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 ∈[1,38]
- Number(p,s) : p represent number of total digit p ∈[1,38] , s: represent scale number of digits after decimal point , s ∈[0,p-1]

it has some sub types like Integer which is ⇔ Number(38)

Examples :

Definition	Input	Stored As
NUMBER	124.56 -99999 44343	124.56 -99999 44343
NUMBER(5)	17.5 123456 44300	18 Error 44300
NUMBER(3)	99.3 -677.9 5432	99 678 Error
INTEGER	16.89 -234532 13.1	17 -234532 13
INTEGER(2)	234.9 10.4 -20	Error 10 -20
INTEGER(4)	1240 932.82 -32330	1240 933 Error
NUMBER(6,2)	34670.56 -9890.98 23.232	Error -9890.98 23.23
NUMBER(5,3)	24.1562 99 343.77	24.156 99.000 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 ∈[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 ∈[1,4000] but in more recent version len ∈[1,32767]

1.4.1 Function

Fonction	Definition
LENGTH(string)	returns intger : length of given string
TRIM(string)	returns string : removes all leading/trailling 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 :

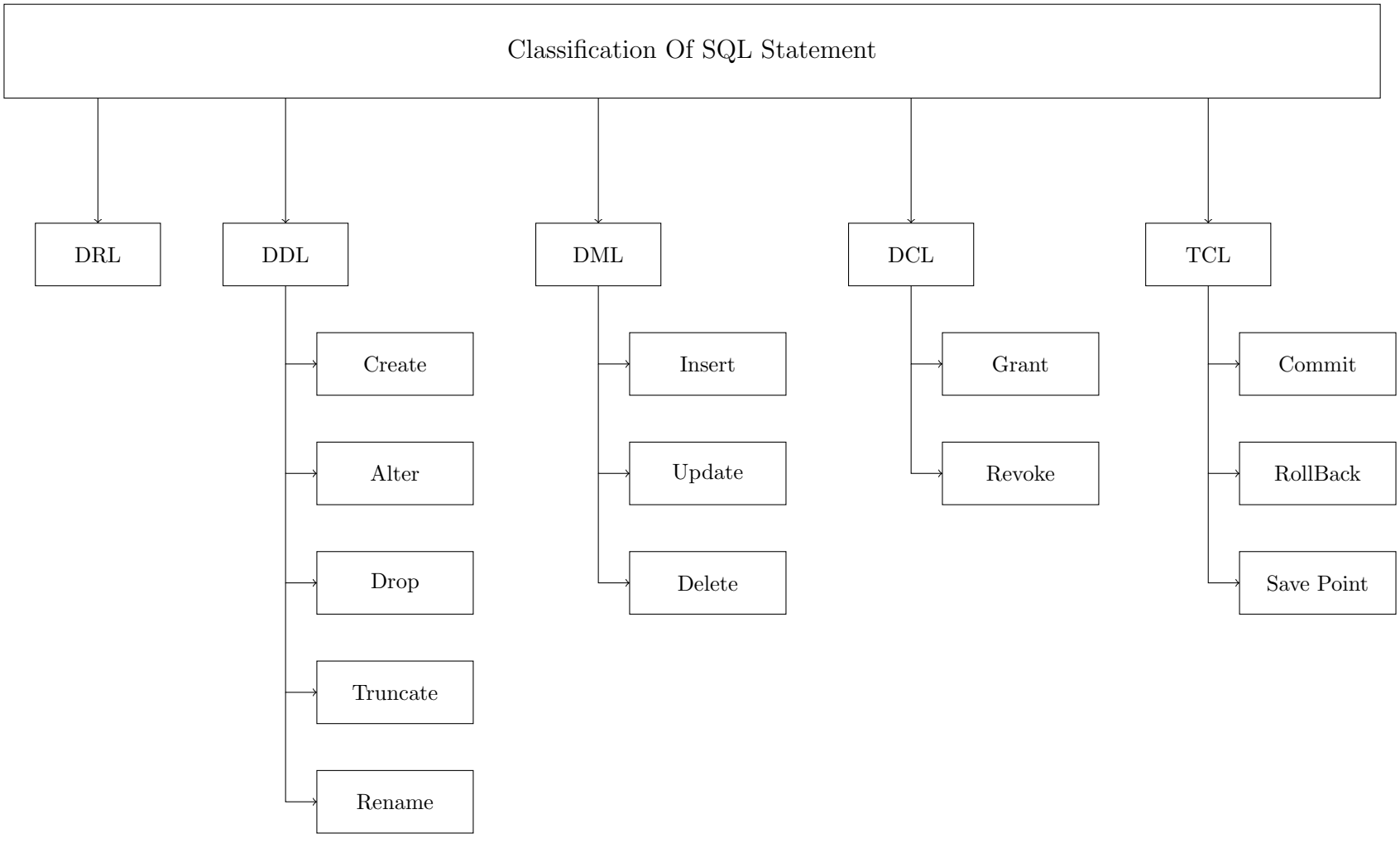
Fonction Call	Output
LENGTH('Hello World')	11
TRIM(' Hello World ')	'Hello World'
TRIM('!' FROM '!!!! Hello World !!!!!!')	' Hello World '
UPPER('Hello World')	'HELLO WORLD'
LOWER('HeLlO WorLd')	'hello world'
CONCAT('hello ','world!')	'hello world!'
SUBSTR('I Love Java',8,11)	'Java'
REPLACE('Hello world , 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 use the CREATE command , 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(in-line method) or on its own line(out-of-line method) , we will see contraint in details in the next section

