

# 1 Data Types

## Data Type

Data types enforce integrity constraints on columns in a database. There are many data types available, but we will focus on the most commonly used ones.

## 1.1 Number

### Number

Number is a generic data type that allows for numerical value : real and integer numbers and we can decide the floating point and size :

- Number : stores large values of integer and decimal numbers
- Number(p) : represent an integer number where p is max number of digits ,  $p \in [1,38]$
- Number(p,s) : p represent number of total digit  $p \in [1,38]$  , s: represent scale number of digits after decimal point ,  $s \in [0,p-1]$

it has some sub types like Integer which is  $\Leftrightarrow$  Number(38)

Examples :

Definition	Input	Stored As
NUMBER	124.56	12.56
	-99999	-99999
	44343	44343
NUMBER(5)	17.5	18
	123456	Error
	44300	44300
NUMBER(3)	99.3	99
	-677.9	678
	5432	Error
INTEGER	16.89	17
	-234532	-234532
	13.1	13
INTEGER(2)	234.9	Error
	10.4	10
	-20	-20
INTEGER(4)	1240	1240
	932.82	933
	-32330	Error
NUMBER(6,2)	34670.56	Error
	-9890.98	-9890.98
	23.232	23.23
NUMBER(5,3)	24.1562	24.156
	99	99.000
	343.77	Error

## Note

s can be > p , i just didn't want to include that case as it can be confusing and is rarely used

## 1.2 Date

### Date

Date is a data type that stores both the date and the time it accept a wide range of format and has many function

#### 1.2.1 Format

Format	Example
YYYY-MM-DD	2024-12-01
DD-MON-YYYY	30-NOV-2022
YYYY/MM/DD	2000/04/19
HH24:MI:SS	14:34:21
HH12:MI:SS AM/PM	07:45:15 AM
YYYY-MM-DD HH24:MI:SS	2021-01-30 22:50:10
YYYY-MM-DD HH12:MI:SS AM/PM	2014-03-19 1:21:45 PM

#### 1.2.2 Function

Fonction	Definition
SYSDATE	returns the current date and time of the machine running the oracle date base(server) in the format YYYY-MM-DD HH24:MI:SS
CURRENT_DATE	returns the current date and time of the user machine connecting to the oracle date base in the format YYYY-MM-DD HH24:MI:SS
TO_DATE(string , format)	converts a string into date in the given format
TO_CHAR(date , format)	converts a date into a formatted (given format) string
ADD_MONTHS(date , n)	returns a date which it adds/substracts n months to the given date
MONTHS_BETWEEN (date1 , date2)	returns an integer number that represents number of months between date1 and date2
NEXT_DAY(date , day_of_week)	returns date of the next given day string ('SUNDAY', 'MONDAY'...etc) starting to search from the given date
EXTRACT(field FROM date)	returns an integer number that represents the given field (MONTH , YEAR, DAY , HOUR , MINUTE , SECOND , WEEK ...etc) from given date

## Note

When inserting a date in a table using TO\_DATE it doesn't matter which format we use , we can use any format we want and the same thing is valid when needing to print a date usin TO\_CHAR , because oracle stores the data object not the format in insert

### 1.3 Char

#### Char

Char(len) stores string of len size , if the inputed string is smaller than the definition oracle will pad it with space char , len  $\in [1,2000]$

### 1.4 VARCHAR2

#### Varchar2

Varchar2(len) stores string of len size , if the inputed string is smaller than the definition oracle will store it without any padding , in older version len  $\in [1,4000]$  but in more recent version len  $\in [1,32767]$

#### 1.4.1 Function

Fonction	Definition
LENGTH(string)	returns intger : length of given string
TRIM(string)	returns string : removes all leading/trailing spaces
TRIM(char FROM string)	returns string : removes all char that are in the beginning or end of given string
UPPER(string)	returns string : convert all characters of the given string to upper case
LOWER(string)	returns string : convert all characters of the given string to lower case
CONCAT(string1,string2)	returns string : concat string1 with string2
SUBSTR(string,i,j)	returns string : extract substring from given string from index i to index j
REPLACE(string,sub_string,replace_string)	returns string : replace all occurences of sub_string in the given string with replace_string , not case sensitive
LPAD(string,nb,char)	returns string : pads the given string to the left Length(string)-nb times with given char
RPAD(string,nb,char)	returns string : pads the given string to the right Length(string)-nb times with given char
INSTR(string,sub_string)	returns integer : find the index of the first occurence of sub_string in the given string if sub string doesn't exist returns 0

