ANNAI WOMEN'S COLLEGE, PUNNAMCHATRAM, KARUR.

DEPARTMENT OF COMMERCE (CA)

(COURSE MATERIAL)

SUBJECT : ORACLE and RDBMS(P16CA23)

CLASS : I.M.COM (CA)

P.Indhu MCA.,M.Phil., Assistant Professor in Computer Science,
Annai Women's College,Karur.

CORE COURSE – VII

ORACLE AND RDBMS

UNIT –I

Database concepts : A relational approach – Database management Systems(DBMS) – RDBMS – Integrity rules – Theoretical Relational Languages – Database Design: Data Modeling and Normalisation.

UNIT –II

Oracle 8: An overview- Personal Databases – Client/server Databases- Table creation & modification : Data types – constraints – creating an oracle Table – Working with tables - Data Management and retrieval.

UNIT – III

Multiple Tables: Join – Set operators – Sub-Query – Advanced Features : Objects , Transactions and Control – Views- Sequences – Synonyms – Index – controlling Access – Object privileges.

UNIT – IV

PL/SQL : Programming Language Basic – History of PL/SQL – Fundamentals – Data types – Variable Declaration – SQL and Control Structures.

UNIT – V

Cursors and Exceptions – Procedures, Functions and Packages.

Text and Reference Books :(Latest revised edition only)

1. Nilesh Shah, “Database Systems Using Oracle” , Prentice – Hall of India private Ltd.

2. Raghu Ramakrishnan & Johannes Gehrke, “Database management systems” , McGraw – Hill – Editions.

3. Abraham silberschatz Henry F.KorthS.Sudarshan, “Database system concepts”. McGraw – Hill – Editions.

P.Indhu MCA.,M.Phil., Assistant Professor in Computer Science,
Annai Women’s College,Karur.

UNIT – I

Basic Relational DBMS Concepts:

A **Relational Database management System**(RDBMS) is a database management system based on the relational model introduced by E.F Codd. In relational model, data is stored in **relations**(tables) and is represented in form of **tuples**(rows).

RDBMS is used to manage Relational database. **Relational database** is a collection of organized set of tables related to each other, and from which data can be accessed easily. Relational Database is the most commonly used database these days.

Data:

Data is nothing but facts and statistics stored or free flowing over a network, generally it's raw and unprocessed. For example: When you visit any website, they might store you IP address, that is data, in return they might add a cookie in your browser, marking you that you visited the website, that is data, your name, it's data, your age, it's data.

Data becomes **information** when it is processed, turning it into something meaningful. Like, based on the cookie data saved on user's browser, if a website can analyse that generally men of age 20-25 visit us more, that is information, derived from the data collected.

Database:

A **Database** is a collection of related data organised in a way that data can be easily accessed, managed and updated. Database can be software based or hardware based, with one sole purpose, storing data.

During early computer days, data was collected and stored on tapes, which were mostly write-only, which means once data is stored on it, it can never be read again. They were slow and bulky, and soon computer scientists realised that they needed a better solution to this problem.

Larry Ellison, the co-founder of **Oracle** was amongst the first few, who realised the need for a software based Database Management System.

DBMS:

A **DBMS** is a software that allows creation, definition and manipulation of database, allowing users to store, process and analyse data easily. DBMS provides us with an interface or a tool, to perform various operations like creating database, storing data in it, updating data, creating tables in the database and a lot more.

DBMS also provides protection and security to the databases. It also maintains data consistency in case of multiple users.

Here are some examples of popular DBMS used these days:

- MySql
- Oracle
- SQL Server
- IBM DB2
- PostgreSQL
- Amazon SimpleDB (cloud based) etc.

Characteristics of Database Management System:

A database management system has following characteristics:

1. **Data stored into Tables:** Data is never directly stored into the database. Data is stored into tables, created inside the database. DBMS also allows to have relationships between tables which makes the data more meaningful and connected. You can easily understand what type of data is stored where by looking at all the tables created in a database.
2. **Reduced Redundancy:** In the modern world hard drives are very cheap, but earlier when hard drives were too expensive, unnecessary repetition of data in database was a big problem. But DBMS follows**Normalisation** which divides the data in such a way that repetition is minimum.
3. **Data Consistency:** On Live data, i.e. data that is being continuosly updated and added, maintaining the consistency of data can become a challenge. But DBMS handles it all by itself.
4. **Support Multiple user and Concurrent Access:** DBMS allows multiple users to work on it(update, insert, delete data) at the same time and still manages to maintain the data consistency.

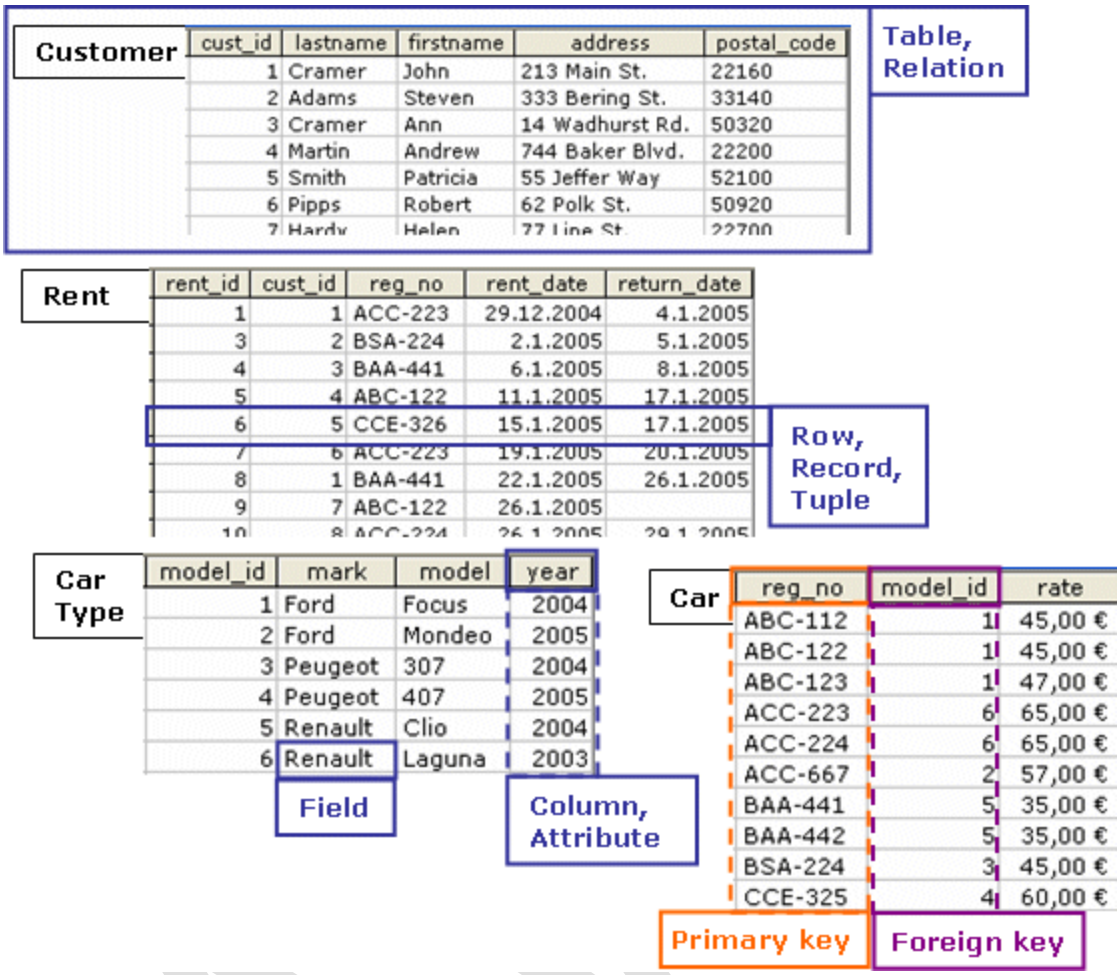
5. **Query Language:** DBMS provides users with a simple Query language, using which data can be easily fetched, inserted, deleted and updated in a database.
6. **Security:** The DBMS also takes care of the security of data, protecting the data from un-authorised access. In a typical DBMS, we can create user accounts with different access permissions, using which we can easily secure our data by restricting user access.
7. DBMS supports **transactions**, which allows us to better handle and manage data integrity in real world applications where multi-threading is extensively used.

Advantages of DBMS

- Segregation of applicaion program.
- Minimal data duplicacy or data redundancy.
- Easy retrieval of data using the Query Language.
- Reduced development time and maintainance need.
- With Cloud Datacenters, we now have Database Management Systems capable of storing almost infinite data.
- Seamless integration into the application programming languages which makes it very easier to add a database to almost any application or website.

Concepts of Relational Database

The terminology and structural concepts of the relational model are explained. In the picture below there is a Car Rental database. In the database there is information about customers, cars and car rentals. The concepts of the relational database are being drawn in the picture.



Picture. The concepts of the relational database (Original designer of the database is Kai Kivimäki, Haaga University of Applied Sciences)

Table (Relation)

A table consists of rows and columns. Each row in the table represents a collection of related values. Tables are used to hold information about the objects to be represented in the database.

In a specific table, there is data of one kind of objects (entities). In the example Car Rental database there are four tables: Customer, Rent, CarType and Car. In the Customer table there is the data of the customers, in the Rent table there is the data of the car rentals.

A table is a database concept, a relation is a relational model concept.

Row (Tuple, Record)

Each row in the table represents a collection of related values of a one object (entity). There is a data of a one customer in one row in Customer table.

A row is a database concept, a tuple is a relational model concept. A record is a little bit outdated term for a tuple or row.

Column (Attribute)

Each column in a table holds a certain kind of data. A Column has a name that describes the data of the column. In the Customer table there are columns e.g. firstname and surname.

A column is a database concept, an attribute is a relational model concept.

Field

A field stores the actual value of an attribute. There are broken lines to show the values of a certain attribute that are stored in the fields.

Primary key

Primary key is the column (or set of columns) which values uniquely indenfy the row. All primary key fields have a different value in a specific table. A table should have a primary key.

The primary key of the table Car is the register number (Reg_No). Two cars can't have the same register number.

Foreign key

Foreign key is a column whose values refer to the primary key of another table.

For example in the Car table the Model_id values refer to the Model_id of the CarType table. The car whose register number is 'ABC-111' is Ford Focus. The Cust_id in the Rent table refers to the Customer table Cust_id. The customer number 5 who has rented a car having register number CCE-326 is Patricia Smith.

Primary Table and Related Table

Primary and related table definition is always between two tables which have a relationship between them.

For example the tables Car and CarType have a relationship. The foreign key (model_id) of the Car table refers to the CarType table primary key (model_id). The CarType table is the primary table and the Car table is the related table.

There is also a relation between the Car and Rent tables. The Reg_no of the Rent table (foreign key) refers to the Reg_no of the Car table (primary key). In this case the Car table is the primary table and the Rent table is the related table.

Primary table is the table whose primary key is referenced from another table's foreing key. The table having this foreign key is a related table.

The primary table is also called a parent table and the related table is called a child table.

What is Normalization? 1NF, 2NF, 3NF & BCNF with Examples

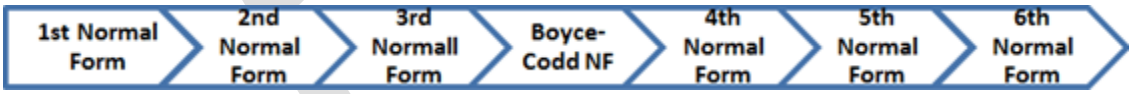
What is Normalization?

Normalization is a database design technique which organizes tables in a manner that reduces redundancy and dependency of data.

It divides larger tables to smaller tables and links them using relationships.

- Database Normal Forms
- 1NF Rules
- What is a KEY?
- What is Composite Key
- 2NF Rules
- Database - Foreign Key
- What are transitive functional dependencies?
- 3NF Rules
- Boyce-Codd Normal Form (BCNF)

The inventor of the relational model Edgar Codd proposed the theory of normalization with the introduction of First Normal Form, and he continued to extend theory with Second and Third Normal Form. Later he joined with Raymond F. Boyce to develop the theory of Boyce-Codd Normal Form.



Database Normalization Examples -

Assume a video library maintains a database of movies rented out. Without any normalization, all information is stored in one table as shown below.

Full Names	Physical Address	Movies rented	Salutation	Category
Janet Jones	First Street Plot No 4	Pirates of the Caribbean, Clash of the Titans	Ms.	Action, Action
Robert Phil	3rd Street 34	Forgetting Sarah Marshal, Daddy's Little Girls	Mr.	Romance, Romance
Robert Phil	5th Avenue	Clash of the Titans	Mr.	Action

Here you see **Movies Rented column has multiple values**.

Database Normal Forms

Now let's move into 1st Normal Forms

1NF (First Normal Form) Rules

- Each table cell should contain a single value.
- Each record needs to be unique.

The above table in 1NF-

1NF Example

FULL NAMES	PHYSICAL ADDRESS	MOVIES RENTED	SALUTATION
Janet Jones	First Street Plot No 4	Pirates of the Caribbean	Ms.
Janet Jones	First Street Plot No 4	Clash of the Titans	Ms.
Robert Phil	3rd Street 34	Forgetting Sarah Marshal	Mr.
Robert Phil	3rd Street 34	Daddy's Little Girls	Mr.
Robert Phil	5th Avenue	Clash of the Titans	Mr.

Before we proceed let's understand a few things --

What is a KEY?

A KEY is a value used to identify a record in a table uniquely. A KEY could be a single column or combination of multiple columns

Note: Columns in a table that are NOT used to identify a record uniquely are called non-key columns.

What is a Primary Key?

A primary is a single column value used to identify a database record uniquely.