Syntax :

```
CREATE TABLE tableName (
    attribute_1 <Data Type> [Inline Constraint_1],
    attribute_2 <Data Type> [Inline Constraint_2],
    ..................................
    attribute_n <Data Type> [Inline Constraint_n],
    [Out-of-Line Constraints]
);
```

Example :

let's create student table

```
CREATE TABLE student (
    id number,
    firstName varchar2(50),
    lastName varchar2(50),
    grade number
);
```

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

- **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 , can be null.
- **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.
- **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.

3.2.2 Dynamic Constraints

Dynamic

- **TRIGGER**: Acts like a call back function , a block of code that gets executed automatically when a defined event is triggered

Syntax

In-Line Method

```
attribute_i <DataType> not null

attribute_i <DataType> unique

attribute_i <DataType> primary key

attribute_i <DataType> references  referenced_table(references_attribute)

attribute_i <DataType> default (value)

attribute_i <DataType> check (Conditions)
```

Out-Of-Line Method

```
constraint  contraint_name  attribute_i not null

constraint contraint_name attribute_i unique

constraint contraint_name primary key (attribute_i,...,attribut_n)
primary key (attribute_1 ,..., attribute_n)

constraint contraint_name foreign key attribute_i references referenced_table(references_attribute)
foreign key (attribute_1 ,..., attribute_n) references referenced_table(attribute_1 ,..., attribute_n)

constraint contraint_name attribute_i default (value)

constraint constraint_name check (Conditions)
attribute_i check (Conditions)
```

Example :

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

Creating Section Table

In-Line Method

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

Out-Of-Line Method

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

Create Student Table

In-Line Method

```
create table student (
    id number primary key,
    lastname varchar2(50) not null,
    firstname varchar2(50) not null,
    id_section number references section(id_section) on delete cascade,
    grade number(4,2) default 00.00 check (grade between 0 and 20),
    dob date not null
);
```

Out-Of-Line Mehtod

```
create table student (
    id number,
    constraint pk_student primary key(id),
    lastname varchar2(50),
    firstname varchar2(50),
    constraint nn_lastname_student lastname not null,
    constraint nn_firstname_student firstname not null,
    id_section number,
    constraint fr_student foreign key (id_section) references section(id_section) on delete cascade,
    grade number(4,2),
    constraint df_grade_student grade default 00.00,
    check (grade between 0 and 20),
    dob date,
    constraint nn_dob_student dob not null
);
```

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

```
drop table tableName;
```