Example :

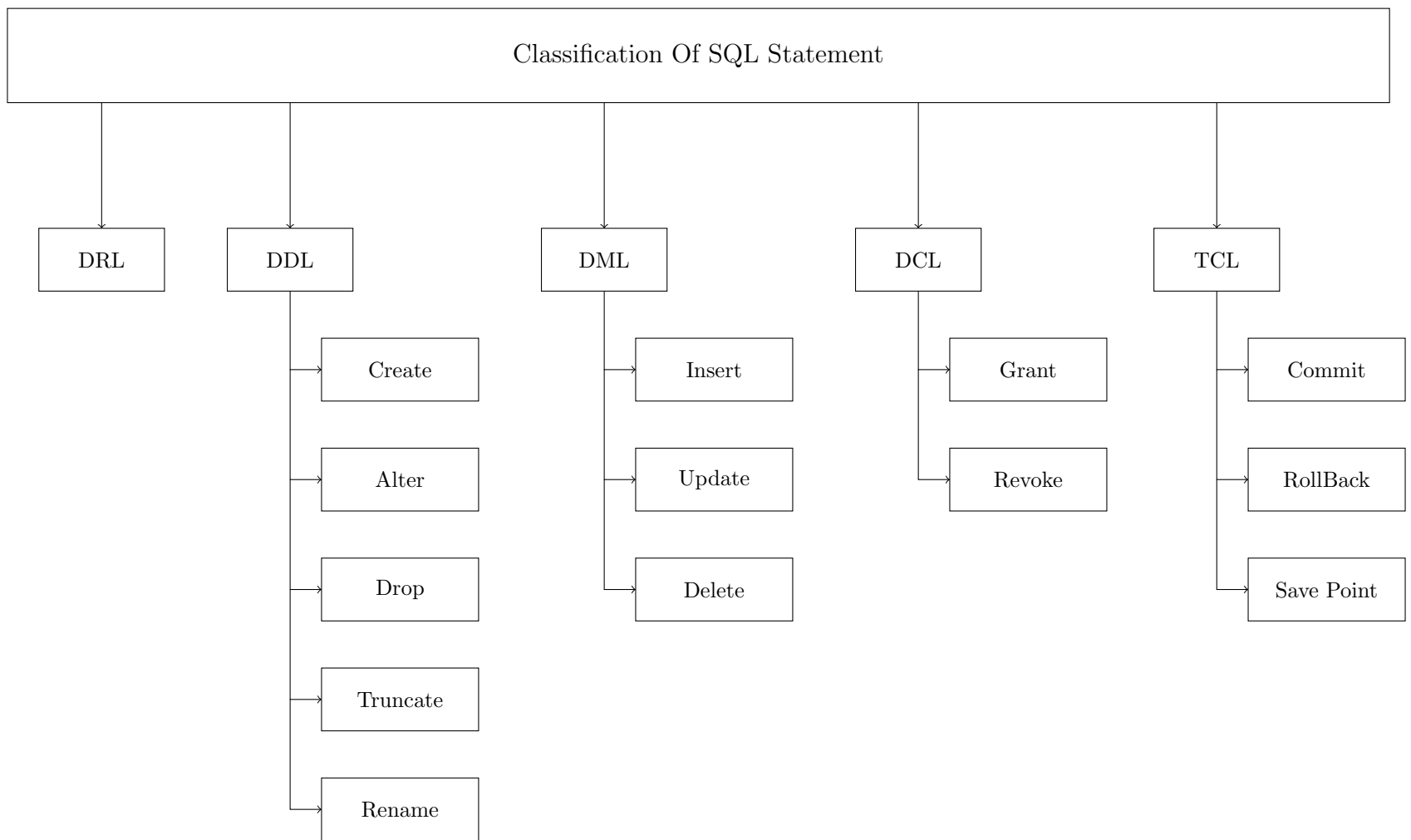
Fonction Call	Output
LENGTH('Hello World')	11
TRIM(' Hello World ')	'Hello World'
TRIM('!' FROM '!!!! Hello World !!!!!')	' Hello World '
UPPER('Hello World')	'HELLO WORLD'
LOWER('HeLlO WorLd')	'hello world'
CONCAT('hello ','world!')	'hello world!'
SUBSTR('I Love Java',8,11)	'Java'
REPLACE('Hello world , I missed you world','world','toto')	Hello toto , I missed you toto
LPAD('hello',10,'*')	'*****hello'
RPAD('world',11,'*')	'world*****'
LPAD('toto',4,'*')	'toto'
INSTR('I Hate Javascript','Java')	8
INSTR('I Hate Javascript','Python')	0

## Note

**Difference between CHAR and VARCHAR2:** CHAR will take up the full specified size, even if not all space is used, and will pad the value with spaces until it reaches the full size. In contrast, VARCHAR2 only stores the exact amount of space needed for the data, without padding with spaces.