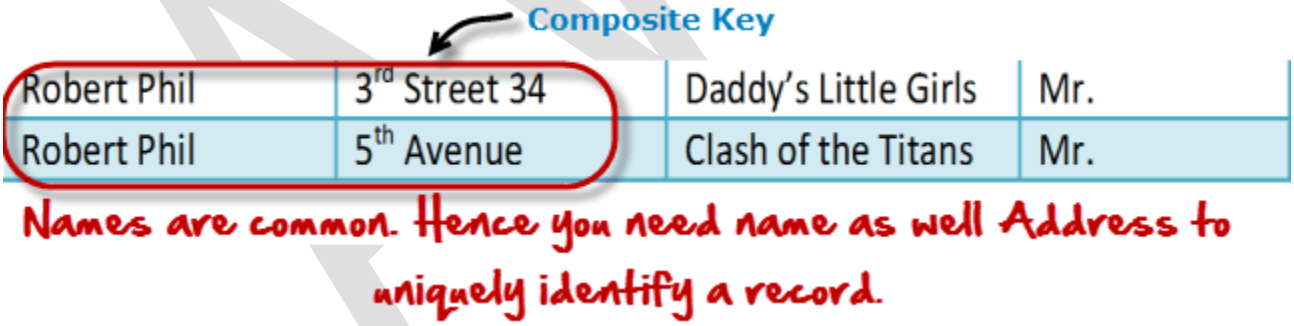
It has following attributes

- A primary key cannot be NULL
- A primary key value must be unique
- The primary key values should rarely be changed
- The primary key must be given a value when a new record is inserted.

What is Composite Key?

A composite key is a primary key composed of multiple columns used to identify a record uniquely

In our database, we have two people with the same name Robert Phil, but they live in different places.



Hence, we require both Full Name and Address to identify a record uniquely. That is a composite key.

Let's move into second normal form 2NF

2NF (Second Normal Form) Rules

- Rule 1- Be in 1NF
- Rule 2- Single Column Primary Key

It is clear that we can't move forward to make our simple database in 2nd Normalization form unless we partition the table above.

MEMBERSHIP ID	FULL NAMES	PHYSICAL ADDRESS	SALUTATION
1	Janet Jones	First Street Plot No 4	Ms.
2	Robert Phil	3rd Street 34	Mr.
3	Robert Phil	5th Avenue	Mr.

MEMBERSHIP ID	MOVIES RENTED
1	Pirates of the Caribbean
1	Clash of the Titans
2	Forgetting Sarah Marshal
2	Daddy's Little Girls
3	Clash of the Titans

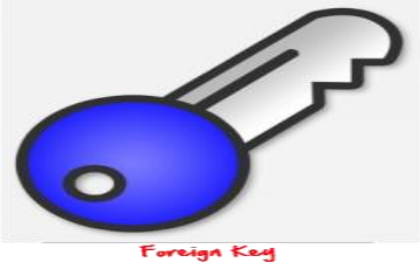
We have divided our 1NF table into two tables viz. Table 1 and Table2. Table 1 contains member information. Table 2 contains information on movies rented.

We have introduced a new column called Membership_id which is the primary key for table 1. Records can be uniquely identified in Table 1 using membership id

Database - Foreign Key

In Table 2, Membership_ID is the Foreign Key

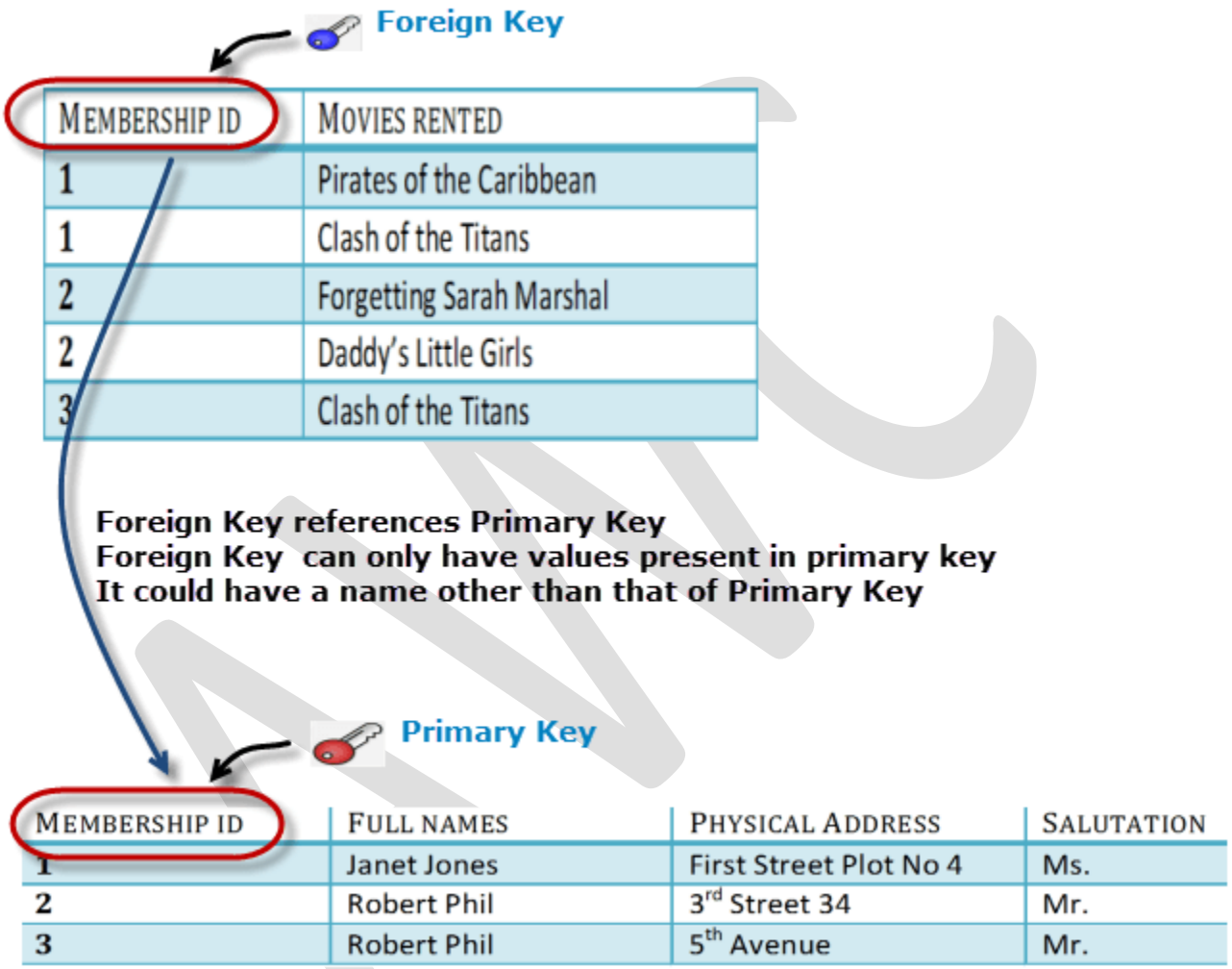
MEMBERSHIP ID	MOVIES RENTED
1	Pirates of the Caribbean
1	Clash of the Titans
2	Forgetting Sarah Marshal
2	Daddy's Little Girls
3	Clash of the Titans



Foreign Key references the primary key of another Table! It helps connect your Tables

- A foreign key can have a different name from its primary key
- It ensures rows in one table have corresponding rows in another
- Unlike the Primary key, they do not have to be unique. Most often they aren't

- Foreign keys can be null even though primary keys can not



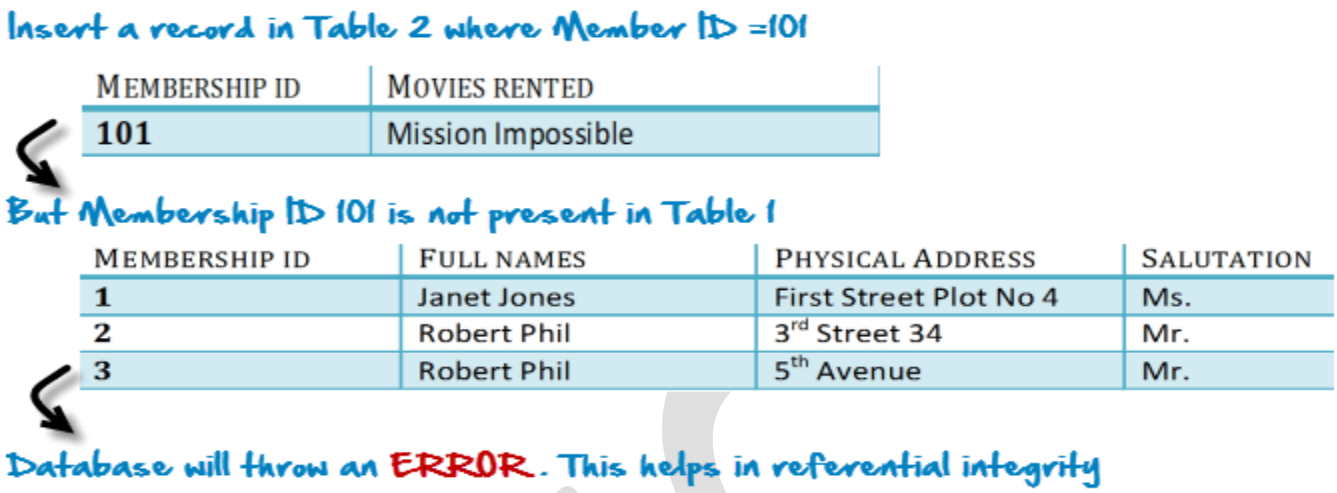
MEMBERSHIP ID	MOVIES RENTED
1	Pirates of the Caribbean
1	Clash of the Titans
2	Forgetting Sarah Marshal
2	Daddy's Little Girls
3	Clash of the Titans

MEMBERSHIP ID	FULL NAMES	PHYSICAL ADDRESS	SALUTATION
1	Janet Jones	First Street Plot No 4	Ms.
2	Robert Phil	3rd Street 34	Mr.
3	Robert Phil	5th Avenue	Mr.

Why do you need a foreign key?

Suppose, a novice inserts a record in Table B such as

You will only be able to insert values into your foreign key that exist in the unique key in the parent table. This helps in referential integrity.



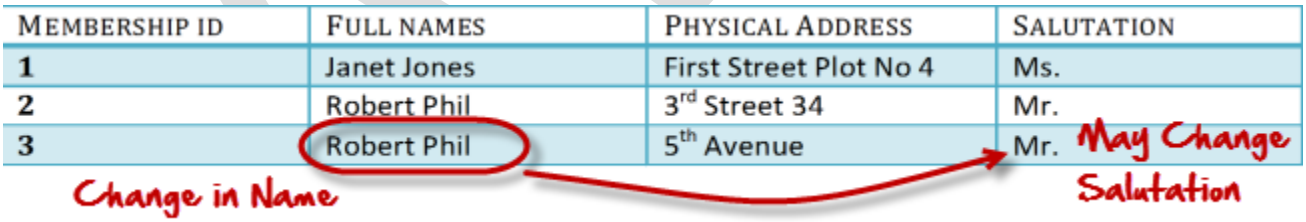
The above problem can be overcome by declaring membership id from Table2 as foreign key of membership id from Table1

Now, if somebody tries to insert a value in the membership id field that does not exist in the parent table, an error will be shown!

What are transitive functional dependencies?

A transitive functional dependency is when changing a non-key column, might cause any of the other non-key columns to change

Consider the table 1. Changing the non-key column Full Name may change Salutation.



Let's move into 3NF

3NF (Third Normal Form) Rules

- Rule 1- Be in 2NF
- Rule 2- Has no transitive functional dependencies

To move our 2NF table into 3NF, we again need to again divide our table.

3NF Example

MEMBERSHIP ID	FULL NAMES	PHYSICAL ADDRESS	SALUTATION ID
1	Janet Jones	First Street Plot No 4	2
2	Robert Phil	3rd Street 34	1
3	Robert Phil	5th Avenue	1

MEMBERSHIP ID	MOVIES RENTED
1	Pirates of the Caribbean
1	Clash of the Titans
2	Forgetting Sarah Marshal
2	Daddy's Little Girls
3	Clash of the Titans

SALUTATION ID	SALUTATION
1	Mr.
2	Ms.
3	Mrs.
4	Dr.

We have again divided our tables and created a new table which stores Salutations.

There are no transitive functional dependencies, and hence our table is in 3NF

In Table 3 Salutation ID is primary key, and in Table 1 Salutation ID is foreign to primary key in Table 3

Now our little example is at a level that cannot further be decomposed to attain higher forms of normalization. In fact, it is already in higher normalization forms. Separate efforts for moving into next levels of normalizing data are normally needed in complex databases. However, we will be discussing next levels of normalizations in brief in the following.

Boyce-Codd Normal Form (BCNF)

Even when a database is in 3rd Normal Form, still there would be anomalies resulted if it has more than one **Candidate** Key.

Sometimes is BCNF is also referred as **3.5 Normal Form.**

4NF (Fourth Normal Form) Rules

If no database table instance contains two or more, independent and multivalued data describing the relevant entity, then it is in 4th Normal Form.

5NF (Fifth Normal Form) Rules

A table is in 5th Normal Form only if it is in 4NF and it cannot be decomposed into any number of smaller tables without loss of data.

UNIT – II

Orcle 8:An Overview

Structured Query Language (SQL), is the set of commands that all programs and users must use to access data in an Oracle database. Application programs and Oracle tools often allow users access to the database without using SQL directly,but these applications in turn must use SQL when executing the user 's request.