Example

lets delete the section table we created

```
drop table section;
```

3.4 Rename Table

Rename

We can rename tables by using the RENAME command

Syntax

```
rename old_tableName to new_tableName;
```

Example

```
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

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

Constraints

```
alter table tableName rename constraint old_constraintName to new_constraintName;
alter table tableName add constraint constraintName [Constraint];
alter table tableName drop constraint constraintName;
alter table tableName enable constraint constraintName;
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

```
truncate table tableName;
```

Example

lets delete all records from student table

```
truncate table student;
```

4 DRL Commands

4.1 Select

Select

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, we can give aliases to tables and columns . When selecting from multiple tables of different size , a Cartesian product occurs, meaning each row from one table is paired with each row from the other.

Syntax

```
select * from tablName;

select t.* from tableName t;

select col_1,...,col_n from tableName;

select t.col_1 alias_col_1,...,t.col_n alias_col_n from tableName t;

select t_1.col_1,...,t_1_col_n,...,t_n_col_1,...,t_n_col_n from tableName_1 t_1,...,tableName_n t_n;
```

4.2 Where

Where Clause

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

```
select col_1,col_2,...,col_n from tableName where [Conditions];
```

4.3 Aggregation Functions

Aggregation Functions

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(column$_i$)** : Calculates the average (mean) of numeric values in a specified column.
- **Min(column$_i$)** : Returns the smallest (minimum) value in a specified column.
- **Max(column$_i$)** : 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$_i$)** : counts number of rows where column$_i$ is not null
  - **count(distinct column$_i$)** : counts number of rows where column$_i$ is not null without repetition
- **Sum(column$_i$)** : Adds up all values in a specified numeric column.

Note

- We can do arithmethical operations inside parameters of some aggregation functions like avg,max,min,sum
- We must use GROUP BY when using aggregation functions if not we will have an error

4.4 Group By

Group By

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

```
select col_1, col_2,...,col_n from tableName where [Conditions]
group by col_1 , col_2 ,... ,column_n;
```

4.5 Having

Having Clause

Similar to WHERE, which filters rows based on conditions, HAVING is used to filter groups of data rather than individual rows. Unlike WHERE, which applies conditions before grouping, HAVING is applied after the GROUP BY clause. This allows you to filter aggregated results.

Syntax

```
select col_1, col_2,...,col_n from tableName where [Conditions]
group by col_1 , col_2 ,... ,column_n
having [Conditions];
```

4.6 Order By

Order By

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

```
select col_1, col_2,...,col_n from tableName where [Conditions]
group by col_1 , col_2 ,... ,column_n
having [Conditions]
order by col_1 desc , col_2 asc ,..., col_n asc;
```

4.7 Joins

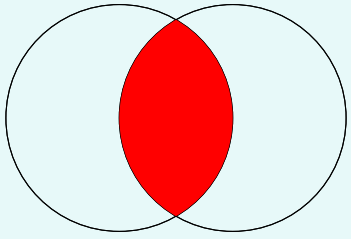
Joins

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

4.7.1 Inner Joins

Inner Joins

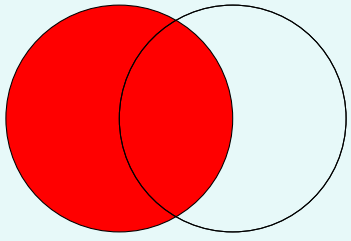
An Inner Join returns only the common rows between tables



4.7.2 Left Join

Left Join

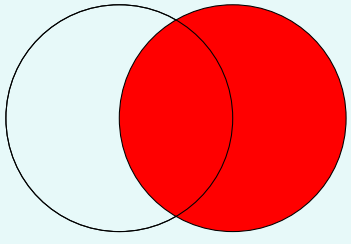
A Left Join returns all rows from the left table and the matched rows from the right table. If there's no match, NULL values are returned for columns from the right table.



4.7.3 Right Join

Right Join

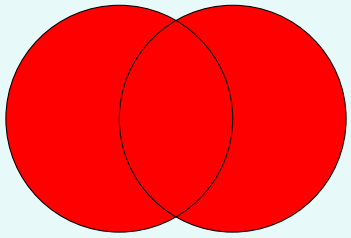
A Right Join returns all rows from the right table and the matched rows from the left table. If there's no match, NULL values are returned for columns from the left table.