Strings are 1-based(first index is 1)

## 2 Classification Of SQL Statement



## 3 DDL Commands

### 3.1 Create Table

#### Table Creation

To create a table in oracle sql we just have to give the table a name and define each column known as attribut by giving each of them a name , a dataType and an optional constraint that can be added in same line of attribut definition or on its own line , we will see contraint in details in the next section

#### Syntax :

```

1 CREATE TABLE tableName (
2     attribute_1 <Data Type> [Inline Constraint_1],
3     attribute_2 <Data Type> [Inline Constraint_2],
4     .....
5     attribute_n <Data Type> [Inline Constraint_n],
6     [Out-of-Line Constraints]
  
```

**Example :**

let's create student table

```
1 CREATE TABLE student (  
2     id number,  
3     firstName varchar2(50),  
4     lastName varchar2(50),  
5     grade number  
6 );
```

## 3.2 Table Constraints

### Constraints

Constraints are conditions set on the columns (attributes) of a table to ensure data integrity and consistency. Constraints can be defined:

- During table creation, either on the same line as the attribute definition(inline) or on a separate line(out-of-line)
- After table creation using the ALTER TABLE command

There are two types of constraints: static and dynamic.

### 3.2.1 Static Constraints

#### Static

Static constraints are fixed conditions that do not change based on data input.

- **Data Type** : Ensures Integrity of the column
- **NOT NULL**: Ensures that the attribute must have a value when inserting into the table.
- **UNIQUE**: Ensures that each value in the attribute is distinct. Unlike PRIMARY KEY, it allows null values.
- **PRIMARY KEY**: Combines UNIQUE and NOT NULL properties to ensure each value is unique and not null. Used to identify rows uniquely.
- **FOREIGN KEY**: References a primary key from another table to establish a relationship between tables.
- **DELETE ON CASCADE**: When deleting a row from the referenced (parent) table, all rows in the child table that contain the matching foreign key are also deleted.

### 3.2.2 Dynamic Constraints

## Dynamic

Dynamic constraints apply conditions that can change based on specified criteria.

- **CHECK:** Validates a specified condition before allowing data to be inserted or updated.
- **DEFAULT:** Sets a default value for the attribute if no value is provided during insertion.
- **TRIGGER:** Acts like a call back function , a block of code that gets executed automatically when defined event is triggered

### Syntax

```
1 attribute_i <DataType> not null
2 attribute_i <DataType> unique
3 attribute_i <DataType> primary key
4 attribute_i <DataType> references referenced_table(references_attribute)
5 attribute_i <DataType> default (value)
```

```
1 constraint constraint_name attribute_i not null
2 attribute_i not null
3
4 constraint constraint_name attribute_i unique
5 attribute_i unique
6
7 constraint constraint_name primary key (attribute_i,...,attribut_n)
8 primary key (attribute_1 ,..., attribute_n)
9
10 constraint constraint_name foreign key attribute_i references referenced_table(references_attribute)
11 foreign key attribute_i references referenced_table(references_attribute)
12
13 constraint constraint_name attribute_i default (value)
14 attribute_i default (value)
```

### Example :

let's create a new table section and recreate the student table with constraints

### Creating Section Table

#### Inline Method

```
1 create table section (
2     id_section number primary key,
3     name varchar2(5) not null check in ('A','B','C','D','1','2','3','4')
4 );
```

#### Outline Method

```
1 create table section (
2     id_section number,
3     name varchar2(5),
4     name not null,
5     constraint pk_sec primary key (id_section),
6     constraint chk_sec check ( name in ('A','B','C','D','1','2','3','4') )
```

```
7 );
```

## Create Student Table Inline Method

```
1 create table student (  
2     id number primary key,  
3     lastname varchar2(50) not null,  
4     firstname varchar2(50) not null,  
5     id_section number references section(id_section) on delete cascade,  
6     grade number(4,2) default 00.00 check (grade between 0 and 20),  
7     dob date not null check (dob<= add_months(sysdate,-18*12))  
8 );
```