The Oracle Client/Server Architecture

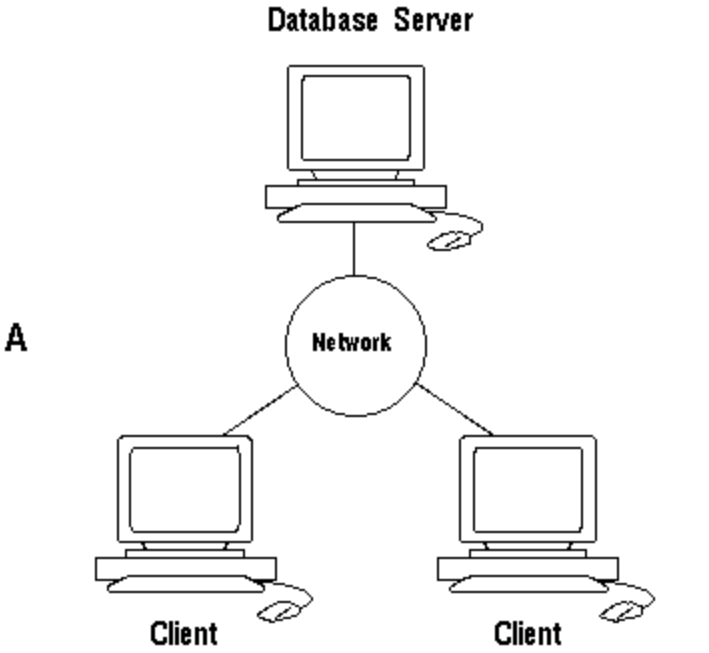
In the Oracle client/server architecture, the database application and the database are separated into two parts: a front-end or client portion, and a back-end or server portion. The client executes the database application that accesses database information and interacts with a user through the keyboard, screen, and pointing device such as a mouse. The server executes the Oracle software and handles the functions required for concurrent, shared data access to an Oracle database.

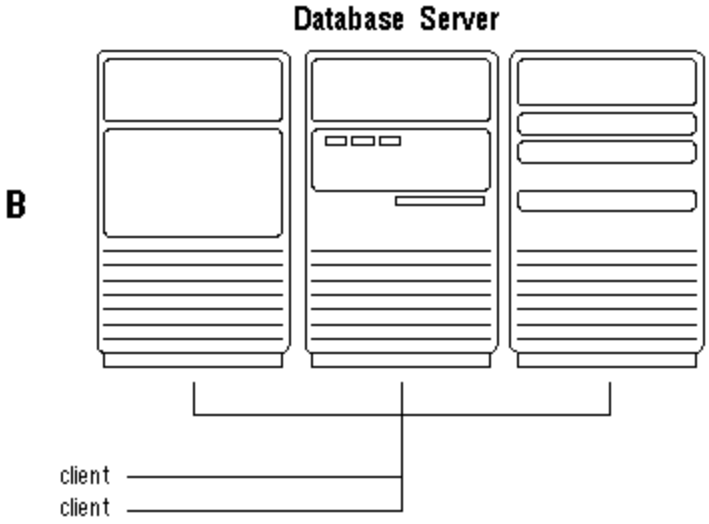
Although the client application and Oracle can be executed on the same computer, it may be more efficient and effective when the client portion(s) and server portion are executed by different computers connected via a network. The following sections discuss possible variants in the Oracle client/server architecture.

Distributed Processing

Distributed processing is the use of more than one processor to divide the processing for an individual task. The following are examples of distributed processing in Oracle database systems:

- The client and server are located on different computers; these computers are connected via a network.
- A single computer has more than one processor, and different processors separate the execution of the client application from Oracle .





The Client/Server Architecture and Distributed Processing

Benefits of the Oracle client/server architecture in a distributed processing environment include the following:

- Client applications are not responsible for performing any data processing. Client applications can concentrate on requesting input from users, requesting desired data from the server, and then analyzing and presenting this data using the display capabilities of the client workstation or the terminal (for example, using graphics or spreadsheets).

- Client applications can be designed with no dependence on the physical location of the data. If the data is moved or distributed to other database servers, the application continues to function with little or no modification.

- Oracle exploits the multitasking and shared-memory facilities of its underlying operating system. As a result, it delivers the highest possible degree of concurrency, data integrity, and performance to its client applications.

- Client workstations or terminals can be optimized for the presentation of data (for example, by providing graphics and mouse support) and the server can be optimized for the processing and storage of data (for example, by having large amounts of memory and disk space).

- If necessary, Oracle can be *scaled*. As your system grows, you can add multiple servers to distribute the database processing load throughout the network (*horizontally scaled*). Alternatively, you can replace Oracle on a less powerful computer, such as a microcomputer, with Oracle running on a minicomputer or mainframe, to take advantage of a larger system's performance (*vertically scaled*). In either case, all data and applications are maintained with little or no modification, since Oracle is portable between systems.

- In networked environments, shared data is stored on the servers, rather than on all computers in the system. This makes it easier and more efficient to manage concurrent access.

- In networked environments, inexpensive, low-end client workstations can be used to access the remote data of the server effectively.

- In networked environments, client applications submit database requests to the server using SQL statements. Once received, the SQL statement is processed by the server, and the results are returned to the client application. Network traffic is kept to a minimum because only the requests and the results are shipped over the network.

SQL Constraints

SQL Create Constraints

Constraints can be specified when the table is created with the CREATE TABLE statement, or after the table is created with the ALTER TABLE statement.

Syntax

```
CREATE TABLE table_name (
  column1 datatype constraint,
  column2 datatype constraint,
```

```
    column3 datatype constraint,
    ....
);
```

SQL Constraints

SQL constraints are used to specify rules for the data in a table.

Constraints are used to limit the type of data that can go into a table. This ensures the accuracy and reliability of the data in the table. If there is any violation between the constraint and the data action, the action is aborted.

Constraints can be column level or table level. Column level constraints apply to a column, and table level constraints apply to the whole table.

The following constraints are commonly used in SQL:

- **NOT NULL** - Ensures that a column cannot have a NULL value
- **UNIQUE** - Ensures that all values in a column are different
- **PRIMARY KEY** - A combination of a NOT NULL and UNIQUE. Uniquely identifies each row in a table
- **FOREIGN KEY** - Uniquely identifies a row/record in another table
- **CHECK** - Ensures that all values in a column satisfies a specific condition
- **DEFAULT** - Sets a default value for a column when no value is specified
- **INDEX** - Used to create and retrieve data from the database very quickly

Oracle PL/SQL Data Types: Character, Number, Boolean, Date, LOB

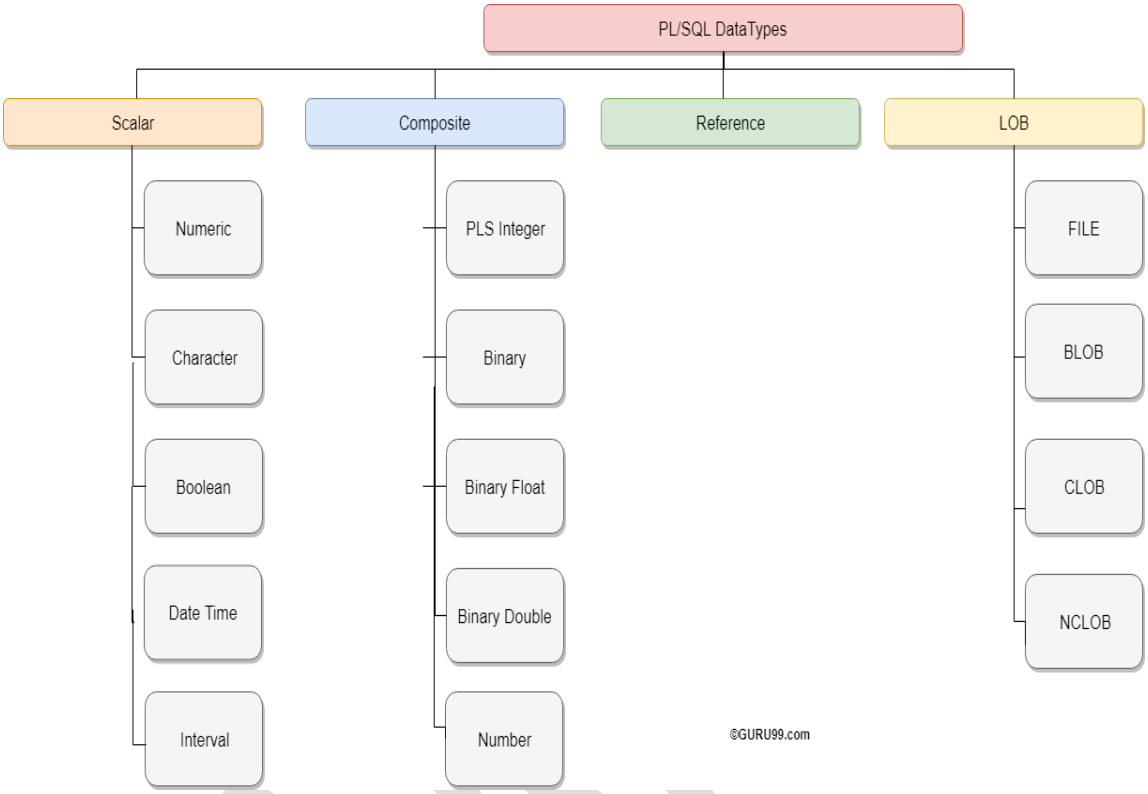
What is PL/SQL Datatypes?

A data type is associated with the specific storage format and range constraints. In Oracle, each value or constant is assigned with a data type.

Basically, it defines how the data is stored, handled and treated by Oracle during the data storage and processing.

The main difference between PL/SQL and SQL data types is, SQL data type are limited to table column while the PL/SQL data types are used in the PL/SQL blocks. More on this later in the tutorial.

Following is the diagram of different Data Types in PL/SQL



In this tutorial, you will learn-

- CH ARACTER Data Type
- NUMBER Data Type
- BOOLEAN Data Type
- DATE Data Type
- LOB Data Type

CHARACTER Data Type:

This data type basically stores alphanumeric characters in string format.

The literal values should always be enclosed in single quotes while assigning them to CHARACTER data type.

This character data type is further classified as follows:

- CHAR Data type (fixed string size)
- VARCHAR2 Data type (variable string size)
- VARCHAR Data type
- NCHAR (native fixed string size)
- NVARCHAR2 (native variable string size)
- LONG and LONG RAW

Data Type	Description
CHAR	This data type stores the string value, and the size of the string is fixed at the time of declaring the variable. • Oracle would be blank-padded the variable if the variable didn't occupy the entire size that has been declared for it, Hence Oracle will allocate the memory for declared size even if the variable didn't occupy it fully. • The size restriction for this data type is 1-2000 bytes. • CHAR data type is more appropriate to use where ever fixed the size of data will be handled.
VARCHAR2	This data type stores the string, but the length of the string is not fixed. • The size restriction for this data type is 1-4000 bytes for table column size and 1-32767 bytes for variables. • The size is defined for each variable at the time of variable declaration. • But Oracle will allocate memory only after the variable is defined, i.e., Oracle will consider only the actual length of the string that is stored in a variable for memory allocation rather than the size that has been given for a variable in the declaration part. • It is always good to use VARCHAR2 instead of CHAR data type to optimize the memory usage.
VARCHAR	This is synonymous with the VARCHAR2 data type. • It is always a good practice to use VARCHAR2 instead of VARCHAR to avoid behavioral changes.
NCHAR	This data type is same as CHAR data type, but the character set will of the national character set. • This character set can be defined for the session using NLS_PARAMETERS. • The character set can be either UTF16 or UTF8. • The size restriction is 1-2000 bytes.
NVARCHAR2	This data type is same as VARCHAR2 data type, but the character set will be of the national character set. • This character set can be defined for the session using NLS_PARAMETERS. • The character set can be either UTF16 or UTF8. • The size restriction is 1-4000 bytes.

	This data type is used to store large text or raw data up to the maximum size of 2GB.
LONG and LONGRAW	• These are mainly used in the data dictionary. • LONG data type is used to store character set data, while LONG RAW is used to store data in binary format. • LONG RAW data type accepts media objects, images, etc. whereas LONG works only on data that can be stored using character set.

NUMBER Data Type:

This data type stores fixed or floating point numbers up to 38 digits of precision. This data type is used to work with fields which will contain only number data. The variable can be declared either with precision and decimal digit details or without this information. Values need not enclose within quotes while assigning for this data type.

Syntax Explanation:

- In the above, the first declaration declares the variable 'A' is of number data type with total precision 8 and decimal digits 2.
- The second declaration declares the variable 'B' is of number data type with total precision 8 and no decimal digits.
- The third declaration is the most generic, declares variable 'C' is of number data type with no restriction in precision or decimal places. It can take up to a maximum of 38 digits.

BOOLEAN Data Type:

This data type stores the logical values. It represents either TRUE or FALSE and mainly used in conditional statements. Values need not enclose within quotes while assigning for this data type.

```
Var1 BOOLEAN;
```

Syntax Explanation:

- In the above, variable 'Var1' is declared as BOOLEAN data type. The output of the code will be either true or false based on the condition set.

DATE Data Type:

This data type stores the values in date format, as date, month, and year. Whenever a variable is defined with DATE data type along with the date it can hold time information and by default time information is set to 12:00:00 if not specified. Values need to enclose within quotes while assigning for this data type.

The standard Oracle time format for input and output is 'DD-MON-YY' and it is again set at NLS_PARAMETERS (NLS_DATE_FORMAT) at the session level.

```
newyear DATE:='01-JAN-2015';
current_date DATE:=SYSDATE;
```

Syntax Explanation:

- In the above, variable 'newyear' is declared as DATE data type and assigned the value of Jan 1st, 2015 date.
- The second declaration declares the variable current_date as DATE data type and assigned the value with current system date.
- Both these variable holds the time information.

LOB Data Type:

This data type is mainly used to store and manipulate large blocks of unstructured data's like images, multimedia files, etc. Oracle prefers LOB instead of the a LONG data type as it is more flexible than the LONG data type. The below are the few main advantage of LOB over LONG data type.

- The number of column in a table with LONG data type is limited to 1, whereas a table has no restriction on a number of columns with LOB data type.
- The data interface tool accepts LOB data type of the table during data replication, but it omits LONG column of the table. These LONG columns need to be replicated manually.
- The size of the LONG column is 2GB, whereas LOB can store up to 128 TB.
- Oracle is constantly improvising the LOB data type in each of their releases according to the modern requirement, whereas LONG data type is constant and not getting many updates.

So, it is always good to use LOB data type instead of the LONG data type. Following are the different LOB data types. They can store up to the size of 128 terabytes.

