produces the following result –

```
Factorial 6 is 720

PL/SQL procedure successfully completed.
```

PL/SQL Cursors

we will discuss the cursors in PL/SQL. Oracle creates a memory area, known as the context area, for processing an SQL statement, which contains all the information needed for processing the statement; for example, the number of rows processed, etc.

A **cursor** is a pointer to this context area. PL/SQL controls the context area through a cursor. A cursor holds the rows (one or more) returned by a SQL statement. The set of rows the cursor holds is referred to as the **active set**.

You can name a cursor so that it could be referred to in a program to fetch and process the rows returned by the SQL statement, one at a time. There are two types of cursors −

- Implicit cursors
- Explicit cursors

Implicit Cursors

Implicit cursors are automatically created by Oracle whenever an SQL statement is executed, when there is no explicit cursor for the statement. Programmers cannot control the implicit cursors and the information in it.

Whenever a DML statement (INSERT, UPDATE and DELETE) is issued, an implicit cursor is associated with this statement. For INSERT operations, the cursor holds the data that needs to be inserted. For UPDATE and DELETE operations, the cursor identifies the rows that would be affected.

In PL/SQL, you can refer to the most recent implicit cursor as the **SQL cursor**, which always has attributes such as **%FOUND, %ISOPEN, %NOTFOUND**, and **%ROWCOUNT**. The SQL cursor has additional attributes, **%BULK_ROWCOUNT** and **%BULK_EXCEPTIONS**, designed for use with the **FORALL** statement. The following table provides the description of the most used attributes −

S.No	Attribute & Description
1	**%FOUND** Returns TRUE if an INSERT, UPDATE, or DELETE statement affected one or more rows or a SELECT INTO statement returned one or more rows. Otherwise, it returns FALSE.
2	**%NOTFOUND** The logical opposite of %FOUND. It returns TRUE if an INSERT, UPDATE, or DELETE statement affected no rows, or a SELECT INTO statement returned no rows. Otherwise, it returns FALSE.

3	**%ISOPEN** Always returns FALSE for implicit cursors, because Oracle closes the SQL cursor automatically after executing its associated SQL statement.
4	**%ROWCOUNT** Returns the number of rows affected by an INSERT, UPDATE, or DELETE statement, or returned by a SELECT INTO statement.

Any SQL cursor attribute will be accessed as **sql%attribute_name** as shown below in the example.

Example

We will be using the CUSTOMERS table we had created and used in the previous chapters.

Select * from customers;

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
+----+----------+-----+-----------+----------+
```

The following program will update the table and increase the salary of each customer by 500 and use the **SQL%ROWCOUNT** attribute to determine the number of rows affected −

```
DECLARE
  total_rows number(2);
BEGIN
  UPDATE customers
  SET salary = salary + 500;
  IF sql%notfound THEN
    dbms_output.put_line('no customers selected');
  ELSIF sql%found THEN
    total_rows := sql%rowcount;
    dbms_output.put_line( total_rows || ' customers selected ');
  END IF;
END;
/
```

When the above code is executed at the SQL prompt, it produces the following result −

```
6 customers selected
```

PL/SQL procedure successfully completed.

If you check the records in customers table, you will find that the rows have been updated –

Select * from customers;

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2500.00 |
|  2 | Khilan   |  25 | Delhi     |  2000.00 |
|  3 | kaushik  |  23 | Kota      |  2500.00 |
|  4 | Chaitali |  25 | Mumbai    |  7000.00 |
|  5 | Hardik   |  27 | Bhopal    |  9000.00 |
|  6 | Komal    |  22 | MP        |  5000.00 |
+----+----------+-----+-----------+----------+
```

Explicit Cursors

Explicit cursors are programmer-defined cursors for gaining more control over the **context area**. An explicit cursor should be defined in the declaration section of the PL/SQL Block. It is created on a SELECT Statement which returns more than one row.