## Outline Method

```
1 create table student (  
2     id number,  
3     constraint pk_student primary key(id),  
4     lastname varchar2(50),  
5     firstname varchar2(50),  
6     constraint nn_student_lastname lastname not null,  
7     constraint nn_student_firstname firstname not null,  
8     id_section number,  
9     constraint fr_student foreign key (id_section) references section(id_section) on delete cascade,  
10    grade number(4,2),  
11    grade default 00.00,  
12    constraint chk_student_grade check (grade between 0 and 20),  
13    dob date not null,  
14    constraint chk_student_dob check (dob<= add_months(sysdate,-18*12))  
15 );
```

## Naming Convention Of Constraints

- Primary Key : PK\_<tableName>
- Foreign Key : FK\_<tableName>
- Unique : UQ\_<tableName>\_<columnName>
- Check : CHK\_<tableName>\_<columnName>
- Default : DF\_<tableName>\_<columnName>
- Not Null : NN\_<tableName>\_<columnName>

## Note

### Constraint Name Must Be Unique

Tables inside the same PDB (pluggable data base) can't share the same constraints name

### Multiple Constraints

It is possible to define multiple constraints on a single attribute using the inline method. However, with the outline method, each constraint needs to be specified individually.



### 3.3 Drop Table

#### Drop

We can remove a table using the drop command

##### Syntax

```
1 drop table tableName;
```

##### Example

lets delete the section table we created

```
1 drop table section;
```

### 3.4 Rename Table

#### Rename

We can rename tables by using the rename command

##### Syntax

```
1 rename old_tableName to new_tableName;
```

##### Example

```
1 rename section to subsection;
```

## 3.5 Alter Table

### Alter

The ‘ALTER’ command is a versatile command that allows us to change various aspects of a table:

- Columns
  - **Renaming Column:** Rename the column.
  - **Modify Column:** Change the constraint and data type.
  - **Add Column:** Add a new column.
  - **Remove Column:** Remove a column.
- Constraints
  - **Add Constraint:** Add a new constraint.
  - **Remove Constraint:** Remove a constraint.
  - **Enable Constraint:** Enable an already existing constraint.
  - **Disable Constraint:** Disable an already existing constraint without deleting it.

#### Syntax

##### Columns Modification

```
1 alter tableName rename column old_colName to new_colName;  
2 alter tableName modify (colName [Constraints]);  
3 alter tableName add (colName [Constraints]);  
4 alter tableName drop column colName;
```

##### Constraints

```
1 alter table tableName rename constraint old_constraintName to new_constraintName;  
2 alter table tableName add constraint constraintName [Constraint];  
3 alter table tableName drop constraint constraintName;  
4 alter table tableName enable constraint constraintName;  
5 alter table tableName disable constraint constraintName;
```

#### Example

## 3.6 Truncate Table

### Truncate

To remove all rows from a table efficiently we use the truncate command

#### Syntax

```
1 truncate table tableName;
```

### Example

lets delete all records from student table

```
1 truncate table student;
```

## 4 DRL Commands

### 4.1 Select

#### Definition

To display the contents of one or more tables at once, we use the **SELECT** command. We can choose specific columns and tables to display, and if a table name is too long, we can assign a shorter alias to columns and tables name using the **AS** keyword. When selecting from multiple tables without a join condition, a Cartesian product occurs, meaning each row from one table is paired with each row from the other.

### Syntax

#### Table Selection

```
SELECT * FROM table1;  
SELECT column1 ,..., columnn FROM table1;  
SELECT t1.col1,...,t1.coln, t2.col1,...,t2.coln FROM table1 AS t1, table2 AS t2;
```

### 4.2 Where

#### Definition

The **WHERE** clause is used to filter rows in a table when displaying data with the **SELECT** command. Only rows that meet the specified condition(s) are shown in the result.

### Syntax

#### Where Clause

```
SELECT column1, column2, ... FROM table WHERE [Conditions];
```

### 4.3 Order By

#### Definition

We can sort the results of a query in either ascending or descending order using **ORDER BY**. This can be applied to one or multiple columns. The order of the columns specified is important; the database first sorts by the first column, and if there are rows with identical values in that column, it then sorts those rows by the next column, and so on. This allows for a prioritized sorting strategy.

### Syntax

#### Order By Clause

```
SELECT column1, column2, ... FROM table WHERE [Conditions] ORDER BY column1 DESC ,..., columnn ASC;
```

## 4.4 Group By

### Definition

To group rows that have the same value in a specified column, we use the **GROUP BY** command. We can group by multiple columns; the order is important because it will first group by the first column. If there are rows that have the same value in the first column but differ in the second column, those rows will appear in separate groups in the output. This allows us to apply aggregate functions to summarize data for each group.