1. BLOB
2. CLOB and NCLOB
3. BFILE

Data Type	Description	Syntax
BLOB	This data type stores the LOB data in the binary file format up to the maximum size of 128 TB. This doesn't store data based on the character set details, so that it can store the unstructured data such as multimedia objects, images, etc.	`Binary_data BLOB;` **Syntax Explanation:** • In the above, variable 'Binary_data' is declared as a BLOB.

CLOB and NCLOB	CLOB data type stores the LOB data into the character set, whereas NCLOB stores the data in the native character set. Since these data types use character set based storage, these cannot store the data like multimedia, images, etc. that cannot be put into a character string. The maximum size of these data types is 128 TB.
BFILE	• BFILE are the data types that stored the unstructured binary format data outside the database as an operating-system file. • The size of BFILE is to a limited operating system, and they are read-only files and can't be

```
Charac_data CLOB;
```

Syntax Explanation:

- In the above, variable 'Charac_data' is declared as CLOB data type.

Introduction to SQL

Structure Query Language(SQL) is a database query language used for storing and managing data in Relational DBMS. SQL was the first commercial language introduced for E.F Codd's **Relational** model of database. Today almost all RDBMS(MySql, Oracle, Infomix, Sybase, MS Access) use **SQL** as the standard database query language. SQL is used to perform all types of data operations in RDBMS.

SQL Command

SQL defines following ways to manipulate data stored in an RDBMS.

DDL: Data Definition Language

This includes changes to the structure of the table like creation of table, altering table, deleting a table etc.

All DDL commands are auto-committed. That means it saves all the changes permanently in the database.

Command	Description

create	to create new table or database
alter	for alteration
truncate	delete data from table
drop	to drop a table
rename	to rename a table

DML: Data Manipulation Language

DML commands are used for manipulating the data stored in the table and not the table itself.

DML commands are not auto-committed. It means changes are not permanent to database, they can be rolled back.

Command	**Description**
insert	to insert a new row
update	to update existing row
delete	to delete a row
merge	merging two rows or two tables

TCL: Transaction Control Language

These commands are to keep a check on other commands and their affect on the database. These commands can annul changes made by other commands by rolling the data back to its original state. It can also make any temporary change permanent.

Command	**Description**
commit	to permanently save
rollback	to undo change
savepoint	to save temporarily

DCL: Data Control Language

Data control language are the commands to grant and take back authority from any database user.

Command	Description
grant	grant permission of right
revoke	take back permission.

DQL: Data Query Language

Data query language is used to fetch data from tables based on conditions that we can easily apply.

Command	Description
select	retrieve records from one or more table

UNIT – III

Oracle / PLSQL: Synonyms

Description

A **synonym** is an alternative name for objects such as tables, views, sequences, stored procedures, and other database objects.

You generally use synonyms when you are granting access to an object from another schema and you don't want the users to have to worry about knowing which schema owns the object.

Create Synonym (or Replace)

You may wish to create a synonym so that users do not have to prefix the table name with the schema name when using the table in a query.

Syntax

The syntax to create a synonym in Oracle is:

```
CREATE [OR REPLACE] [PUBLIC] SYNONYM [schema .] synonym_name
  FOR [schema .] object_name [@ dblink];
```

OR REPLACE

Allows you to recreate the synonym (if it already exists) without having to issue a DROP synonym command.

PUBLIC

It means that the synonym is a public synonym and is accessible to all users. Remember though that the user must first have the appropriate privileges to the object to use the synonym.

schema

The appropriate schema. If this phrase is omitted, Oracle assumes that you are referring to your own schema.

object_name

The name of the object for which you are creating the synonym. It can be one of the following:

- table
- view
- sequence
- stored procedure
- function
- package
- materialized view
- java class schema object
- user-defined object
- synonym

Example

```
CREATE PUBLIC SYNONYM suppliers
FOR app.suppliers;
```

This first CREATE SYNONYM example demonstrates how to create a synonym called *suppliers*. Now, users of other schemas can reference the table called *suppliers* without having to prefix the table name with the schema named *app*. For example:

```
SELECT *
FROM suppliers;
```

If this synonym already existed and you wanted to redefine it, you could always use the *OR REPLACE* phrase as follows:

```
CREATE OR REPLACE PUBLIC SYNONYM suppliers
FOR app.suppliers;
```

Drop synonym

Once a synonym has been created in Oracle, you might at some point need to drop the synonym.

Syntax

The syntax to drop a synonym in Oracle is:

```
DROP [PUBLIC] SYNONYM [schema .] synonym_name [force];
```

PUBLIC

Allows you to drop a public synonym. If you have specified *PUBLIC*, then you don't specify a *schema*.

force

It will force Oracle to drop the synonym even if it has dependencies. It is probably not a good idea to use *force* as it can cause invalidation of Oracle objects.

Example

```
DROP PUBLIC SYNONYM suppliers;
```

This DROP statement would drop the synonym called *suppliers* that we defined earlier.

Oracle / PLSQL: Sequences (Autonumber)

This Oracle tutorial explains how to **create and drop sequences** in Oracle with syntax and examples.

Description

In Oracle, you can create an autonumber field by using sequences. A sequence is an object in Oracle that is used to generate a number sequence. This can be useful when you need to create a unique number to act as a primary key.

Create Sequence

You may wish to create a sequence in Oracle to handle an autonumber field.

Syntax

The syntax to create a sequence in Oracle is:

```
CREATE SEQUENCE sequence_name
  MINVALUE value
  MAXVALUE value
  START WITH value
  INCREMENT BY value
  CACHE value;
```

sequence_name

The name of the sequence that you wish to create.

Example

Let's look at an example of how to create a sequence in Oracle.

For example:

```
CREATE SEQUENCE supplier_seq
  MINVALUE 1
  MAXVALUE 999999999999999999999999999
  START WITH 1
  INCREMENT BY 1
  CACHE 20;
```

This would create a sequence object called *supplier_seq*. The first sequence number that it would use is 1 and each subsequent number would increment by 1 (ie: 2,3,4,...}. It will cache up to 20 values for performance.

If you omit the MAXVALUE option, your sequence will automatically default to:

```
MAXVALUE 999999999999999999999999999
```

So you can simplify your CREATE SEQUENCE command as follows:

```
CREATE SEQUENCE supplier_seq
  MINVALUE 1
  START WITH 1
  INCREMENT BY 1
  CACHE 20;
```

Now that you've created a sequence object to simulate an autonumber field, we'll cover how to retrieve a value from this sequence object. To retrieve the next value in the sequence order, you need to use *nextval*.

Drop Sequence

Once you have created your sequence in Oracle, you might find that you need to remove it from the database.

Syntax

The syntax to a drop a sequence in Oracle is:

```
DROP SEQUENCE sequence_name;
```

sequence_name

The name of the sequence that you wish to drop.

Example

Let's look at an example of how to drop a sequence in Oracle.

For example:

```
DROP SEQUENCE supplier_seq;
```

This example would drop the sequence called *supplier_seq*.

Oracle / PLSQL: Grant/Revoke Privileges

This Oracle tutorial explains how to **grant and revoke privileges** in Oracle with syntax and examples.

Description

You can GRANT and REVOKE privileges on various database objects in Oracle. We'll first look at how to grant and revoke privileges on tables and then how to grant and revoke privileges on functions and procedures in Oracle.

Grant Privileges on Table

You can grant users various privileges to tables. These privileges can be any combination of SELECT, INSERT, UPDATE, DELETE, REFERENCES, ALTER, INDEX, or ALL.

Syntax

The syntax for granting privileges on a table in Oracle is:

```
GRANT privileges ON object TO user;
```
privileges

The privileges to assign. It can be any of the following values:

Privilege	Description
SELECT	Ability to perform SELECT statements on the table.
INSERT	Ability to perform INSERT statements on the table.
UPDATE	Ability to perform UPDATE statements on the table.
DELETE	Ability to perform DELETE statements on the table.
REFERENCES	Ability to create a constraint that refers to the table.
ALTER	Ability to perform ALTER TABLE statements to change the table definition.
INDEX	Ability to create an index on the table with the create index statement.
ALL	All privileges on table.

object

The name of the database object that you are granting privileges for. In the case of granting privileges on a table, this would be the table name.

user

The name of the user that will be granted these privileges.

Example

For example, if you wanted to grant SELECT, INSERT, UPDATE, and DELETE privileges on a table called *suppliers* to a user name *smithj*, you would run the following GRANT statement:

```
GRANT SELECT, INSERT, UPDATE, DELETE ON suppliers TO smithj;
```

You can also use the ALL keyword to indicate that you wish ALL permissions to be granted for a user named *smithj*. For example:

```
GRANT ALL ON suppliers TO smithj;
```

If you wanted to grant only SELECT access on your table to all users, you could grant the privileges to the public keyword. For example:

```
GRANT SELECT ON suppliers TO public;
```

Revoke Privileges on Table

Once you have granted privileges, you may need to revoke some or all of these privileges. To do this, you can run a revoke command. You can revoke any combination of SELECT, INSERT, UPDATE, DELETE, REFERENCES, ALTER, INDEX, or ALL.

Syntax

The syntax for revoking privileges on a table in Oracle is:

```
REVOKE privileges ON object FROM user;
```

privileges

The privileges to revoke. It can be any of the following values:

Privilege	Description
SELECT	Ability to perform SELECT statements on the table.
INSERT	Ability to perform INSERT statements on the table.
UPDATE	Ability to perform UPDATE statements on the table.
DELETE	Ability to perform DELETE statements on the table.
REFERENCES	Ability to create a constraint that refers to the table.

Privilege	Description
ALTER	Ability to perform ALTER TABLE statements to change the table definition.
INDEX	Ability to create an index on the table with the create index statement.
ALL	All privileges on table.

object

The name of the database object that you are revoking privileges for. In the case of revoking privileges on a table, this would be the table name.

user

The name of the user that will have these privileges revoked.

Example

For example, if you wanted to revoke DELETE privileges on a table called *suppliers* from a user named *anderson*, you would run the following REVOKE statement:

```
REVOKE DELETE ON suppliers FROM anderson;
```

If you wanted to revoke ALL privileges on a table for a user named *anderson*, you could use the ALL keyword as follows:

```
REVOKE ALL ON suppliers FROM anderson;
```

If you had granted ALL privileges to public (all users) on the *suppliers* table and you wanted to revoke these privileges, you could run the following REVOKE statement:

```
REVOKE ALL ON suppliers FROM public;
```

Grant Privileges on Functions/Procedures

When dealing with functions and procedures, you can grant users the ability to EXECUTE these functions and procedures.

Syntax

The syntax for granting EXECUTE privileges on a function/procedure in Oracle is:

```
GRANT EXECUTE ON object TO user;
```

EXECUTE

The ability to compile the function/procedure. The ability to execute the function/procedure directly.

object

The name of the database object that you are granting privileges for. In the case of granting EXECUTE privileges on a function or procedure, this would be the function name or the procedure name.

user

The name of the user that will be granted the EXECUTE privileges.

Example

For example, if you had a function called *Find_Value* and you wanted to grant EXECUTE access to the user named *smithj*, you would run the following GRANT statement:

```
GRANT EXECUTE ON Find_Value TO smithj;
```

If you wanted to grant ALL users the ability to EXECUTE this function, you would run the following GRANT statement:

```
GRANT EXECUTE ON Find_Value TO public;
```

Revoke Privileges on Functions/Procedures

Once you have granted EXECUTE privileges on a function or procedure, you may need to REVOKE these privileges from a user. To do this, you can execute a REVOKE command.

Syntax

The syntax for the revoking privileges on a function or procedure in Oracle is:

```
REVOKE EXECUTE ON object FROM user;
```

EXECUTE

The ability to compile the function/procedure. The ability to execute the function/procedure directly.

object

The name of the database object that you are revoking privileges for. In the case of revoking EXECUTE privileges on a function or procedure, this would be the function name or the procedure name.

user

The name of the user that will be revoked the EXECUTE privileges.

Example

Let's look at some examples of how to revoke EXECUTE privileges on a function or procedure in Oracle.

If you wanted to revoke EXECUTE privileges on a function called *Find_Value* from a user named *anderson*, you would run the following REVOKE statement:

```
REVOKE execute ON Find_Value FROM anderson;
```

If you had granted EXECUTE privileges to public (all users) on the function called *Find_Value* and you wanted to revoke these EXECUTE privileges, you could run the following REVOKE statement:

```
REVOKE EXECUTE ON Find_Value FROM public;
```

Oracle / PLSQL: Indexes

This Oracle tutorial explains how to **create, rename and drop indexes** in Oracle with syntax and examples.

What is an Index in Oracle?

An index is a performance-tuning method of allowing faster retrieval of records. An index creates an entry for each value that appears in the indexed columns. By default, Oracle creates B-tree indexes.

Create an Index

Syntax

The syntax for creating an index in Oracle/PLSQL is:

```
CREATE [UNIQUE] INDEX index_name
  ON table_name (column1, column2, ... column_n)
  [ COMPUTE STATISTICS ];
```

UNIQUE
: It indicates that the combination of values in the indexed columns must be unique.

index_name
: The name to assign to the index.

table_name
: The name of the table in which to create the index.

column1, column2, ... column_n
: The columns to use in the index.

COMPUTE STATISTICS
: It tells Oracle to collect statistics during the creation of the index. The statistics are then used by the optimizer to choose a "plan of execution" when SQL statements are executed.

Example

```
CREATE INDEX supplier_idx
  ON supplier (supplier_name);
```

In this example, we've created an index on the supplier table called supplier_idx. It consists of only one field - the supplier_name field.