The syntax for creating an explicit cursor is –

CURSOR cursor_name IS select_statement;

Working with an explicit cursor includes the following steps –

- Declaring the cursor for initializing the memory
- Opening the cursor for allocating the memory
- Fetching the cursor for retrieving the data
- Closing the cursor to release the allocated memory

Declaring the Cursor

Declaring the cursor defines the cursor with a name and the associated SELECT statement. For example –

```
CURSOR c_customers IS
  SELECT id, name, address FROM customers;
```

Opening the Cursor

Opening the cursor allocates the memory for the cursor and makes it ready for fetching the rows returned by the SQL statement into it. For example, we will open the above defined cursor as follows –

```
OPEN c_customers;
```

Fetching the Cursor

Fetching the cursor involves accessing one row at a time. For example, we will fetch rows from the above-opened cursor as follows −

```
FETCH c_customers INTO c_id, c_name, c_addr;
```

Closing the Cursor

Closing the cursor means releasing the allocated memory. For example, we will close the above-opened cursor as follows −

```
CLOSE c_customers;
```

Example

Following is a complete example to illustrate the concepts of explicit cursors &minua;

```
DECLARE
  c_id customers.id%type;
  c_name customer.name%type;
  c_addr customers.address%type;
  CURSOR c_customers is
    SELECT id, name, address FROM customers;
BEGIN
  OPEN c_customers;
  LOOP
  FETCH c_customers into c_id, c_name, c_addr;
    EXIT WHEN c_customers%notfound;
    dbms_output.put_line(c_id || ' ' || c_name || ' ' || c_addr);
  END LOOP;
  CLOSE c_customers;
END;
/
```

When the above code is executed at the SQL prompt, it produces the following result −

```
1 Ramesh Ahmedabad
2 Khilan Delhi
3 kaushik Kota
4 Chaitali Mumbai
5 Hardik Bhopal
6 Komal MP

PL/SQL procedure successfully completed.
```

PL/SQL Exceptions

we will discuss Exceptions in PL/SQL. An exception is an error condition during a program execution. PL/SQL supports programmers to catch such conditions using **EXCEPTION** block in the program and an appropriate action is taken against the error condition. There are two types of exceptions −

- System-defined exceptions
- User-defined exceptions

Syntax for Exception Handling

The general syntax for exception handling is as follows. Here you can list down as many exceptions as you can handle. The default exception will be handled using ***WHEN others THEN*** −

```
DECLARE
  <declarations section>
BEGIN
  <executable command(s)>
EXCEPTION
  <exception handling goes here >
  WHEN exception1 THEN
    exception1-handling-statements
  WHEN exception2  THEN
    exception2-handling-statements
  WHEN exception3 THEN
    exception3-handling-statements
  ........
  WHEN others THEN
    exception3-handling-statements
END;
```

Example

Let us write a code to illustrate the concept. We will be using the CUSTOMERS table we had created and used in the previous chapters −

```
DECLARE
  c_id customers.id%type := 8;
  c_name customerS.Name%type;
  c_addr customers.address%type;
BEGIN
  SELECT  name, address INTO  c_name, c_addr
  FROM customers
  WHERE id = c_id;
  DBMS_OUTPUT.PUT_LINE ('Name: '||  c_name);
  DBMS_OUTPUT.PUT_LINE ('Address: ' || c_addr);

EXCEPTION
  WHEN no_data_found THEN
```

```
    dbms_output.put_line('No such customer!');
  WHEN others THEN
    dbms_output.put_line('Error!');
END;
/
```

When the above code is executed at the SQL prompt, it produces the following result −

```
No such customer!

PL/SQL procedure successfully completed.
```

The above program displays the name and address of a customer whose ID is given. Since there is no customer with ID value 8 in our database, the program raises the run-time exception **NO_DATA_FOUND**, which is captured in the **EXCEPTION block**.

Raising Exceptions

Exceptions are raised by the database server automatically whenever there is any internal database error, but exceptions can be raised explicitly by the programmer by using the command **RAISE**. Following is the simple syntax for raising an exception −

```
DECLARE
  exception_name EXCEPTION;
BEGIN
  IF condition THEN
    RAISE exception_name;
  END IF;
EXCEPTION
  WHEN exception_name THEN
  statement;
END;
```

You can use the above syntax in raising the Oracle standard exception or any user-defined exception. In the next section, we will give you an example on raising a user-defined exception. You can raise the Oracle standard exceptions in a similar way.

User-defined Exceptions

PL/SQL allows you to define your own exceptions according to the need of your program. A user-defined exception must be declared and then raised explicitly, using either a RAISE statement or the procedure **DBMS_STANDARD.RAISE_APPLICATION_ERROR**.