4.7.4 Full Join

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

```
select t1.col_1 ,..., t1.col_n , t2.col_1 ,..., t2.col_n , tn.col_1 , ... , tn.col_n from
table1 t1 inner join table2 t2 on t1.communCol = t2.communCol inner join ..... inner join
tablen tn on tn-1.communCol = tn.communCol;

select t1.col_1 ,..., t1.col_n , t2.col_1 ,..., t2.col_n , tn.col_1 , ... , tn.col_n from
table1 t1 left join table2 t2 on t1.communCol = t2.communCol left join ..... left join
tablen tn on tn-1.communCol = tn.communCol;

select t1.col_1 ,..., t1.col_n , t2.col_1 ,..., t2.col_n , tn.col_1 , ... , tn.col_n from
table1 t1 right join table2 t2 on t1.communCol = t2.communCol right join ..... right join
tablen tn on tn-1.communCol = tn.communCol;

select t1.col_1 ,..., t1.col_n , t2.col_1 ,..., t2.col_n , tn.col_1 , ... , tn.col_n from
table1 t1 full outer join table2 t2 on t1.communCol = t2.communCol full outer join ..... full outer join
tablen tn on tn-1.communCol = tn.communCol;
```

Note

We can have different types of joins in one select query

4.8 Operators

Operators

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

4.8.1 Logical Operators

Logical

Used to combine conditions.

- Logical And : AND
- Logical Or : OR
- Logical Not : NOT

4.8.2 Comparison Operators

Comparison

Used to compare values.

- Equal : =
- Not Equal : !=
- Greater : >
- Greater Or Equal : >=
- Less : <
- Less Or Equal : <=
- Between : BETWEEN value_1 AND value_2
- In : IN (set of values)

4.8.3 Arithmetic Operators

Arithmetic

Used for mathematical calculations.

- Multiplication : *
- Division : /
- Sum : +
- Subtraction : -

5 DML Commands

5.1 Insert

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 Once

```
Insert into tableName (col_1,...,col_n) VALUES (value_1,...,value_n);
```

Insert In Multiple Tables

```
insert all
into tableName_1 (col_1,...,col_n) VALUES (value_1,...,value_n)
into tableName_2 (col_1,...,col_n) VALUES (value_1,...,value_n)
.....................................
into tableName_n (col_1,...,col_n) VALUES (value_1,...,value_n)
select * from dual;
```

Example

Tables Definition

Student Table

Column Name	Data Type	Constraints
id	number	primary key
lastname	varchar2(50)	not null
firstname	varchar2(50)	not null
id_section	number	foreign key section(id_section) delete on cascade
grade	number(4,2)	default 0 check between 0 and 20
dob	date	not null

Section Table

Column Name	Data Type	Constraints
id_section	number	primary key
name	varchar2(5)	not null check in ('A','B','C','D',1,2,3,4)

Insert Once

```
insert into section (id_section,name)
 values (1,'A');

insert into section (id_section,name)
 values (2,'B');

insert into student (id,lastname,firstname,id_section,grade,dob)
 values (1,chabane,rabah,2,11.80,to_date('2002-03-19','YYYY-MM-DD'));

insert into student (id,lastname,firstname,id_section,grade,dob)
 values (2,adem,lyna,1,13.451,to_date('2004-07-19','YYYY-MM-DD'));

insert into student (id,lastname,firstname,id_section,grade,dob)
 values (3,chaouche,mohamed,null,12.125,to_date('2004-02-20','YYYY-MM-DD'));
```

Tables After Insert

Section Table

id_section	name
1	'A'
2	'B'

Student Table

id	lastname	firstname	id_section	grade	dob
1	'chabane'	'rabah'	2	11.80	2002-03-19
2	adem	lyna	1	13.24	2004-07-19
3	chaouche	mohamed	null	12.13	2004-02-20