We could also create an index with more than one field as in the example below:

```
CREATE INDEX supplier_idx
  ON supplier (supplier_name, city);
```

We could also choose to collect statistics upon creation of the index as follows:

```
CREATE INDEX supplier_idx
  ON supplier (supplier_name, city)
  COMPUTE STATISTICS;
```

Create a Function-Based Index

Syntax

The syntax for creating a function-based index in Oracle/PLSQL is:

```
CREATE [UNIQUE] INDEX index_name
  ON table_name (function1, function2, ... function_n)
  [ COMPUTE STATISTICS ];
```

UNIQUE

It indicates that the combination of values in the indexed columns must be unique.

index_name

The name to assign to the index.

table_name

The name of the table in which to create the index.

function1, function2, ... function_n

The functions to use in the index.

COMPUTE STATISTICS

It tells Oracle to collect statistics during the creation of the index. The statistics are then used by the optimizer to choose a "plan of execution" when SQL statements are executed.

Example

```
CREATE INDEX supplier_idx
  ON supplier (UPPER(supplier_name));
```

In this example, we've created an index based on the uppercase evaluation of the *supplier_name* field.

However, to be sure that the Oracle optimizer uses this index when executing your SQL statements, be sure that UPPER(supplier_name) does not evaluate to a NULL value. To ensure this, add **UPPER(supplier_name) IS NOT NULL** to your WHERE clause as follows:

```
SELECT supplier_id, supplier_name, UPPER(supplier_name)
FROM supplier
WHERE UPPER(supplier_name) IS NOT NULL
```

```
ORDER BY UPPER(supplier_name);
```

Rename an Index

Syntax

The syntax for renaming an index in Oracle/PLSQL is:

```
ALTER INDEX index_name
  RENAME TO new_index_name;
```

index_name
: The name of the index that you wish to rename.

new_index_name
: The new name to assign to the index.

Example

```
ALTER INDEX supplier_idx
  RENAME TO supplier_index_name;
```

In this example, we're renaming the index called *supplier_idx* to *supplier_index_name*.

Drop an Index

Syntax

The syntax for dropping an index in Oracle/PLSQL is:

```
DROP INDEX index_name;
```

index_name
: The name of the index to drop.

Example

Let's look at an example of how to drop an index in Oracle/PLSQL.

For example:

```
DROP INDEX supplier_idx;
```

In this example, we're dropping an index called supplier_idx.

Oracle / PLSQL: VIEW

This Oracle tutorial explains how to **create, update, and drop Oracle VIEWS** with syntax and examples.

What is a VIEW in Oracle?

An Oracle VIEW, in essence, is a virtual table that does not physically exist. Rather, it is created by a query joining one or more tables.

Create VIEW

Syntax

The syntax for the CREATE VIEW Statement in Oracle/PLSQL is:

```
CREATE VIEW view_name AS
  SELECT columns
  FROM tables
  [WHERE conditions];
```

view_name

The name of the Oracle VIEW that you wish to create.

WHERE conditions

Optional. The conditions that must be met for the records to be included in the VIEW.

Example

Here is an example of how to use the Oracle CREATE VIEW:

```
CREATE VIEW sup_orders AS
  SELECT suppliers.supplier_id, orders.quantity, orders.price
  FROM suppliers
  INNER JOIN orders
  ON suppliers.supplier_id = orders.supplier_id
  WHERE suppliers.supplier_name = 'Microsoft';
```

This Oracle CREATE VIEW example would create a virtual table based on the result set of the SELECT statement. You can now query the Oracle VIEW as follows:

```
SELECT *
FROM sup_orders;
```

Update VIEW

You can modify the definition of an Oracle VIEW without dropping it by using the Oracle CREATE OR REPLACE VIEW Statement.

Syntax

The syntax for the CREATE OR REPLACE VIEW Statement in Oracle/PLSQL is:

```
CREATE OR REPLACE VIEW view_name AS
  SELECT columns
  FROM table
  WHERE conditions;
```

view_name

The name of the Oracle VIEW that you wish to create or replace.

Example

Here is an example of how you would use the Oracle CREATE OR REPLACE VIEW Statement:

```
CREATE or REPLACE VIEW sup_orders AS
  SELECT suppliers.supplier_id, orders.quantity, orders.price
  FROM suppliers
  INNER JOIN orders
  ON suppliers.supplier_id = orders.supplier_id
  WHERE suppliers.supplier_name = 'Apple';
```

This Oracle CREATE OR REPLACE VIEW example would update the definition of the Oracle VIEW called *sup_orders* without dropping it. If the Oracle VIEW did not yet exist, the VIEW would merely be created for the first time.

Drop VIEW

Once an Oracle VIEW has been created, you can drop it with the Oracle DROP VIEW Statement.

Syntax

The syntax for the DROP VIEW Statement in Oracle/PLSQL is:

```
DROP VIEW view_name;
```

view_name

The name of the view that you wish to drop.

Example

Here is an example of how to use the Oracle DROP VIEW Statement:

```
DROP VIEW sup_orders;
```

This Oracle DROP VIEW example would drop/delete the Oracle VIEW called *sup_orders*.

Oracle / PLSQL: Subqueries

This Oracle tutorial explains how to use Oracle **subqueries** with syntax and examples.

What is a subquery in Oracle?

In Oracle, a subquery is a query within a query. You can create subqueries within your SQL statements. These subqueries can reside in the WHERE clause, the FROM clause, or the SELECT clause.

WHERE clause

Most often, the subquery will be found in the WHERE clause. These subqueries are also called nested subqueries.

For example:

```
SELECT *
FROM all_tables tabs
WHERE tabs.table_name IN (SELECT cols.table_name
                          FROM all_tab_columns cols
                          WHERE cols.column_name = 'SUPPLIER_ID');
```

FROM clause

A subquery can also be found in the FROM clause. These are called **inline views**.

For example:

```
SELECT suppliers.name, subquery1.total_amt
FROM suppliers,
 (SELECT supplier_id, SUM(orders.amount) AS total_amt
  FROM orders
  GROUP BY supplier_id) subquery1
WHERE subquery1.supplier_id = suppliers.supplier_id;
```

In this example, we've created a subquery in the FROM clause as follows:

```
(SELECT supplier_id, SUM(orders.amount) AS total_amt
 FROM orders
 GROUP BY supplier_id) subquery1
```

This subquery has been aliased with the name *subquery1*. This will be the name used to reference this subquery or any of its fields.

SELECT clause

A subquery can also be found in the SELECT clause.

For example:

```
SELECT tbls.owner, tbls.table_name,
```

```
(SELECT COUNT(column_name) AS total_columns
 FROM all_tab_columns cols
 WHERE cols.owner = tbls.owner
 AND cols.table_name = tbls.table_name) subquery2
```

Oracle / PLSQL: Joins

This Oracle tutorial explains how to use **JOINS** (inner and outer) in Oracle with syntax, visual illustrations, and examples.

Description

Oracle JOINS are used to retrieve data from multiple tables. An Oracle JOIN is performed whenever two or more tables are joined in a SQL statement.

There are 4 different types of Oracle joins:

- Oracle INNER JOIN (or sometimes called simple join)
- Oracle LEFT OUTER JOIN (or sometimes called LEFT JOIN)
- Oracle RIGHT OUTER JOIN (or sometimes called RIGHT JOIN)
- Oracle FULL OUTER JOIN (or sometimes called FULL JOIN)

So let's discuss Oracle JOIN syntax, look at visual illustrations of Oracle JOINS, and explore Oracle JOIN examples.

INNER JOIN (simple join)

Chances are, you've already written a statement that uses an Oracle INNER JOIN. It is the most common type of join. Oracle INNER JOINS return all rows from multiple tables where the join condition is met.

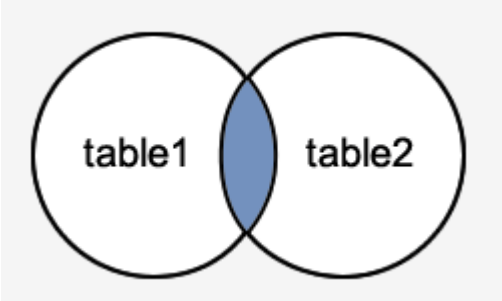
Syntax

The syntax for the INNER JOIN in Oracle/PLSQL is:

```
SELECT columns
FROM table1
INNER JOIN table2
ON table1.column = table2.column;
```

Visual Illustration

In this visual diagram, the Oracle INNER JOIN returns the shaded area:



The Oracle INNER JOIN would return the records where *table1* and *table2* intersect.

Example

Here is an example of an Oracle INNER JOIN:

```
SELECT suppliers.supplier_id, suppliers.supplier_name, orders.order_date
FROM suppliers
INNER JOIN orders
ON suppliers.supplier_id = orders.supplier_id;
```

This Oracle INNER JOIN example would return all rows from the suppliers and orders tables where there is a matching supplier_id value in both the suppliers and orders tables.

Let's look at some data to explain how the INNER JOINS work:

We have a table called *suppliers* with two fields (supplier_id and supplier_name). It contains the following data:

supplier_id	supplier_name
10000	IBM
10001	Hewlett Packard
10002	Microsoft
10003	NVIDIA

We have another table called *orders* with three fields (order_id, supplier_id, and order_date). It contains the following data:

order_id	supplier_id	order_date
500125	10000	2003/05/12
500126	10001	2003/05/13
500127	10004	2003/05/14

If we run the Oracle SELECT statement (that contains an INNER JOIN) below:

```
SELECT suppliers.supplier_id, suppliers.supplier_name, orders.order_date
FROM suppliers
INNER JOIN orders
ON suppliers.supplier_id = orders.supplier_id;
```

Our result set would look like this:

supplier_id	name	order_date
10000	IBM	2003/05/12
10001	Hewlett Packard	2003/05/13

The rows for *Microsoft* and *NVIDIA* from the supplier table would be omitted, since the supplier_id's 10002 and 10003 do not exist in both tables. The row for 500127 (order_id) from the orders table would be omitted, since the supplier_id 10004 does not exist in the suppliers table.

LEFT OUTER JOIN

Another type of join is called an Oracle LEFT OUTER JOIN. This type of join returns all rows from the LEFT-hand table specified in the ON condition and **only** those rows from the other table where the joined fields are equal (join condition is met).

Syntax

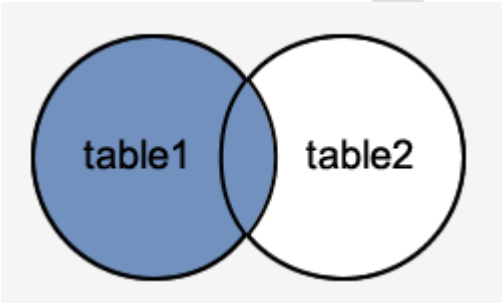
The syntax for the Oracle **LEFT OUTER JOIN** is:

```
SELECT columns
FROM table1
LEFT [OUTER] JOIN table2
ON table1.column = table2.column;
```

In some databases, the LEFT OUTER JOIN keywords are replaced with LEFT JOIN.

Visual Illustration

In this visual diagram, the Oracle LEFT OUTER JOIN returns the shaded area:



The Oracle LEFT OUTER JOIN would return the all records from *table1* and only those records from *table2* that intersect with *table1*.

Example

Here is an example of an Oracle LEFT OUTER JOIN:

```
SELECT suppliers.supplier_id, suppliers.supplier_name, orders.order_date
FROM suppliers
LEFT OUTER JOIN orders
ON suppliers.supplier_id = orders.supplier_id;
```

This LEFT OUTER JOIN example would return all rows from the suppliers table and only those rows from the orders table where the joined fields are equal.

If a supplier_id value in the suppliers table does not exist in the orders table, all fields in the orders table will display as <null> in the result set.

Let's look at some data to explain how LEFT OUTER JOINS work:

We have a table called *suppliers* with two fields (supplier_id and supplier_name). It contains the following data:

supplier_id	supplier_name
10000	IBM
10001	Hewlett Packard
10002	Microsoft
10003	NVIDIA

We have a second table called *orders* with three fields (order_id, supplier_id, and order_date). It contains the following data:

order_id	supplier_id	order_date
500125	10000	2003/05/12
500126	10001	2003/05/13

If we run the SELECT statement (that contains a LEFT OUTER JOIN) below:

```
SELECT suppliers.supplier_id, suppliers.supplier_name, orders.order_date
FROM suppliers
LEFT OUTER JOIN orders
ON suppliers.supplier_id = orders.supplier_id;
```

Our result set would look like this:

supplier_id	supplier_name	order_date
10000	IBM	2003/05/12
10001	Hewlett Packard	2003/05/13
10002	Microsoft	<null>
10003	NVIDIA	<null>

The rows for *Microsoft* and *NVIDIA* would be included because a LEFT OUTER JOIN was used. However, you will notice that the order_date field for those records contains a <null> value.

RIGHT OUTER JOIN

Another type of join is called an Oracle RIGHT OUTER JOIN. This type of join returns all rows from the RIGHT-hand table specified in the ON condition and **only** those rows from the other table where the joined fields are equal (join condition is met).

Syntax

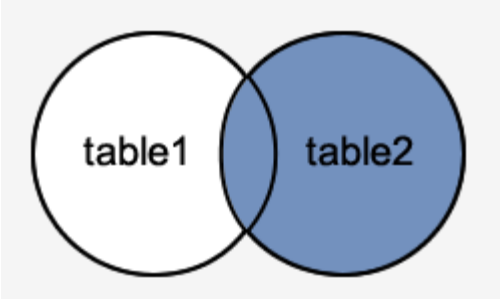
The syntax for the Oracle **RIGHT OUTER JOIN** is:

```
SELECT columns
FROM table1
RIGHT [OUTER] JOIN table2
ON table1.column = table2.column;
```

In some databases, the RIGHT OUTER JOIN keywords are replaced with RIGHT JOIN.

Visual Illustration

In this visual diagram, the Oracle RIGHT OUTER JOIN returns the shaded area:



The Oracle RIGHT OUTER JOIN would return the all records from *table2* and only those records from *table1* that intersect with *table2*.

Example

Here is an example of an Oracle RIGHT OUTER JOIN:

```
SELECT orders.order_id, orders.order_date, suppliers.supplier_name
FROM suppliers
RIGHT OUTER JOIN orders
ON suppliers.supplier_id = orders.supplier_id;
```

This RIGHT OUTER JOIN example would return all rows from the orders table and only those rows from the suppliers table where the joined fields are equal.