The syntax for declaring an exception is −

```
DECLARE
  my-exception EXCEPTION;
```

Example

The following example illustrates the concept. This program asks for a customer ID, when the user enters an invalid ID, the exception **invalid_id** is raised.

```
DECLARE
  c_id customers.id%type := &cc_id;
  c_name customerS.Name%type;
  c_addr customers.address%type;
  -- user defined exception
  ex_invalid_id  EXCEPTION;
BEGIN
  IF c_id <= 0 THEN
    RAISE ex_invalid_id;
  ELSE
    SELECT  name, address INTO  c_name, c_addr
    FROM customers
    WHERE id = c_id;
    DBMS_OUTPUT.PUT_LINE ('Name: '||  c_name);
    DBMS_OUTPUT.PUT_LINE ('Address: ' || c_addr);
  END IF;

EXCEPTION
  WHEN ex_invalid_id THEN
    dbms_output.put_line('ID must be greater than zero!');
  WHEN no_data_found THEN
    dbms_output.put_line('No such customer!');
  WHEN others THEN
    dbms_output.put_line('Error!');
END;
/
```

When the above code is executed at the SQL prompt, it produces the following result −

```
Enter value for cc_id: -6 (let's enter a value -6)
old  2: c_id customers.id%type := &cc_id;
new  2: c_id customers.id%type := -6;
ID must be greater than zero!

PL/SQL procedure successfully completed.
```

Pre-defined Exceptions

PL/SQL provides many pre-defined exceptions, which are executed when any database rule is violated by a program. For example, the predefined exception NO_DATA_FOUND is raised when a SELECT INTO statement returns no rows. The following table lists few of the important pre-defined exceptions −

Exception	**Oracle Error**	**SQLCODE**	**Description**
ACCESS_INTO_NULL	06530	-6530	It is raised when a null object is automatically

			assigned a value.
CASE_NOT_FOUND	06592	-6592	It is raised when none of the choices in the WHEN clause of a CASE statement is selected, and there is no ELSE clause.
COLLECTION_IS_NULL	06531	-6531	It is raised when a program attempts to apply collection methods other than EXISTS to an uninitialized nested table or varray, or the program attempts to assign values to the elements of an uninitialized nested table or varray.
DUP_VAL_ON_INDEX	00001	-1	It is raised when duplicate values are attempted to be stored in a column with unique index.
INVALID_CURSOR	01001	-1001	It is raised when attempts are made to make a cursor operation that is not allowed, such as closing an unopened cursor.
INVALID_NUMBER	01722	-1722	It is raised when the conversion of a character string into a number fails because the string does not represent a valid number.
LOGIN_DENIED	01017	-1017	It is raised when a program attempts to log on to the database with an invalid username or password.
NO_DATA_FOUND	01403	+100	It is raised when a SELECT INTO statement returns no rows.
NOT_LOGGED_ON	01012	-1012	It is raised when a database call is issued without being connected to the database.

PROGRAM_ERROR	06501	-6501	It is raised when PL/SQL has an internal problem.
ROWTYPE_MISMATCH	06504	-6504	It is raised when a cursor fetches value in a variable having incompatible data type.
SELF_IS_NULL	30625	-30625	It is raised when a member method is invoked, but the instance of the object type was not initialized.
STORAGE_ERROR	06500	-6500	It is raised when PL/SQL ran out of memory or memory was corrupted.
TOO_MANY_ROWS	01422	-1422	It is raised when a SELECT INTO statement returns more than one row.
VALUE_ERROR	06502	-6502	It is raised when an arithmetic, conversion, truncation, or sizeconstraint error occurs.
ZERO_DIVIDE	01476	1476	It is raised when an attempt is made to divide a number by zero.

PL/SQL Triggers

we will discuss Triggers in PL/SQL. Triggers are stored programs, which are automatically executed or fired when some events occur. Triggers are, in fact, written to be executed in response to any of the following events −

- A **database manipulation (DML)** statement (DELETE, INSERT, or UPDATE)
- A **database definition (DDL)** statement (CREATE, ALTER, or DROP).
- A **database operation** (SERVERERROR, LOGON, LOGOFF, STARTUP, or SHUTDOWN).

Triggers can be defined on the table, view, schema, or database with which the event is associated.

Benefits of Triggers

Triggers can be written for the following purposes −

- Generating some derived column values automatically
- Enforcing referential integrity
- Event logging and storing information on table access
- Auditing
- Synchronous replication of tables
- Imposing security authorizations
- Preventing invalid transactions

Creating Triggers

The syntax for creating a trigger is −

```
CREATE [OR REPLACE ] TRIGGER trigger_name
{BEFORE | AFTER | INSTEAD OF }
{INSERT [OR] | UPDATE [OR] | DELETE}
[OF col_name]
ON table_name
[REFERENCING OLD AS o NEW AS n]
[FOR EACH ROW]
WHEN (condition)
DECLARE
Declaration-statements
BEGIN
Executable-statements
EXCEPTION
Exception-handling-statements
END;
```