### Syntax

#### Group By Clause

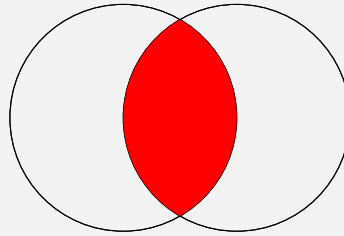
```
SELECT column1, column2, ... FROM table WHERE [Conditions] GROUP BY column1 , ... ,columnn  
ORDER BY column1 DESC ,..., columnn ASC;
```

## 4.5 Joins

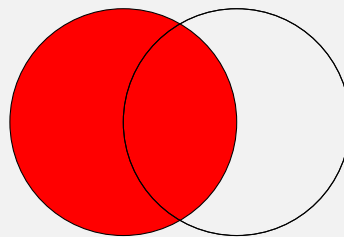
### Definition

joins allow you to combine rows from two or more tables based on related columns (referenced key)

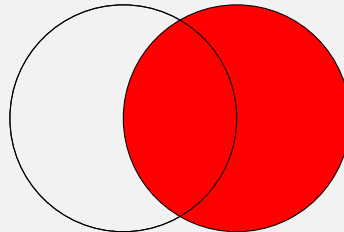
**Inner Join** An Inner Join returns only the common rows between tables



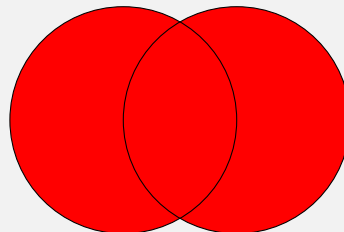
**Left Join** A Left Join returns all rows from the left table and the matched rows from the right table. If there's no match, NULL values are returned for columns from the right table.



**Right Join** A Right Join returns all rows from the right table and the matched rows from the left table. If there's no match, NULL values are returned for columns from the left table.



**Full Join** A Full Join returns all rows when there is a match in either left or right table. If there is no match, NULL values are returned for unmatched columns.



## Syntax

### Joins

#### Inner Join

```
SELECT column1,...,columnn FROM table1 INNER JOIN table2 ON table1.common_column = table2.common_column;
```

#### Left Join

```
SELECT column1,...,columnn FROM table1 LEFT JOIN table2 ON table1.common_column = table2.common_column;
```

#### Right Join

```
SELECT column1,...,columnn FROM table1 RIGHT JOIN table2 ON table1.common_column = table2.common_column;
```

#### Full Join

```
SELECT column1,...,columnn FROM table1 FULL OUTER JOIN table2 ON table1.common_column = table2.common_column;
```

## 4.6 Aggregation Functions

### Definition

Aggregation functions perform calculations on a set of values and return a single result. They are commonly used in conjunction with the **GROUP BY** clause to summarize data.

- **Avg()** : Calculates the average (mean) of numeric values in a specified column.
- **Min()** : Returns the smallest (minimum) value in a specified column.
- **Max()** : Returns the largest (maximum) value in a specified column.
- **Count()** : Counts the number of non-null entries in a specified column (or all rows if \* is used).
  - **count(\*)** : Counts All rows
  - **count(column<sub>i</sub>)** : counts number of rows where column<sub>i</sub> is not null
  - **count(distinct column<sub>i</sub>)** : counts number of rows where column<sub>i</sub> is not null without repetition
- **Sum()** : Adds up all values in a specified numeric column.

## 4.7 Operators

### Definition

Operators are symbols that specify operations to be performed on operands. They can be categorized as follows:

- **Logical Operators:** Used to combine conditions.
  - Logical And : AND
  - Logical Or : OR
  - Logical Not : NOT
- **Comparison Operators:** Used to compare values.
  - Equal : =
  - Not Equal : !=
  - Greater : >
  - Greater Or Equal : >=
  - Less : <
  - Less Or Equal : <=
  - Between : BETWEEN value<sub>1</sub> AND value<sub>2</sub>
  - In : IN (set of values)
- **Arithmetic Operators:** Used for mathematical calculations.
  - Multiplication : \*
  - Division : /
  - Sum : +
  - Subtraction : -

## 5 DML Commands

### 5.1 Insert

To insert rows into a table, we use the [INSERT](#) command. We can insert one row at a time or multiple rows at once from the same or different tables using the [ALL](#) keyword.