If a supplier_id value in the orders table does not exist in the suppliers table, all fields in the suppliers table will display as <null> in the result set.

Let's look at some data to explain how RIGHT OUTER JOINS work:

We have a table called *suppliers* with two fields (supplier_id and supplier_name). It contains the following data:

supplier_id	supplier_name
10000	Apple
10001	Google

We have a second table called *orders* with three fields (order_id, supplier_id, and order_date). It contains the following data:

order_id	supplier_id	order_date
500125	10000	2013/08/12
500126	10001	2013/08/13
500127	10002	2013/08/14

If we run the SELECT statement (that contains a RIGHT OUTER JOIN) below:

```
SELECT orders.order_id, orders.order_date, suppliers.supplier_name
FROM suppliers
RIGHT OUTER JOIN orders
ON suppliers.supplier_id = orders.supplier_id;
```

Our result set would look like this:

order_id	order_date	supplier_name
500125	2013/08/12	Apple
500126	2013/08/13	Google
500127	2013/08/14	<null>

The row for *500127* (order_id) would be included because a RIGHT OUTER JOIN was used.

FULL OUTER JOIN

Another type of join is called an Oracle FULL OUTER JOIN. This type of join returns all rows from the LEFT-hand table and RIGHT-hand table with nulls in place where the join condition is not met.

Syntax

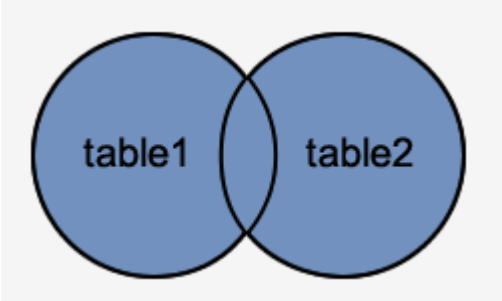
The syntax for the Oracle **FULL OUTER JOIN** is:

```
SELECT columns
FROM table1
FULL [OUTER] JOIN table2
ON table1.column = table2.column;
```

In some databases, the FULL OUTER JOIN keywords are replaced with FULL JOIN.

Visual Illustration

In this visual diagram, the Oracle FULL OUTER JOIN returns the shaded area:



The Oracle FULL OUTER JOIN would return the all records from both *table1* and *table2*.

Example

Here is an example of an Oracle FULL OUTER JOIN:

```
SELECT suppliers.supplier_id, suppliers.supplier_name, orders.order_date
FROM suppliers
FULL OUTER JOIN orders
ON suppliers.supplier_id = orders.supplier_id;
```

This FULL OUTER JOIN example would return all rows from the suppliers table and all rows from the orders table and whenever the join condition is not met, <nulls> would be extended to those fields in the result set.

If a supplier_id value in the suppliers table does not exist in the orders table, all fields in the orders table will display as <null> in the result set. If a supplier_id value in the orders table does not exist in the suppliers table, all fields in the suppliers table will display as <null> in the result set.

Let's look at some data to explain how FULL OUTER JOINS work:

We have a table called *suppliers* with two fields (supplier_id and supplier_name). It contains the following data:

supplier_id	supplier_name
10000	IBM
10001	Hewlett Packard
10002	Microsoft
10003	NVIDIA

We have a second table called *orders* with three fields (order_id, supplier_id, and order_date). It contains the following data:

order_id	supplier_id	order_date
500125	10000	2013/08/12
500126	10001	2013/08/13
500127	10004	2013/08/14

If we run the SELECT statement (that contains a FULL OUTER JOIN) below:

```
SELECT suppliers.supplier_id, suppliers.supplier_name, orders.order_date
FROM suppliers
FULL OUTER JOIN orders
ON suppliers.supplier_id = orders.supplier_id;
```

Our result set would look like this:

supplier_id	supplier_name	order_date
10000	IBM	2013/08/12
10001	Hewlett Packard	2013/08/13
10002	Microsoft	<null>
10003	NVIDIA	<null>
<null>	<null>	2013/08/14

The rows for *Microsoft* and *NVIDIA* would be included because a FULL OUTER JOIN was used. However, you will notice that the order_date field for those records contains a <null> value.

UNIT - IV

What is PL/SQL

PL/SQL is a Procedural Language (PL) that extends the Structured Query Language (SQL). If you have been programming Pascal or Ada, you will find much familiar syntax in PL/SQL.

PL/SQL Advantages

PL/SQL is a highly structured language

PL/SQL provides a very expressive syntax that makes it easy for anyone who wants to learn PL/SQL. If you are programming in other languages, you can get familiar with PL/SQL very quickly and understand the intent of the code without difficulty.

PL/SQL language features include the following elements:

- Variables
- Block structure, nested block structure
- Conditional and sequential statements: IF, CASE, GOTO, CONTINUE and NULL
- Loop statements: WHILE loop, FOR loop
- Exception and error handling
- Data types: string, numbers, date & timestamp, Boolean, and LOB
- Record
- Collection
- Cursor
- Procedures, functions, packages
- Object-orientation features
- Dynamic SQL and dynamic PL/SQL

PL/SQL is a portable and standard language for Oracle development

Once you develop a PL/SQL program in an Oracle Database, you can move it to the other Oracle Databases without changes, with the assumption that the versions of Oracle database are compatible.

PL/SQL is an embedded language

PL/SQL programs such as functions and procedures are stored in Oracle database in compiled form. This allows applications or users to share the same functionality stored in Oracle database.

PL/SQL also allows you to define triggers that can be invoked automatically in response to particular events in associated tables.

PL/SQL is a high-performance language inside Oracle Databases

Oracle adds many enhancements to the PL/SQL to make it more efficient to interact with Oracle databases.

Fundamentals of the PL/SQL Language

Like other programming languages, PL/SQL has a character set, reserved words, punctuation, datatypes, and fixed syntax rules.

Character Sets and Lexical Units

PL/SQL programs are written as lines of text using a specific set of characters:

■ Upper- and lower-case letters A .. Z and a .. z

■ Numerals 0 .. 9

■ Symbols ( ) + – * / < > = ! ~ ^ ; : . ‘ @ % , ” # $ & _ | { } ? [ ]

■ Tabs, spaces, and carriage returns

PL/SQL keywords are not case-sensitive, so lower-case letters are equivalent to corresponding upper-case letters except within string and character literals.

A line of PL/SQL text contains groups of characters known as lexical units:

■ **Delimiters (simple and compound symbols)** – A delimiter is a simple or compound symbol that has a special meaning to PL/SQL. For example, you use delimiters to represent arithmetic operations such as addition and subtraction.

■ **Identifiers(which include reserved words)** – We use identifiers to name PL/SQL program items and units, which include constants, variables, exceptions, cursors, cursor variables, subprograms, and packages.

■ **Literals** – A literal is an explicit numeric, character, string, or BOOLEAN value not represented by an identifier.

■ **Comments** – The PL/SQL compiler ignores comments, but you should not. Adding comments to your program promotes readability and aids understanding. Generally, you use comments to describe the purpose and use of each code segment. PL/SQL supports two comment styles: single-line and multi-line.

Declarations

You can declare variables and constants in the declarative part of any PL/SQL block, subprogram, or package. Declarations allocate storage space for a value, specify its datatype, and name the storage location so that you can reference it.

Some examples follow:

DECLARE

birthday DATE;

emp_count SMALLINT := 0;

Constants – To declare a constant, put the keyword CONSTANT before the type specifier.

Example –

DECLARE

credit_limit CONSTANT REAL := 5000.00;

Using DEFAULT

You can use the keyword DEFAULT instead of the assignment operator to initialize variables. For example, the declaration

blood_type CHAR := 'O';

can be rewritten as follows:

blood_type CHAR DEFAULT 'O';

Using NOT NULL –

Besides assigning an initial value, declarations can impose the NOT NULL constraint:

DECLARE

acct_id INTEGER(4) NOT NULL := 9999;

Using the %TYPE Attribute

The %TYPE attribute provides the datatype of a variable or database column. This is particularly useful when declaring variables that will hold database values. For example, assume there is a column named last_name in a table named employees.

To declare a variable named v_last_name that has the same datatype as column title, use dot notation and the %TYPE attribute, as follows:

v_last_name employees.last_name%TYPE;

Using the %ROWTYPE Attribute

The %ROWTYPE attribute provides a record type that represents a row in a table or view. Columns in a row and corresponding fields in a record have the same names and datatypes. However, fields in a %ROWTYPE record do not inherit constraints, such as the NOT NULL or check constraint, or default values.

DECLARE

dept_rec departments%ROWTYPE; -- declare record variable

- **Restrictions on Declarations**

PL/SQL does not allow forward references. You must declare a variable or constant before referencing it in other statements, including other declarative statements.

DECLARE

-- Multiple declarations not allowed.

-- i, j, k, l SMALLINT;

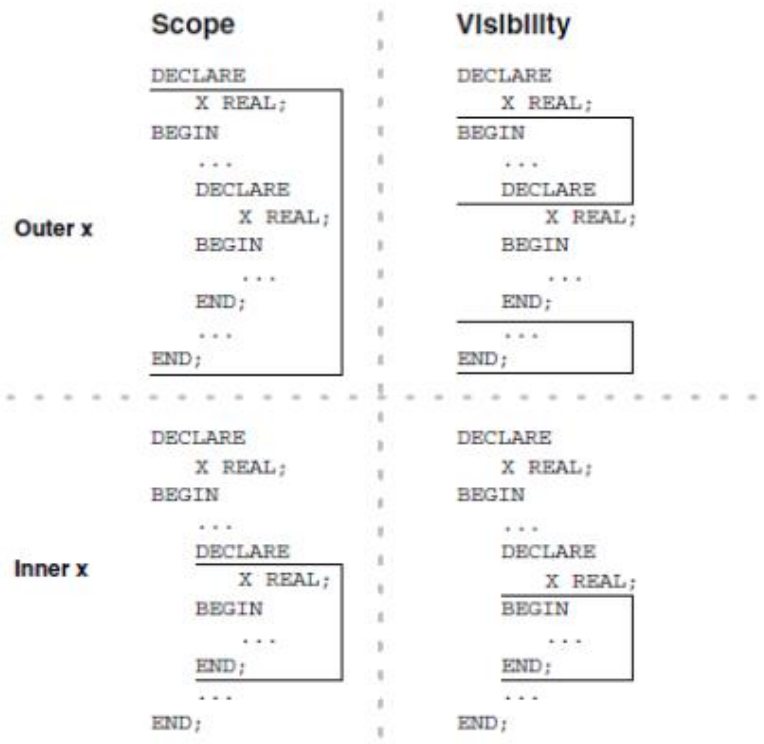
-- Instead, declare each separately.

i SMALLINT;

j SMALLINT;

-- To save space, you can declare more than one on a line.

k SMALLINT; l SMALLINT;



PL/SQL Expressions and Comparisons

Expressions are constructed using operands and operators. An operand is a variable, constant, literal, or function call that contributes a value to an expression. An example of a simple arithmetic expression follows:

-X / 2 + 3

Unary operators such as the negation operator (-) operate on one operand; binary operators such as the division operator (/) operate on two operands. PL/SQL has no ternary operators.

Table: Order of Operations

Operator	Operation
**	exponentiation
+, -	identity, negation
*, /	multiplication, division
+, -, ||	addition, subtraction, concatenation
=, <, >, <=, >=, <>, !=, ~=, ^=, IS NULL, LIKE, BETWEEN, IN	comparison
NOT	logical negation
AND	conjunction
OR	inclusion

Logical Operators

The logical operators AND, OR, and NOT.AND and OR are binary operators; NOT is a unary operator.

- **IS NULL Operator**

The IS NULL operator returns the BOOLEAN value TRUE if its operand is null or FALSE if it is not null. Comparisons involving nulls always yield NULL. Test whether a value is null as follows:

IF variable IS NULL THEN ...

- **LIKE Operator**

You use the LIKE operator to compare a character, string, or CLOB value to a pattern. Case is significant. LIKE returns the BOOLEAN value TRUE if the patterns match or FALSE if they do not match.

The patterns matched by LIKE can include two special-purpose characters called wildcards. An underscore (_) matches exactly one character; a percent sign (%) matches zero or more characters. For example, if the value of last_name is 'JOHNSON', the following expression is true:

last_name LIKE 'J%S_N'

- **BETWEEN Operator**

The BETWEEN operator tests whether a value lies in a specified range or not. It means “greater than or equal to low value and less than or equal to high value.” For example, the following expression is false:

45 BETWEEN 38 AND 44

IN Operator

The IN operator tests set membership. It means “equal to any member of.” The set can contain nulls, but they are ignored. For example, the following expression tests whether a value is part of a set of values:

letter IN ('a','b','c')

Concatenation Operator

Double vertical bars (||) serve as the concatenation operator, which appends one string (CHAR, VARCHAR2, CLOB, or the equivalent Unicode-enabled type) to another.

For example, the expression

'suit' || 'case'

returns the following value:

'suitcase'

BOOLEAN Expressions

BOOLEAN expressions consist of simple or complex expressions separated by relational operators. Often, BOOLEAN expressions are connected by the logical operators AND, OR, and NOT. A BOOLEAN expression always yields TRUE, FALSE, or NULL.

There are three kinds of BOOLEAN expressions: arithmetic, character, and date.

CASE Expressions

There are two types of expressions used in CASE statements: simple and searched. These expressions correspond to the type of CASE statement in which they are used.

- **Simple CASE expression**

A simple CASE expression selects a result from one or more alternatives, and returns the result. Although it contains a block that might stretch over several lines, it really is an expression that forms part of a larger statement, such as an assignment or a procedure call. The CASE expression uses a selector, an expression whose value determines which alternative to return.

- **Searched CASE Expression**

A searched CASE expression lets you test different conditions instead of comparing a single expression to various values. A searched CASE expression has no selector. Each WHEN clause contains a search condition that yields a BOOLEAN value, so you can test different variables or multiple conditions in a single WHEN clause.