Where,

- CREATE [OR REPLACE] TRIGGER trigger_name − Creates or replaces an existing trigger with the *trigger_name*.
- {BEFORE | AFTER | INSTEAD OF} − This specifies when the trigger will be executed. The INSTEAD OF clause is used for creating trigger on a view.
- {INSERT [OR] | UPDATE [OR] | DELETE} − This specifies the DML operation.
- [OF col_name] − This specifies the column name that will be updated.
- [ON table_name] − This specifies the name of the table associated with the trigger.
- [REFERENCING OLD AS o NEW AS n] − This allows you to refer new and old values for various DML statements, such as INSERT, UPDATE, and DELETE.
- [FOR EACH ROW] − This specifies a row-level trigger, i.e., the trigger will be executed for each row being affected. Otherwise the trigger will execute just once when the SQL statement is executed, which is called a table level trigger.
- WHEN (condition) − This provides a condition for rows for which the trigger would fire. This clause is valid only for row-level triggers.

Example

To start with, we will be using the CUSTOMERS table we had created and used in the previous chapters –

Select * from customers;

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
+----+----------+-----+-----------+----------+
```

The following program creates a **row-level** trigger for the customers table that would fire for INSERT or UPDATE or DELETE operations performed on the CUSTOMERS table. This trigger will display the salary difference between the old values and new values –

```
CREATE OR REPLACE TRIGGER display_salary_changes
BEFORE DELETE OR INSERT OR UPDATE ON customers
FOR EACH ROW
WHEN (NEW.ID > 0)
DECLARE
sal_diff number;
BEGIN
sal_diff := :NEW.salary  - :OLD.salary;
dbms_output.put_line('Old salary: ' || :OLD.salary);
dbms_output.put_line('New salary: ' || :NEW.salary);
dbms_output.put_line('Salary difference: ' || sal_diff);
END;
/
```

When the above code is executed at the SQL prompt, it produces the following result –

```
Trigger created.
```

The following points need to be considered here –

- OLD and NEW references are not available for table-level triggers, rather you can use them for record-level triggers.
- If you want to query the table in the same trigger, then you should use the AFTER keyword, because triggers can query the table or change it again only after the initial changes are applied and the table is back in a consistent state.
- The above trigger has been written in such a way that it will fire before any DELETE or INSERT or UPDATE operation on the table, but you can write your trigger on a single or multiple

operations, for example BEFORE DELETE, which will fire whenever a record will be deleted using the DELETE operation on the table.

Triggering a Trigger

Let us perform some DML operations on the CUSTOMERS table. Here is one INSERT statement, which will create a new record in the table −

```
INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (7, 'Kriti', 22, 'HP', 7500.00 );
```

When a record is created in the CUSTOMERS table, the above create trigger, **display_salary_changes** will be fired and it will display the following result −

```
Old salary:
New salary: 7500
Salary difference:
```

Because this is a new record, old salary is not available and the above result comes as null. Let us now perform one more DML operation on the CUSTOMERS table. The UPDATE statement will update an existing record in the table −

```
UPDATE customers
SET salary = salary + 500
WHERE id = 2;
```

When a record is updated in the CUSTOMERS table, the above create trigger, **display_salary_changes** will be fired and it will display the following result −

```
Old salary: 1500
New salary: 2000
Salary difference: 500
```

PL/SQL Package

we will discuss the Packages in PL/SQL. Packages are schema objects that groups logically related PL/SQL types, variables, and subprograms.

A package will have two mandatory parts −

- Package specification
- Package body or definition

Package Specification

The specification is the interface to the package. It just **DECLARES** the types, variables, constants, exceptions, cursors, and subprograms that can be referenced from outside the package. In other words, it contains all information about the content of the package, but excludes the code for the subprograms.

All objects placed in the specification are called **public** objects. Any subprogram not in the package specification but coded in the package body is called a **private** object.

The following code snippet shows a package specification having a single procedure. You can have many global variables defined and multiple procedures or functions inside a package.

```
CREATE PACKAGE cust_sal AS
  PROCEDURE find_sal(c_id customers.id%type);
END cust_sal;
/
```

When the above code is executed at the SQL prompt, it produces the following result −

```
Package created.
```

Package Body

The package body has the codes for various methods declared in the package specification and other private declarations, which are hidden from the code outside the package.

The **CREATE PACKAGE BODY** Statement is used for creating the package body. The following code snippet shows the package body declaration for the ***cust_sal*** package created above. I assumed that we already have CUSTOMERS table created in our database as mentioned in the PL/SQL - Variables chapter.