## Syntax

### Insert

#### **Insert One Row At A Time**

```
INSERT INTO tableName (column1,...,columnn) VALUES (value1,...,valuen);
```

#### **Insert Multiple Rows**

```
INSERT ALL
```

```
INTO tableName1 (column1,...,columnn) VALUES (value1,...,valuen);
```

```
INTO tableName2 (column1,...,columnn) VALUES (value1,...,valuen);
```

```
.....
```

```
INTO tableNamen (column1,...,columnn) VALUES (value1,...,valuen);
```

```
SELECT * FROM dual;
```

## 5.2 Update

### Definition

To change the values of some rows in a table, we use the **UPDATE** command, accompanied by the **WHERE** clause to update only specific rows.

## Syntax

### Update

```
UPDATE tableName SET column1 = value1,...,columnn = valuen  
WHERE [condition];
```

## 5.3 Delete

### Definition

To delete rows from a table, we use the **DELETE** command, accompanied by the **WHERE** clause to delete specific rows. Although it is possible to delete all rows using **DELETE**, it is better to use **TRUNCATE** for that purpose due to performance considerations.

## Syntax

### Delete

#### **DELETE SPECIFIC ROWS**

```
DELETE FROM tableName WHERE [condition];
```

#### **DELETE ALL ROWS**

```
DELETE FROM tableName;
```



## 6 PL/SQL

### 6.1 Introduction

#### Definition

PL/SQL, or Procedural Language/Structured Query Language, is an extension of SQL. While SQL (Structured Query Language) is primarily used for CRUD operations (querying, inserting, updating, and deleting data in relational databases), PL/SQL allows for full programmatic control with features such as control structures (loops and conditionals), variables, and error handling with exceptions. This enables the creation of scripts that can automate tasks with functions, procedures, and triggers, implement complex business logic, and manipulate data at a higher level than SQL alone.

#### Differences Between PL/SQL and SQL

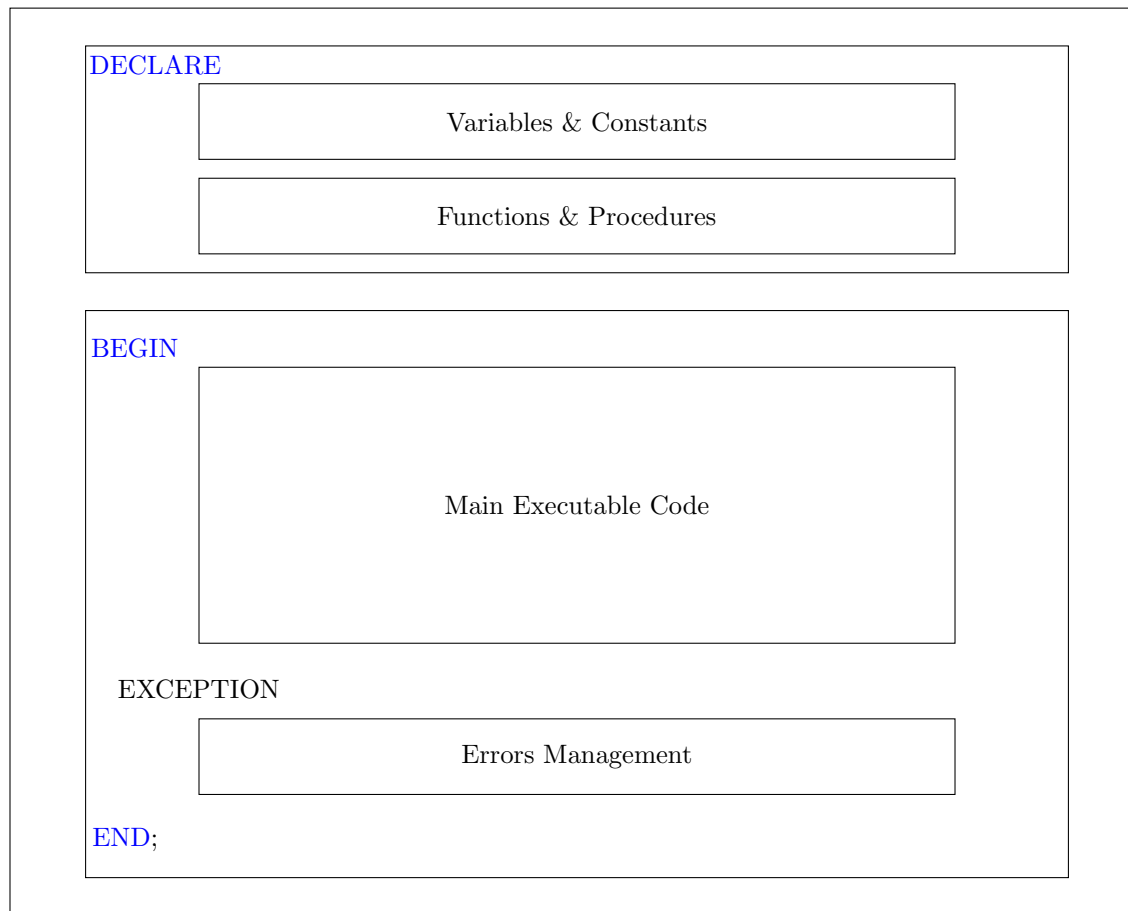
- SQL is limited to CRUD operations; PL/SQL adds procedural programming capabilities.
- PL/SQL provides advanced error handling through exceptions.
- PL/SQL supports modular programming with functions, procedures, and triggers.
- PL/SQL is specific to Oracle databases, whereas SQL is standardized across various databases.