Insert In Multiple Tables

```
insert all

into section (id_section,name) VALUES (3,'C')

into section (id_section,name) VALUES (4,'D')

into student (id,lastname,firstname,id_section,grade,dob)
 values (4,bakhti,sohaib,3,10.51,to_date('2000-10-01','YYYY-MM-DD'))

into student (id,lastname,firstname,id_section,grade,dob)
 values (5,ibtissame,ahlem,4,14.834,to_date('2001-08-21','YYYY-MM-DD'))

into student (id,lastname,firstname,id_section,grade,dob)
 values (6,yacine,salem,null,9.801,to_date('2000-11-06','YYYY-MM-DD'))

select * from dual;
```

Tables After Insert

Section Table

id_section	name
1	'A'
2	'B'
3	'C'
4	'D'

Student Table

id	lastname	firstname	id_section	grade	dob
1	'chabane'	'rabah'	2	11.80	2002-03-19
2	adem	lyna	1	13.24	2004-07-19
3	chaouche	mohamed	null	12.13	2004-02-20
4	bakhti	sohaib	4	10.51	2000-10-01
5	ibtissame	ahlem	3	14.83	2001-08-21
6	yacine	salem	null	9.80	2000-11-06

Note

We don't have to precise columns names when inserting , it's optional it just makes the code more readable

5.2 Update

Update

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 tableName set col_1 = value_1 , col_2 = value_2,..., col_n = value_n
where [conditions];
```

5.3 Delete

Delete

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 from tableName where [Conditions];
```

6 PL/SQL

6.1 Introduction

What's PL/SQL ?

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

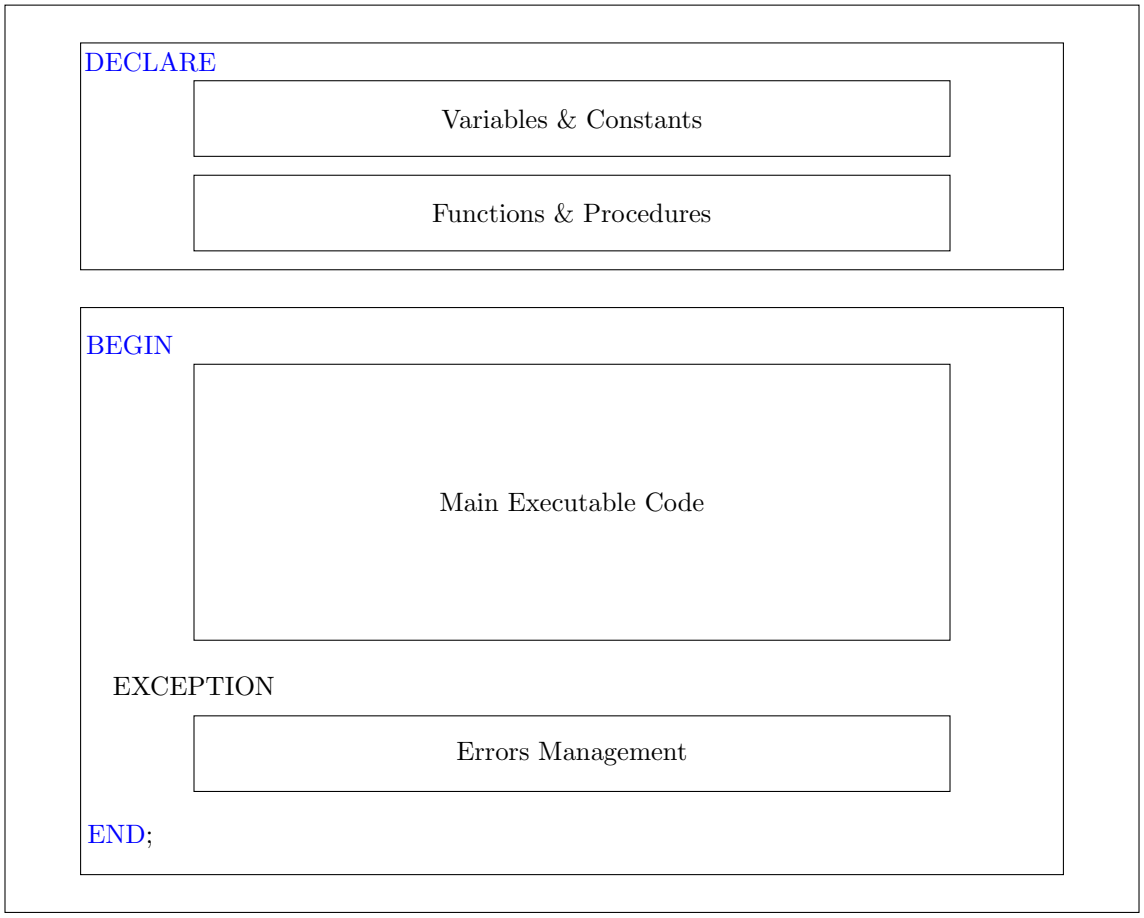
- SQL is limited to CRUD operations; PL/SQL adds procedural programming capabilities.
- PL/SQL provides advanced error handling through exceptions.
- PL/SQL supports modular programming with functions, procedures, and triggers.
- PL/SQL is specific to Oracle databases, whereas SQL is standardized across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