```
CREATE OR REPLACE PACKAGE BODY cust_sal AS

  PROCEDURE find_sal(c_id customers.id%TYPE) IS
  c_sal customers.salary%TYPE;
  BEGIN
    SELECT salary INTO c_sal
    FROM customers
    WHERE id = c_id;
    dbms_output.put_line('Salary: '|| c_sal);
  END find_sal;
END cust_sal;
/
```

When the above code is executed at the SQL prompt, it produces the following result −

```
Package body created.
```

Using the Package Elements

The package elements (variables, procedures or functions) are accessed with the following syntax −

```
package_name.element_name;
```

Consider, we already have created the above package in our database schema, the following program uses the ***find_sal*** method of the ***cust_sal*** package −

```
DECLARE
  code customers.id%type := &cc_id;
```

```
BEGIN
  cust_sal.find_sal(code);
END;
/
```

When the above code is executed at the SQL prompt, it prompts to enter the customer ID and when you enter an ID, it displays the corresponding salary as follows −

```
Enter value for cc_id: 1
Salary: 3000

PL/SQL procedure successfully completed.
```

Example

The following program provides a more complete package. We will use the CUSTOMERS table stored in our database with the following records −

```
Select * from customers;

+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  3000.00 |
|  2 | Khilan   |  25 | Delhi     |  3000.00 |
|  3 | kaushik  |  23 | Kota      |  3000.00 |
|  4 | Chaitali |  25 | Mumbai    |  7500.00 |
|  5 | Hardik   |  27 | Bhopal    |  9500.00 |
|  6 | Komal    |  22 | MP        |  5500.00 |
+----+----------+-----+-----------+----------+
```

The Package Specification

```
CREATE OR REPLACE PACKAGE c_package AS
  -- Adds a customer
  PROCEDURE addCustomer(c_id   customers.id%type,
  c_name  customerS.No.ame%type,
  c_age  customers.age%type,
  c_addr customers.address%type,
  c_sal  customers.salary%type);

  -- Removes a customer
  PROCEDURE delCustomer(c_id  customers.id%TYPE);
  --Lists all customers
  PROCEDURE listCustomer;

END c_package;
/
```

When the above code is executed at the SQL prompt, it creates the above package and displays the following result −

```
Package created.
```

Creating the Package Body

```
CREATE OR REPLACE PACKAGE BODY c_package AS
  PROCEDURE addCustomer(c_id  customers.id%type,
    c_name customerS.No.ame%type,
    c_age  customers.age%type,
    c_addr  customers.address%type,
    c_sal   customers.salary%type)
  IS
  BEGIN
    INSERT INTO customers (id,name,age,address,salary)
      VALUES(c_id, c_name, c_age, c_addr, c_sal);
  END addCustomer;

  PROCEDURE delCustomer(c_id   customers.id%type) IS
  BEGIN
    DELETE FROM customers
    WHERE id = c_id;
  END delCustomer;

  PROCEDURE listCustomer IS
  CURSOR c_customers is
    SELECT  name FROM customers;
  TYPE c_list is TABLE OF customers.Name%type;
  name_list c_list := c_list();
  counter integer :=0;
  BEGIN
    FOR n IN c_customers LOOP
    counter := counter +1;
    name_list.extend;
    name_list(counter) := n.name;
    dbms_output.put_line('Customer(' ||counter|| ')'||name_list(counter));
    END LOOP;
  END listCustomer;

END c_package;
/
```

The above example makes use of the **nested table**. We will discuss the concept of nested table in the next chapter.

When the above code is executed at the SQL prompt, it produces the following result −

```
Package body created.
```

Using The Package

The following program uses the methods declared and defined in the package *c_package*.

```
DECLARE
  code customers.id%type:= 8;
BEGIN
  c_package.addcustomer(7, 'Rajnish', 25, 'Chennai', 3500);
  c_package.addcustomer(8, 'Subham', 32, 'Delhi', 7500);
  c_package.listcustomer;
  c_package.delcustomer(code);
  c_package.listcustomer;
END;
/
```

When the above code is executed at the SQL prompt, it produces the following result −

```
Customer(1): Ramesh
Customer(2): Khilan
Customer(3): kaushik
Customer(4): Chaitali
Customer(5): Hardik
Customer(6): Komal
Customer(7): Rajnish
Customer(8): Subham
Customer(1): Ramesh
Customer(2): Khilan
Customer(3): kaushik
Customer(4): Chaitali
Customer(5): Hardik
Customer(6): Komal
Customer(7): Rajnish