### 6.2 Overview Of Plsql's Structure

#### Programme Structure

A PL/SQL has 3 blocks :

- DECLARE(Optional Block) : contains all the declared variables , constants & modules(functions,procedure)
- MAIN : contains the main executable code
- EXCEPTION(Optional Block) : handles erros with exceptions



### 6.3 Comments

#### Comments Syntax

##### One Line Comment:

-- Comment here

##### Multiple Lines Comment:

```
/*  
This is a Multiple  
Lines Comment  
*/
```

### 6.4 Printing

#### DBMS\_OUTPUT.PUT\_LINE

To print messages in the console, we use the `DBMS_OUTPUT.PUT_LINE` command. The message should be enclosed in single quotes `' '` and we use double pipes `||` to concatenate with variables:

```
DBMS_OUTPUT.PUT_LINE('Hello ' || name || '!' );
```

#### Note

To be able to see the printed messages in the console of SQL\*Plus, SQL Developer, etc., we need to activate the buffer responsible for printing the messages by using the command:

```
SET SERVEROUTPUT ON;
```

Note that this is only needed once , and it will remain active unless you explicitly turn it off.

## 6.5 Variables Declaration & Types

#### Variables & Constants

All variables and constants must be declared in the **DECLARE** scope the syntax is as follows

```
varName dataType := value           - - Variable Declaration
constName CONSTANT dataType := value - - Constant Declaration
```

#### Types

PL/SQL supports many standard data types, as seen previously. Here, we introduce two additional types:

- **Type:** Used to define a variable with the same data type as a column in a table:

```
varName tableName.columnName%TYPE;
```

- **RowType:** Used to define a variable as a record with the structure of a row in a table:

```
varName tableName%ROWTYPE;
```

#### Store Select Output In Variables

We can store the output of the **SELECT** command in variables using the **INTO** clause as follows:

```
SELECT col1, col2, ..., coln INTO var1, var2, ..., varn FROM tableName WHERE [conditions];
```

#### Note

##### **Order Of Variables Is Important**

The order of the variables in the **INTO** clause must match the order of the selected columns

##### **Select Should Output One Line Only**

When Storing the output of **SELECT** in variables , the output should be one line and not a table if not we will have to use cursor to navigate through the table we will cover that later on

## 6.6 Control Structures

#### Definition

In PL/SQL, control structures are constructs that help control the flow of execution in a block of code. They determine the order and conditions under which statements are executed and help make the code dynamic and responsive to varying conditions. The main types of control structures in PL/SQL are:

### 6.6.1 Conditional Control

#### If

```
IF condition1 THEN
- - statements to execute if condition1 is true
ELSIF condition2 THEN
- - statements to execute if condition2 is true
ELSE
- - statements to execute if none of the conditions are true
END IF;
```

#### Switch Case

```
CASE
WHEN condition1 THEN
- - statements to execute if condition1 is true
WHEN condition2 THEN
- - statements to execute if condition2 is true
ELSE
- - statements to execute if none of the conditions are true
END CASE;
```

### 6.6.2 Looping Control

#### Simple Loop

```
LOOP
- - statements to execute
EXIT WHEN condition; - - condition to exit the loop
END LOOP;
```

#### While Loop

```
WHILE condition LOOP
- - statements to execute while condition is true
END LOOP;
```

#### For Loop

##### Ascending

```
FOR counter IN start..end LOOP
- - statements to execute for each value of counter
END LOOP;
```

##### Descending

```
FOR counter IN REVERSE end..start LOOP
- - statements to execute for each value of counter
END LOOP;
```