6.3 Comments

Syntax

Single Comment

```
-- this is a single comment
```

Multi-Line Comment

```
/*

this is a

multi-line

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.

Syntax

```
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 , we use := to affect values to variables

Syntax

```
varName <DataType> := value;              -- variable declaration

constName constant <DataType> := value;   -- constant declaration
```

Types

PL/SQL supports all data types normal sql has like those we've 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:

Syntax

```
select col_1 , col_2 , ..., col_n
INTO var_1 , var_2, ..., var_n from tableName  -- var_i is tableName.col_i%type

select * INTO var from tableName               -- var is tableName%rowtype
```

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 Ouput One Line Only

When Storing the output of SELECT in variables , the ouput 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

We Must Use Store Select Output

In PL/SQL we have to always store the output of a select if not we will have a complilation error

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

Syntax

```
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

Syntax

```
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

Syntax

```
loop

    -- statements to execute

    exit when condition; -- condition to exit the loop

end loop;
```

While Loop

Syntax

```
while condition loop

 -- statements to execute while condition is true

end loop
```

For Loop

Syntax

```
--Ascending For Loop

for counter in start..end loop

 -- statements to execute for each value of counter

end loop;


--Descending For Loop

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.

Syntax

```
declare

sqlCode_1 number := -20004; -- value must be between -20000 and 20999

begin

--code

if Condition then

raise_application_error(sqlCode_1,'custom error message') User_EXC;    -- Raise Application Error

end if;

-- Rest of code

end;

/
```

6.8 Exceptions

Exception

Exceptions help manage errors . Under the hood, exceptions are built on RAISE_APPLICATION_ERROR , only difference is that it's more readable. 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

```
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;

/
```

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.

Cursor

Cursors are used when a SELECT query returns a table (more than one row). To use a cursor, we first declare a variable of the CURSOR data type and associate it with a SELECT query.

Then, inside the BEGIN-END block, we perform the following steps:

- Open the cursor using the OPEN keyword.
- Load the first row using the FETCH keyword and store the output in variables.
- Loop through the table using the FOUND function, which is a boolean function that returns **true** if a row is successfully loaded.

Inside the WHILE loop:

- Process the current row.
- Load the next row using FETCH.

When FOUND returns false, indicating no more rows, the loop exits. Finally, close the cursor using the CLOSE keyword to free up memory.

Syntax

```
declare
    -- Declare a cursor that selects data
    cursor cr is
      [select query]

    -- Variables to hold fetched data
    var_1 table.col1%TYPE;
    var_2 table.col2%TYPE;
    .....
    var_n table.coln%TYPE;

begin
    -- Open the cursor
    open cr;

    -- load first line
    fetch cr INTO var_1,var_2,...,var_n

    -- Loop through the result set
    while(cr%FOUND) loop

        --traitements

        --loads next line
        fetch cr into var1,var2,...,varn

    end loop;