PL/SQL procedure successfully completed
```

PL/SQL Transactions

we will discuss the transactions in PL/SQL. A database **transaction** is an atomic unit of work that may consist of one or more related SQL statements. It is called atomic because the database modifications brought about by the SQL statements that constitute a transaction can collectively be either committed, i.e., made permanent to the database or rolled back (undone) from the database.

A successfully executed SQL statement and a committed transaction are not same. Even if an SQL statement is executed successfully, unless the transaction containing the statement is committed, it can be rolled back and all changes made by the statement(s) can be undone.

Starting and Ending a Transaction

A transaction has a **beginning** and an **end**. A transaction starts when one of the following events take place −

- The first SQL statement is performed after connecting to the database.
- At each new SQL statement issued after a transaction is completed.

A transaction ends when one of the following events take place −

- A **COMMIT** or a **ROLLBACK** statement is issued.
- A **DDL** statement, such as **CREATE TABLE** statement, is issued; because in that case a COMMIT is automatically performed.
- A **DCL** statement, such as a **GRANT** statement, is issued; because in that case a COMMIT is automatically performed.
- User disconnects from the database.
- User exits from **SQL*PLUS** by issuing the **EXIT** command, a COMMIT is automatically performed.
- SQL*Plus terminates abnormally, a **ROLLBACK** is automatically performed.
- A **DML** statement fails; in that case a ROLLBACK is automatically performed for undoing that DML statement.

Committing a Transaction

A transaction is made permanent by issuing the SQL command COMMIT. The general syntax for the COMMIT command is −

COMMIT;

For example,

```
INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (1, 'Ramesh', 32, 'Ahmedabad', 2000.00 );

INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (2, 'Khilan', 25, 'Delhi', 1500.00 );

INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (3, 'kaushik', 23, 'Kota', 2000.00 );

INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (4, 'Chaitali', 25, 'Mumbai', 6500.00 );

INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (5, 'Hardik', 27, 'Bhopal', 8500.00 );

INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (6, 'Komal', 22, 'MP', 4500.00 );

COMMIT;
```

Rolling Back Transactions

Changes made to the database without COMMIT could be undone using the ROLLBACK command.

The general syntax for the ROLLBACK command is −

ROLLBACK [TO SAVEPOINT < savepoint_name>];

When a transaction is aborted due to some unprecedented situation, like system failure, the entire transaction since a commit is automatically rolled back. If you are not using **savepoint**, then simply use the following statement to rollback all the changes −

ROLLBACK;

Savepoints

Savepoints are sort of markers that help in splitting a long transaction into smaller units by setting some checkpoints. By setting savepoints within a long transaction, you can roll back to a checkpoint if required. This is done by issuing the **SAVEPOINT** command.

The general syntax for the SAVEPOINT command is −

SAVEPOINT < savepoint_name >;

For example

```
INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (7, 'Rajnish', 27, 'HP', 9500.00 );

INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (8, 'Riddhi', 21, 'WB', 4500.00 );
SAVEPOINT sav1;

UPDATE CUSTOMERS
SET SALARY = SALARY + 1000;
ROLLBACK TO sav1;

UPDATE CUSTOMERS
SET SALARY = SALARY + 1000
WHERE ID = 7;
UPDATE CUSTOMERS
SET SALARY = SALARY + 1000
WHERE ID = 8;

COMMIT;
```

ROLLBACK TO sav1 − This statement rolls back all the changes up to the point, where you had marked savepoint sav1.

After that, the new changes that you make will start.

Automatic Transaction Control

To execute a **COMMIT** automatically whenever an **INSERT, UPDATE** or **DELETE** command is executed, you can set the **AUTOCOMMIT** environment variable as −

```
SET AUTOCOMMIT ON;
```

You can turn-off the auto commit mode using the following command −