PL/SQL Datatypes

Predefined PL/SQL Datatypes

Predefined PL/SQL datatypes are grouped into composite, LOB, reference, and scalar type categories.

- A composite type has internal components that can be manipulated individually, such as the elements of an array, record, or table.

- A LOB type holds values, called lob locators, that specify the location of large objects, such as text blocks or graphic images, that are stored separately from other database data. LOB types include BFILE, BLOB, CLOB, and NCLOB.
- A reference type holds values, called pointers, that designate other program items. These types include REF CURSORS and REFs to object types.
- A scalar type has no internal components. It holds a single value, such as a number or character string. The scalar types fall into four families, which store number, character, Boolean, and date/time data. The scalar families with their datatypes are:

1. PL/SQL Number Types

BINARY_DOUBLE, BINARY_FLOAT, BINARY_INTEGER, DEC, DECIMAL, DOUBLE PRECISION, FLOAT, INT, INTEGER, NATURAL, NATURALN, NUMBER, NUMERIC, PLS_INTEGER, POSITIVE, POSITIVEN, REAL, SIGNTYPE, SMALLINT

2. PL/SQL Character and String Types and PL/SQL National Character Types

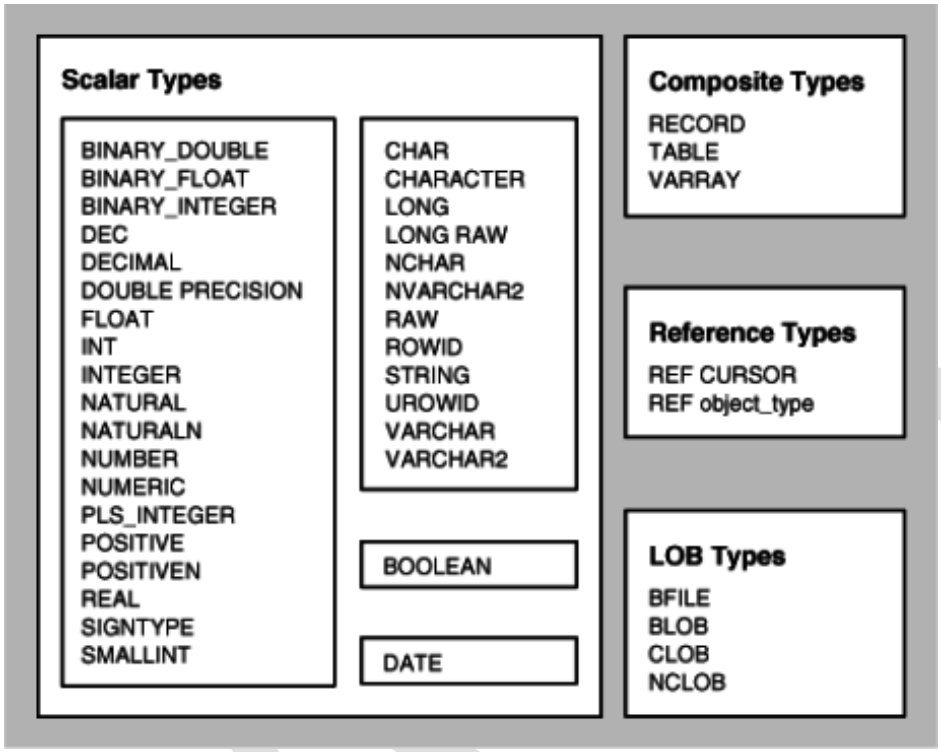
CHAR, CHARACTER, LONG, LONG RAW, NCHAR, NVARCHAR2, RAW, ROWID, STRING, UROWID, VARCHAR, VARCHAR2 Note that the LONG and LONG RAW datatypes are supported only for backward compatibility Information.

3. PL/SQL Boolean Types

BOOLEAN

4. PL/SQL Date, Time, and Interval Types

DATE, TIMESTAMP, TIMESTAMP WITH TIMEZONE, TIMESTAMP WITH LOCAL TIMEZONE, INTERVAL YEAR TO MONTH, INTERVAL DAY TO SECOND



Converting PL/SQL Datatypes

Sometimes it is necessary to convert a value from one datatype to another. For example, to use a DATE value in a report, you must convert it to a character string. PL/SQL supports both explicit and implicit (automatic) datatype conversion.

- **Explicit Conversion**

Using explicit conversions, particularly when passing parameters to subprograms, can avoid unexpected errors or wrong results. For example, the TO_CHAR function lets you specify the format for a DATE value, rather than relying on language settings in the database. Including an arithmetic expression among strings being concatenated with the || operator can produce an error unless you put parentheses or a call to TO_CHAR around the entire arithmetic expression.

- **Implicit Conversion**

When it makes sense, PL/SQL can convert the data type of a value implicitly. For example, you can pass a numeric literal to a subprogram that expects a string value, and the subprogram receives the string representation of the number.

PL/SQL Variables

These are placeholders that store the values that can change through the PL/SQL Block.

General Syntax to declare a variable is

variable_name datatype [NOT NULL := value];

variable_name is the name of the variable.

datatype is a valid PL/SQL datatype.

NOT NULL is an optional specification on the variable.

value or DEFAULT *value*is also an optional specification, where you can initialize a variable.

Each variable declaration is a separate statement and must be terminated by a semicolon.

For example, if you want to store the current salary of an employee, you can use a variable.

DECLARE

salary number (6);

* "salary" is a variable of datatype number and of length 6.

When a variable is specified as NOT NULL, you must initialize the variable when it is declared.

For example: The below example declares two variables, one of which is a not null.

DECLARE
salary number(4);
dept varchar2(10) NOT NULL := "HR Dept";

The value of a variable can change in the execution or exception section of the PL/SQL Block. We can assign values to variables in the two ways given below.

1) We can directly assign values to variables.
The General Syntax is:

variable_name:= value;

2) We can assign values to variables directly from the database columns by using a SELECT.. INTO statement. The General Syntax is:

SELECT column_name

```
INTO variable_name
FROM table_name
[WHERE condition];
```

Example: The below program will get the salary of an employee with id '1116' and display it on the screen.

```
DECLARE
 var_salary number(6);
 var_emp_id number(6) = 1116;
BEGIN
 SELECT salary
 INTO var_salary
 FROM employee
 WHERE emp_id = var_emp_id;
 dbms_output.put_line(var_salary);
 dbms_output.put_line('The employee '
          || var_emp_id || ' has  salary  ' || var_salary);
END;
/
```

Scope of PS/SQL Variables

PL/SQL allows the nesting of Blocks within Blocks i.e, the Execution section of an outer block can contain inner blocks. Therefore, a variable which is accessible to an outer Block is also accessible to all nested inner Blocks. The variables declared in the inner blocks are not accessible to outer blocks. Based on their declaration we can classify variables into two types.

- *Local* variables - These are declared in a inner block and cannot be referenced by outside Blocks.
- *Global* variables - These are declared in a outer block and can be referenced by its itself and by its inner blocks.

For Example: In the below example we are creating two variables in the outer block and assigning thier product to the third variable created in the inner block. The variable 'var_mult' is declared in the inner block, so cannot be accessed in the outer block i.e. it cannot be accessed after line 11. The variables 'var_num1' and 'var_num2' can be accessed anywhere in the block.

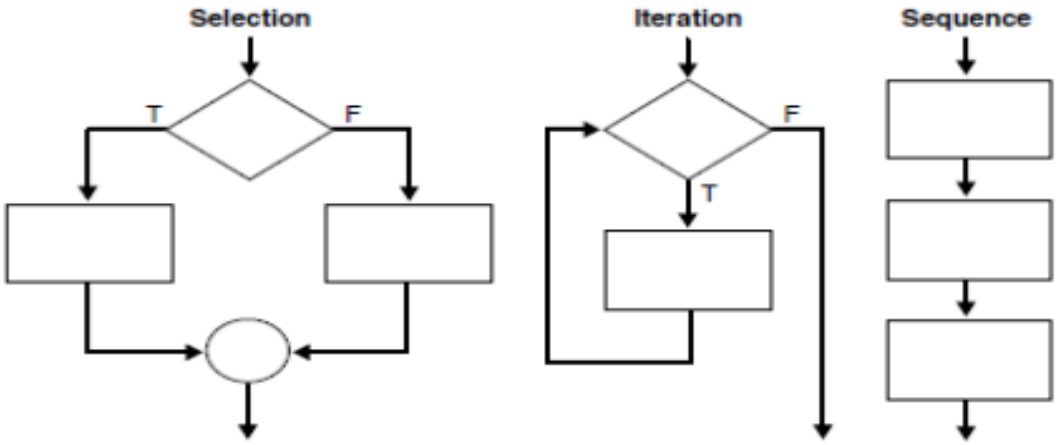
```
1> DECLARE
2>  var_num1 number;
3>  var_num2 number;
4> BEGIN
5>  var_num1 := 100;
6>  var_num2 := 200;
7>  DECLARE
8>   var_mult number;
```

```
9>   BEGIN
10>    var_mult := var_num1 * var_num2;
11>   END;
12> END;
13> /
```

PL/SQL Control Structures

PL/SQL Control Structures

Procedural computer programs use the basic control structures.



- The selection structure tests a condition, then executes one sequence of statements instead of another, depending on whether the condition is true or false. A condition is any variable or expression that returns a BOOLEAN value (TRUE or FALSE).
- The iteration structure executes a sequence of statements repeatedly as long as a condition holds true.
- The sequence structure simply executes a sequence of statements in the order in which they occur.

Testing Conditions: IF and CASE Statements

The IF statement executes a sequence of statements depending on the value of a condition. There are three forms of IF statements: IF-THEN, IF-THEN-ELSE, and IF-THEN-ELSIF.

The CASE statement is a compact way to evaluate a single condition and choose between many alternative actions. It makes sense to use CASE when there are three or more alternatives to choose from.

Using the IF-THEN Statement

The simplest form of IF statement associates a condition with a sequence ofstatements enclosed by the keywords THEN and END IF (not ENDIF)

The sequence of statements is executed only if the condition is TRUE. If the condition is FALSE or NULL, the IF statement does nothing. In either case, control passes to the next statement.

Example: Using a Simple IF-THEN Statement

```
DECLARE

sales NUMBER(8,2) := 10100;

quota NUMBER(8,2) := 10000;

bonus NUMBER(6,2);

emp_id NUMBER(6) := 120;

BEGIN

IF sales > (quota + 200) THEN

bonus := (sales - quota)/4;

UPDATE employees SET salary = salary + bonus WHERE employee_id = emp_id;

END IF;

END;

/
```

Using CASE Statements

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.

Example: Using the CASE-WHEN Statement

```
DECLARE
grade CHAR(1);
BEGIN
grade := 'B';
CASE grade
WHEN 'A' THEN DBMS_OUTPUT.PUT_LINE('Excellent');
WHEN 'B' THEN DBMS_OUTPUT.PUT_LINE('Very Good');
WHEN 'C' THEN DBMS_OUTPUT.PUT_LINE('Good');
WHEN 'D' THEN DBMS_OUTPUT.PUT_LINE('Fair');
WHEN 'F' THEN DBMS_OUTPUT.PUT_LINE('Poor');
ELSE DBMS_OUTPUT.PUT_LINE('No such grade');
END CASE;
END;
/
```

Controlling Loop Iterations: LOOP and EXIT Statements

LOOP statements execute a sequence of statements multiple times. There are three forms of LOOP statements: LOOP, WHILE-LOOP, and FOR-LOOP.

- **Using the LOOP Statement**

The simplest form of LOOP statement is the basic loop, which encloses a sequence of statements between the keywords LOOP and END LOOP, as follows:

```
LOOP
sequence_of_statements
END LOOP;
```

With each iteration of the loop, the sequence of statements is executed, then control resumes at the top of the loop. You use an EXIT statement to stop looping and prevent an infinite loop. You can place one or more EXIT statements anywhere inside a loop, but not outside a loop. There are two forms of EXIT statements: EXIT and

EXIT-WHEN.

- **Using the EXIT Statement**

The EXIT statement forces a loop to complete unconditionally. When an EXIT statement is encountered, the loop completes immediately and control passes to the next statement.

- **Using the EXIT-WHEN Statement**

The EXIT-WHEN statement lets a loop complete conditionally. When the EXIT statement is encountered, the condition in the WHEN clause is evaluated. If the condition is true, the loop completes and control passes to the next statement after the loop.

- **Labeling a PL/SQL Loop**

Like PL/SQL blocks, loops can be labeled. The optional label, an undeclared identifier enclosed by double angle brackets, must appear at the beginning of the LOOP statement. The label name can also appear at the end of the LOOP statement. When you nest labeled loops, use ending label names to improve readability.

- **Using the WHILE-LOOP Statement**

The WHILE-LOOP statement executes the statements in the loop body as long as a condition is true:

WHILE condition LOOP

sequence_of_statements

END LOOP;

Using the FOR-LOOP Statement

Simple FOR loops iterate over a specified range of integers. The number of iterations is known before the loop is entered. A double dot (..) serves as the range operator. The range is evaluated when the FOR loop is first entered and is never re-evaluated. If the lower bound equals the higher bound, the loop body is executed once.

Example: Using a Simple FOR..LOOP Statement

```
DECLARE

p NUMBER := 0;

BEGIN

FOR k IN 1..500 LOOP -- calculate pi with 500 terms

p := p + ( ( (-1) ** (k + 1) ) / ((2 * k) - 1) );

END LOOP;

p := 4 * p;

DBMS_OUTPUT.PUT_LINE( 'pi is approximately : ' || p ); -- print result

END;

/
```

Sequential Control: GOTO and NULL Statements

The GOTO statement is seldom needed. Occasionally, it can simplify logic enough to warrant its use. The NULL statement can improve readability by making the meaning and action of conditional statements clear.

Overuse of GOTO statements can result in code that is hard to understand and maintain. Use GOTO statements sparingly. For example, to branch from a deeply nested structure to an error-handling routine, raise an exception rather than use a GOTO statement.