## 6.7 Raise Application Error

### Raise Errors

RAISE\_APPLICATION\_ERROR is a procedure used to raise an error that halts code execution, with a custom error message. Each error\_code (between -20000 and -20999) is associated with an error message retrieved by SQLERRM, while SQLCODE captures the error code itself.

```
RAISE_APPLICATION_ERROR(error_code, error_message);
```

Though commonly used to handle user-defined exceptions, RAISE\_APPLICATION\_ERROR can also be used internally by the system for predefined exceptions, supporting error control in both system and custom PL/SQL operations.

## 6.8 Exceptions

### Definition

Exceptions help manage errors and improve readability compared to directly using RAISE\_APPLICATION\_ERROR. Under the hood, exceptions are built on RAISE\_APPLICATION\_ERROR. There are two main types of exceptions:

- **Predefined Exceptions:** These are system-defined exceptions, such as:
  - NO\_DATA\_FOUND: Raised when a [SELECT](#) statement returns no rows.
  - TOO\_MANY\_ROWS: Raised when a [SELECT](#) statement returns more than one row.
- **User-defined Exceptions:** Defined by the user using the [EXCEPTION](#) DataType.

### Syntax Exception

Exception management happens in the EXCEPTION block:

```
DECLARE
```

```
User_EXC EXCEPTION; - - Declare a custom exception
```

```
BEGIN
```

```
IF [Condition] THEN
```

```
RAISE User_EXC; - - Raise the custom exception
```

```
END IF;
```

```
- - Rest of code
```

```
EXCEPTION
```

```
WHEN NO\_DATA\_FOUND THEN
```

```
DBMS_OUTPUT.PUT_LINE('No data found.');
```

```
WHEN TOO\_MANY\_ROWS THEN
```

```
DBMS_OUTPUT.PUT_LINE('Too many rows returned.');
```

```
WHEN User\_EXC THEN
```

```
DBMS_OUTPUT.PUT_LINE('Custom error occurred.');
```

```
WHEN OTHERS THEN
```

```
DBMS_OUTPUT.PUT_LINE('An unexpected error occurred: ' || SQLERRM);
```

```
END;
```

## Exceptions with SQLCODE

```
DECLARE
sqlcode_1 := -20001;
sqlcode_2 := -20002;
BEGIN
-- Check for specific conditions and raise custom errors with codes
IF [Condition] THEN
RAISE_APPLICATION_ERROR(sqlcode_1, 'error message1');
[Condition] THEN
RAISE_APPLICATION_ERROR(sqlcode_2, 'error message2');
END IF;
EXCEPTION
WHEN OTHERS THEN
-- Use SQLCODE to check error codes directly
IF SQLCODE = sqlcode_1 THEN
DBMS_OUTPUT.PUT_LINE('Error: ' || SQLERRM);
ELSIF SQLCODE = sqlcode_2 THEN
DBMS_OUTPUT.PUT_LINE('Error: ' || SQLERRM);
ELSE
DBMS_OUTPUT.PUT_LINE('Unexpected error: ' || SQLERRM);
END IF;
END;
```

## Note

### What is OTHERS?

It's best practice to add OTHERS as the last exception handler, as it catches any exceptions not explicitly defined. This ensures any unexpected errors are managed gracefully.

### When to Use RAISE\_APPLICATION\_ERROR vs. Exceptions?

Although exceptions are built on RAISE\_APPLICATION\_ERROR, they offer better readability and manageability in complex code. Use exceptions for organized error handling, while RAISE\_APPLICATION\_ERROR provides a more direct and minimalistic approach.

## 6.9 Cursors

cursor

```
DECLARE
  -- Declare a cursor that selects data
  CURSOR cr IS
    select query

  -- Variables to hold fetched data
  var1 table.col1%TYPE;
  var2 table.col2%TYPE;
  .....
  varn table.coln%TYPE;
BEGIN
  -- Open the cursor
  OPEN emp_cursor;
  FETCH cr INTO var1,var2,...,varn
  -- Loop through the result set
  WHILE(cr%FOUND) LOOP
    --traitements
    FETCH cr INTO var1,var2,...,varn

  END LOOP;

  -- Close the cursor
  CLOSE emp_cursor;
END;
```

### 6.10 Triggers

### 6.11 Functions

### 6.12 Procedure