```
SET AUTOCOMMIT OFF;
```

PL/SQL Date and Time

we will discuss the Date and Time in PL/SQL. There are two classes of date and time related data types in PL/SQL −

- Datetime data types
- Interval data types

The Datetime data types are −

- DATE
- TIMESTAMP
- TIMESTAMP WITH TIME ZONE
- TIMESTAMP WITH LOCAL TIME ZONE

The Interval data types are −

- INTERVAL YEAR TO MONTH
- INTERVAL DAY TO SECOND

Field Values for Datetime and Interval Data Types

Both **datetime** and **interval** data types consist of **fields**. The values of these fields determine the value of the data type. The following table lists the fields and their possible values for datetimes and intervals.

Field Name	**Valid Datetime Values**	**Valid Interval Values**
YEAR	-4712 to 9999 (excluding year 0)	Any nonzero integer
MONTH	01 to 12	0 to 11

DAY	01 to 31 (limited by the values of MONTH and YEAR, according to the rules of the calendar for the locale)	Any nonzero integer
HOUR	00 to 23	0 to 23
MINUTE	00 to 59	0 to 59
SECOND	00 to 59.9(n), where 9(n) is the precision of time fractional seconds The 9(n) portion is not applicable for DATE.	0 to 59.9(n), where 9(n) is the precision of interval fractional seconds
TIMEZONE_HOUR	-12 to 14 (range accommodates daylight savings time changes) Not applicable for DATE or TIMESTAMP.	Not applicable
TIMEZONE_MINUTE	00 to 59 Not applicable for DATE or TIMESTAMP.	Not applicable
TIMEZONE_REGION	Not applicable for DATE or TIMESTAMP.	Not applicable
TIMEZONE_ABBR	Not applicable for DATE or TIMESTAMP.	Not applicable

The Datetime Data Types and Functions

Following are the Datetime data types −

DATE

It stores date and time information in both character and number datatypes. It is made of information on century, year, month, date, hour, minute, and second. It is specified as −

TIMESTAMP

It is an extension of the DATE data type. It stores the year, month, and day of the DATE datatype, along with hour, minute, and second values. It is useful for storing precise time values.

TIMESTAMP WITH TIME ZONE

It is a variant of TIMESTAMP that includes a time zone region name or a time zone offset in its value. The time zone offset is the difference (in hours and minutes) between local time and UTC. This data type is useful for collecting and evaluating date information across geographic regions.

TIMESTAMP WITH LOCAL TIME ZONE

It is another variant of TIMESTAMP that includes a time zone offset in its value.

Following table provides the Datetime functions (where, x has the datetime value) −

S.No	Function Name & Description
1	**ADD_MONTHS(x, y);** Adds **y** months to **x**.
2	**LAST_DAY(x);** Returns the last day of the month.
3	**MONTHS_BETWEEN(x, y);** Returns the number of months between **x** and **y**.
4	**NEXT_DAY(x, day);** Returns the datetime of the next *day* after **x**.
5	**NEW_TIME;** Returns the time/day value from a time zone specified by the user.
6	**ROUND(x [, unit]);** Rounds **x**.
7	**SYSDATE();** Returns the current datetime.
8	**TRUNC(x [, unit]);**

	Truncates **x**.

Timestamp functions (where, x has a timestamp value) −

S.No	Function Name & Description
1	**CURRENT_TIMESTAMP();** Returns a TIMESTAMP WITH TIME ZONE containing the current session time along with the session time zone.
2	**EXTRACT({ YEAR | MONTH | DAY | HOUR | MINUTE | SECOND } | { TIMEZONE_HOUR | TIMEZONE_MINUTE } | { TIMEZONE_REGION | } TIMEZONE_ABBR) FROM x)** Extracts and returns a year, month, day, hour, minute, second, or time zone from **x**.
3	**FROM_TZ(x, time_zone);** Converts the TIMESTAMP x and the time zone specified by time_zone to a TIMESTAMP WITH TIMEZONE.
4	**LOCALTIMESTAMP();** Returns a TIMESTAMP containing the local time in the session time zone.
5	**SYSTIMESTAMP();** Returns a TIMESTAMP WITH TIME ZONE containing the current database time along with the database time zone.
6	**SYS_EXTRACT_UTC(x);** Converts the TIMESTAMP WITH TIMEZONE x to a TIMESTAMP containing the date and time in UTC.
7	**TO_TIMESTAMP(x, [format]);** Converts the string x to a TIMESTAMP.
8	**TO_TIMESTAMP_TZ(x, [format]);**

	Converts the string x to a TIMESTAMP WITH TIMEZONE.

Examples

The following code snippets illustrate the use of the above functions –

Example 1