- **Using the GOTO Statement**

The GOTO statement branches to a label unconditionally. The label must be unique within its scope and must precede an executable statement or a PL/SQL block. When executed, the GOTO statement transfers control to the labeled statement or block. The labeled statement or block can be down or up in the sequence of statements.

Example : Using a Simple GOTO Statement

```
DECLARE

p VARCHAR2(30);

n PLS_INTEGER := 37; -- test any integer > 2 for prime
```

```
BEGIN

FOR j in 2..ROUND(SQRT(n)) LOOP

IF n MOD j = 0 THEN -- test for prime

p := ' is not a prime number'; -- not a prime number

GOTO print_now;

END IF;

END LOOP;

p := ' is a prime number';

<<print_now>>

DBMS_OUTPUT.PUT_LINE(TO_CHAR(n) || p);

END;

/
```

UNIT - V

PL/SQL Cursors And Exceptions :

Understand Cursors

A **cursor** is a mechanism by which you can assign a name to a "select statement" and manipulate the information within that SQL statement. In other words, a cursor is a SELECT statement that is defined within the *declaration* section of your PLSQL code. You will take a look at three different syntaxes for cursors. There are two types of cursors: ***Implicit*** *and* ***Explicit****.*

- An Implicit cursor is used for all other SQL statements. Implicit Cursors gives less programmatic control.
- In explicit cursor the cursor name is explicitly attached to a select statement

The four PL/SQL steps necessary for explicit cursor processing are as follows:

1. **Declare the cursor**
2. **Open the cursor**
3. **Fetch the results into PL/SQL variables**
4. **Close the cursor**

Declare the cursor

To use a cursor, it must be declared first.

Syntax

Code:

```
CURSOR cursor_name IS SELECT_statement;
```

A cursor without parameters

Code:

```
CURSOR comp IS SELECT compid FROM company;
```

A cursor with parameters

Code:

```
CURSOR comp (mcompid IN NUMBER) IS SELECT name FROM
company WHERE compid = mcomid;
```

Open Cursors:

Once you have declared your cursor, the next step is to open the cursor
The basic syntax to OPEN the cursor is as follows:

Code:

```
OPEN cursor_name;
```

For example, you could open a cursor called c1 with the following command:

Code:

```
OPEN c1;
```

While opening a cursor:

- The values of the bind variables are examined
- Based on the bind variable the active set is determined
- The active set pointer is set to the first row.

Following is a function that demonstrates how to use the OPEN statement:
Code:

```
CREATE OR REPLACE Function FindCourse
( name_in IN varchar2 )
 RETURN number
IS
cnumber number;
CURSOR c1
IS
SELECT course_number from courses_tbl where course_name = name_in;
 BEGIN
open c1;
 fetch c1 into cnumber;
if c1%notfound then
 cnumber := 9999;
 end if;
close c1;
RETURN cnumber;
 END;
```

Fetch Cursor

The purpose of using a cursor, in most cases, is to retrieve the rows from your cursor so that some type of operation can be performed on the data. After declaring and opening your cursor, the next step is to FETCH the rows from your cursor.
Fetching a cursor has two forms:
Code:

```
FETCH cursor_name INTO list_of_variables;
```

Or
Code:

```
FETCH cursor_name INTO PL/SQL_record;
```

- After each FETCH, the active set pointer is increased to next row.
- Thus, each FETCH will return successive rows in the active set, until the entire set is returned.
- The %NOTFOUND attribute is used to determine when the active set has been retrieved.

The basic syntax for a FETCH statement is:
Code:

```
FETCH cursor_name INTO <list of variables>;
```

For example, you could have a cursor defined as:
Code:

```
CURSOR c1 IS SELECT course_number from courses_tbl where course_name = name_in;
```

The command that would be used to fetch the data from this cursor is:

Close Cursor

The final step of working with cursors is to close the cursor once you have finished using it. The basic syntax to CLOSE the cursor is:

Code:

```
CLOSE cursor_name;
```

For example, you could close a cursor called c1 with the following command:

Code:

```
CLOSE c1;
```

Understanding Exceptions

What is an Exception?

- Exceptions are errors raised whenever there is any in a particular PL/SQL block. This causes a termination in the program by Oracle. Control then is transferred to the separate exception section of the program, if one exists, to handle the exception.
- Oracle raises ERRORS whenever any abnormal situation arises in a PL/SQL block and performs an illegal termination of the execution of the program.
- PL/SQL traps and responds to errors using architecture of exception handler.
- Occurrence of any error in PL/SQL, whether a system error or an application error, an exception is rose.
- This halts the processing in the current PL/SQL block's execution and control is transferred to the separate exception section of the program, if one exists, to handle the exception.
- The control never returns to that block after you finish handling the exception. Instead, control is passed to the enclosing block, if any.
- When an exception is raised, control passes to the exception section of the block. The exception section consists of handlers for all the exceptions.

Code:

```
EXCEPTION
WHEN exception_name THEN
sequence_of_statements1;
WHEN exception_name THEN
sequence_of_statements1;
END;
```

Types of Exceptions:

1. **Predefined Exception**
2. **User Defined Exception**

Predefined Exceptions

Some exceptions are already defined in Oracle called the pre-defined exception. Mostly they are generated with the SELECT statement. They are raised implicitly at runtime. Every exception is associated with an error code. These exceptions are already defined in the STANDARD package (An Oracle supplied package).

Oracle Exception Name	**Oracle Error**	**Explanation**
DUP_VAL_ON_INDEX	ORA-00001	You tried to execute an INSERT or UPDATE statement that has created a duplicate value in a field restricted by a unique index.
TIMEOUT_ON_RESOURCE	ORA-00051	You were waiting for a resource and you timed out.
TRANSACTION_BACKED_OUT	ORA-00061	The remote portion of a transaction has rolled back.
INVALID_CURSOR	ORA-01001	You tried to reference a cursor that does not yet exist. This may have happened because you have executed a FETCH cursor or CLOSE cursor before opening the cursor.
NOT_LOGGED_ON	ORA-01012	You tried to execute a call to Oracle before logging in.
LOGIN_DENIED	ORA-01017	You tried to log into Oracle with an invalid username or password combination.
NO_DATA_FOUND	ORA-01403	You tried one of the following: 1. You executed a SELECT INTO statement and no rows were returned. 2. You referenced an uninitialized row in a table. 3. You read past the end of file with the UTL_FILE package.
TOO_MANY_ROWS	ORA-01422	You tried to execute a SELECT INTO statement

		and more than one row was returned.
ZERO_DIVIDE	ORA-01476	You tried to divide a number by zero.

Exception functions:
SQLCODE: Returns the numeric value for the error code.
SQLERRM: Returns the message associated with the error number.

Example:
Code:

```
DECLARE
v_err_code NUMBER;
v_err_text VARCHAR2(255);
v_product_name product_dim.product_name%type;
BEGIN
SELECT product_name into v_product_name
FROM product_dim
WHERE product_id=&input_product_id;
dbms_output.put_line(v_product_name);
EXCEPTION
WHEN NO DATA FOUND THEN
v_err_code:=SQLCODE;
v_err_text:=SQLERM;
insert into errors values (v_err_code,v_err_text)
commit;
END:
```

User Defined Exception

Sometimes, it is necessary for programmers to name and trap their own exceptions - ones that aren't defined already by PL/SQL. These are called Named Programmer or User-Defined Exceptions.

The syntax for the named programmer-defined exception in a procedure is as follows:

Code:

```
CREATE [OR REPLACE] PROCEDURE procedure_name [ (parameter [,parameter]) ] IS
[declaration_section]
exception_name EXCEPTION;
BEGIN executable_section
RAISE exception_name ;
EXCEPTION WHEN exception_name THEN [statements]
WHEN OTHERS THEN [statements]
END [procedure_name];
```

The syntax for the named **programmer-defined exception** in a function is:

Code:

```
CREATE [OR REPLACE] FUNCTION function_name [(parameter [, parameter])] RETURN
return_datatype IS | AS [declaration_section]
exception_name EXCEPTION;
BEGIN executable_section
RAISE exception_name ;
EXCEPTION WHEN exception_name THEN [statements]
WHEN OTHERS THEN [statements]
END [function_name];
```

Here is an example of a procedure that uses a named **programmer-defined exception**:

Code:

```
CREATE OR REPLACE PROCEDURE add_new_order (order_id_in IN NUMBER, sales_in IN
NUMBER) IS no_sales EXCEPTION;
BEGIN IF sales_in = 0 THEN RAISE no_sales;
ELSE INSERT INTO orders (order_id, total_sales ) VALUES ( order_id_in, sales_in
); END IF;
EXCEPTION WHEN no_sales THEN raise_application_error (-20001,'You must have
sales in order to submit the order.');
WHEN OTHERS THEN raise_application_error (-20002,'An error has occurred
inserting an order.');
END;
```

Syntax:

Code:

```
RAISE_APPLICATION_ERROR (error_number in NUMBER, error_msg in VARCHAR2);
```

Example:

Code:

```
CREATE OR REPLACE PROCEDURE sp_addproduct(
IN V_prdouct_code Varchar2(100);
IN V_prdouct_name Varchar2(100);) AS
BEGIN
UPDATE product_dim
SET product_name=V_prdouct_name
WHERE product_key=V_prdouct_code
IF SQL%NOTFOUND THEN
RAISE_APPLICATION_ERROR(-20202,'This is not a valid product');
END IF;
COMMIT;
END sp_addproduct;
```

Oracle / PLSQL: Procedures

This Oracle tutorial explains how to **create and drop procedures** in Oracle/PLSQL with syntax and examples.

Create Procedure

Just as you can in other languages, you can create your own procedures in Oracle.

Syntax

The syntax to create a procedure in Oracle is:

```
CREATE [OR REPLACE] PROCEDURE procedure_name
    [ (parameter [,parameter]) ]

IS
    [declaration_section]

BEGIN
    executable_section

[EXCEPTION
    exception_section]

END [procedure_name];
```

When you create a procedure or function, you may define parameters. There are three types of parameters that can be declared:

1. **IN** - The parameter can be referenced by the procedure or function. The value of the parameter can not be overwritten by the procedure or function.
2. **OUT** - The parameter can not be referenced by the procedure or function, but the value of the parameter can be overwritten by the procedure or function.
3. **IN OUT** - The parameter can be referenced by the procedure or function and the value of the parameter can be overwritten by the procedure or function.

Example

Let's look at an example of how to create a procedure in Oracle.

The following is a simple example of a procedure:

```
CREATE OR REPLACE Procedure UpdateCourse
   ( name_in IN varchar2 )

IS
   cnumber number;
```

```
   cursor c1 is
   SELECT course_number
    FROM courses_tbl
    WHERE course_name = name_in;

BEGIN

   open c1;
   fetch c1 into cnumber;

   if c1%notfound then
      cnumber := 9999;
   end if;

   INSERT INTO student_courses
   ( course_name,
     course_number )
   VALUES
   ( name_in,
     cnumber );

   commit;

   close c1;

EXCEPTION
WHEN OTHERS THEN
   raise_application_error(-20001,'An error was encountered - '||SQLCODE||' -
ERROR- '||SQLERRM);
END;
```

This procedure is called *UpdateCourse*. It has one parameter called *name_in*. The procedure will lookup the course_number based on course name. If it does not find a match, it defaults the course number to 99999. It then inserts a new record into the student_courses table.

Drop Procedure

Once you have created your procedure in Oracle, you might find that you need to remove it from the database.

Syntax

The syntax to a drop a procedure in Oracle is:

```
DROP PROCEDURE procedure_name;
```

procedure_name

The name of the procedure that you wish to drop.

Example

Let's look at an example of how to drop a procedure in Oracle.

For example:

```
DROP PROCEDURE UpdateCourse;
```

Oracle / PLSQL: Functions

This Oracle tutorial explains how to **create and drop functions** in Oracle/PLSQL with syntax and examples.

Create Function

Just as you can in other languages, you can create your own functions in Oracle.

Syntax

The syntax to create a function in Oracle is:

```
CREATE [OR REPLACE] FUNCTION function_name
   [ (parameter [,parameter]) ]

   RETURN return_datatype

IS | AS

   [declaration_section]

BEGIN
   executable_section

[EXCEPTION
   exception_section]

END [function_name];
```

When you create a procedure or function, you may define parameters. There are three types of parameters that can be declared:

1. **IN** - The parameter can be referenced by the procedure or function. The value of the parameter can not be overwritten by the procedure or function.
2. **OUT** - The parameter can not be referenced by the procedure or function, but the value of the parameter can be overwritten by the procedure or function.
3. **IN OUT** - The parameter can be referenced by the procedure or function and the value of the parameter can be overwritten by the procedure or function.

Example

Let's look at an example of how to create a function in Oracle.

The following is a simple example of an Oracle function:

```
CREATE OR REPLACE Function FindCourse
   ( name_in IN varchar2 )
   RETURN number
IS
   cnumber number;

   cursor c1 is
   SELECT course_number
     FROM courses_tbl
     WHERE course_name = name_in;

BEGIN

   open c1;
   fetch c1 into cnumber;

   if c1%notfound then
      cnumber := 9999;
   end if;

   close c1;

RETURN cnumber;

EXCEPTION
WHEN OTHERS THEN
   raise_application_error(-20001,'An error was encountered - '||SQLCODE||' -
ERROR- '||SQLERRM);
END;
```

This function is called *FindCourse*. It has one parameter called *name_in* and it returns a number. The function will return the course number if it finds a match based on course name. Otherwise, it returns a 99999.

You could then reference your new function in a SQL statement as follows:

```
SELECT course_name, FindCourse(course_name) AS course_id
FROM courses
WHERE subject = 'Mathematics';
```

Drop Function

Once you have created your function in Oracle, you might find that you need to remove it from the database.

Syntax

The syntax to a drop a function in Oracle is:

```
DROP FUNCTION function_name;
```

function_name

The name of the function that you wish to drop.

For example:

```
DROP FUNCTION FindCourse;
```

This example would drop the function called *FindCourse*.