```
SELECT SYSDATE FROM DUAL;
```

Output –

```
08/31/2012 5:25:34 PM
```

Example 2

```
SELECT TO_CHAR(CURRENT_DATE, 'DD-MM-YYYY HH:MI:SS') FROM DUAL;
```

Output –

```
31-08-2012 05:26:14
```

Example 3

```
SELECT ADD_MONTHS(SYSDATE, 5) FROM DUAL;
```

Output –

```
01/31/2013 5:26:31 PM
```

Example 4

```
SELECT LOCALTIMESTAMP FROM DUAL;
```

Output –

```
8/31/2012 5:26:55.347000 PM
```

The Interval Data Types and Functions

Following are the Interval data types –

- IINTERVAL YEAR TO MONTH – It stores a period of time using the YEAR and MONTH datetime fields.
- INTERVAL DAY TO SECOND – It stores a period of time in terms of days, hours, minutes, and seconds.

Interval Functions

S.No	Function Name & Description

1	**NUMTODSINTERVAL(x, interval_unit);** Converts the number x to an INTERVAL DAY TO SECOND.
2	**NUMTOYMINTERVAL(x, interval_unit);** Converts the number x to an INTERVAL YEAR TO MONTH.
3	**TO_DSINTERVAL(x);** Converts the string x to an INTERVAL DAY TO SECOND.
4	**TO_YMINTERVAL(x);** Converts the string x to an INTERVAL YEAR TO MONTH.

References

1. http://blog.unisoftindia.org/2014/12/oracle-architecture-explained-in.html
2. https://drive.google.com/file/d/188s5lNpx9zIKuOKJpjiJ8iM2gh9pbOXz/view
3. http://www.indoreindira.com/UG/images/BCA/BCA%20IV%20%20Notes/BCA%20IV%20PDF/BCA%20IV%20Sem%20Database%20Management%20System.pdf
4. https://beginnersbook.com/2015/04/levels-of-abstraction-in-dbms/.
5. https://www.tutorialspoint.com/dbms/dbms_data_independence.htm
6. https://whatisdbms.com/data-definition-language-ddl-in-dbms-with-examples/
7. https://www.geeksforgeeks.org/sql-ddl-dql-dml-dcl-tcl-commands/
8. http://www.exploredatabase.com/2017/07/reduce-er-diagram-to-relation-table-solved-exercise.html

Question Paper

1. 20 questions of 2 Marks

1. What is Tuple?
2. Define Database Model.
3. Differentiate between entity and entity set.
4. Difference between File organization and DBMS.
5. List Union compatible rules.
6. List Role of Database administrator.
7. What is Foreign Key.
8. Draw PL/SQL block of code and explain in short.
9. What are database languages list them with example.
10. Explain Views in oracle.
11. What is Tuple?
12. Define Database Model.
13. Differentiate between entity and entity set.
14. Difference between File organization and DBMS.
15. List Role of Database administrator.
16. List the types of Attributes.
17. Draw an ER diagram for a College.
18. What are database languages list them?
19. Write down advantages of DBMS
20. What are the disadvantages of DBMS.?

2. 5 questions of 7 Marks

1. Explain Difference Between Function and procedure.
2. What the advantage of relational database?
3. What do you understand by redundancy of Data?
4. Differentiate between strong entity and weak entity?
5. Define Overall DBMS Architecture
6. What is DBA Stands for? What are the Roles of DBA in DBMS.?
7. What are the different Views of DBMS?

3. 3 questions of 10 Marks

1. a) Explain Oracle Architecture.
   b) Draw an E-R Diagram for College like organization.

2. a) Explain different types of Attributes with suitable example.

   b) Write short note on
      1. Generalization and Specialization
      2. Cursor

3. a) Explain Trigger with an Example

   b) Explain Functions in PL/SQL.

4. a). Create a PL/SQL block of code to find total number of records in a table.

   b). Write a query to Apply a check Constraint on product table so that all the product number should

   start with 'P' like 'P101'.

Assignments to Students

Q1- Write a program in PL/SQL to draw a table of a given number using

a. While loop
b. For loop
c. Simple loop

Q2- Write a program in PL/SQL to reverse of a given string.

Hint: “INDIA” Reverse “AIDNI”

Q3- Write a program in PL/SQL to find the largest among three given numbers.

Q4- Write a program in PL/SQL to find square of a given number.

Q5- Write a program in PL/SQL to check whether the given character is vowel or not

Hint: vowel = a, e, i, o, u

Q.6- Write a PL/SQL block of code to retrieve record for a given key value of a table.

Q.7-Write a PL/SQL block of code to print grade for a given percentage using if...eslif...end if statement.

Q.8- Write a program to interchange two given numbers.

Q.9- Create a procedure to square of a given number.

Q.10- Create a Function to factorial or a given number.

Q.11- Create a PL/SQL code of block to store radius and area of a circle in a table for the given radius 3,4,5,6,7

Q.12- Create a procedure to make some of digits for a given number.

Q.13- Create a statement trigger applied on product table and store values in product_check table;

Q.14- Create a row level Trigger applied on product table and store values in product_check table.