    -- Close the cursor
    close cr;
end;
/
```

6.9 Trigger

Trigger

Triggers are standalone PL/SQL code blocks that execute automatically in response to a specified event.

There are two types of triggers:

- Row-level triggers
- Table-level triggers

Triggers can be associated with various events, such as INSERT, UPDATE, or DELETE.

They are useful for automating tasks, enforcing rules, or logging changes.

Syntax

```
create or replace trigger triggerName

event  -- when trigger execute

level  -- row or table level

declare

-- variables

begin

-- traitements

end;
/
```

6.9.1 Row Level Trigger

Row Level

We use the FOR EACH ROW keyword, which instructs the trigger to execute for every row that is deleted or updated.
This also provides us with access to:

- :NEW.colName – This gives the value of a column for the newly inserted or updated row.
- :OLD.colName – This gives the value of a column for a deleted row or the value before an update.

We use row-level triggers when we need to access :NEW and :OLD, or when the event is row-specific, such as DELETE or UPDATE.

Syntax

```
create or replace trigger triggerName

event  -- when trigger execute

for each row  -- row level

declare

-- variables

begin

-- traitements

end;
/
```

6.9.2 Table Level Trigger

Table Level

A trigger is considered table-level if we omit the FOR EACH ROW keyword. Unlike row-level triggers, table-level triggers do not have access to :NEW and :OLD.

We use table-level triggers when:

- There is no need to access :NEW or :OLD values.
- The event is not row-specific, such as when performing actions like DROP, ALTER, or other table-wide operations.
- We want to override a command using INSTEAD OF.

Syntax

```
create or replace trigger triggerName

event  -- when trigger execute

-- table level

declare

-- variables

begin

-- traitements

end;
/
```

6.9.3 Trigger Events

Events

Triggers are defined for different types of events, which can be categorized as follows:

Trigger Types:

- BEFORE : The trigger executes before an event.
- AFTER : The trigger executes after an event.
- INSTEAD OF : The trigger overrides the event entirely (typically used with views).

Operations:

- INSERT on tableName : Trigger fires when a row is inserted into the table.
- DELETE on tableName : Trigger fires when a row is deleted from the table.
- UPDATE on tableName : Trigger fires when a row is updated in the table.
- UPDATE on tableName.columnName : Trigger fires when a specific column in the table is updated.
- DROP on tableName : Trigger fires when a table is dropped .

You can use the OR keyword in a trigger definition to specify multiple events for the same table. This allows the trigger to fire on different types of events.

Syntax

```
create or replace trigger triggername
    { after | before | instead of }
    { insert | delete | update | drop }
    [ or { insert | delete | update | drop } ]  -- optional: multiple events
    on tablename

[ for each row ]  -- optional: row-level or table-level

declare

    -- variables

begin

    -- logic

end;
/
```

Note

A Trigger Can Be Linked to Only One Table

A trigger can only be linked to a single table. Therefore, attempting something like BEFORE UPDATE ON table1 OR AFTER INSERT ON table2 is incorrect.

Avoid Using Multiple Events in a Single Trigger

It is generally not considered a best practice to combine multiple events with the OR keyword in a single trigger. This can make the trigger logic harder to maintain and understand.

6.10 Function

Syntax

Function Definition

Local Function

```
declare

-- local function
create or replace function functionName (par_1 in dataType_1 ,..., par_n in dataType_n)
return return_DataType

is

-- variables

begin

-- function body

 return result;

end functionName;


-- variables


begin

-- main pl/sql executable block

end;
/
```

Stand Alone Function

```
-- standalone function
create or replace function functionName (par_1 in dataType_1 ,..., par_n in dataType_n)
return return_DataType

is

-- variables

begin

-- function body

 return result;

end functionName;
/
```

Function Call

Inside Begin End Block

```
begin

   functionName(par_1,...,par_n);

end;
/
```

Call Directly With Exec

```
exec functionName(par_1,...,par_n);
```

Note

Standalone vs. Local Functions and Procedures

- **Standalone:** These are stored in the DBMS schema and can be called from anywhere.
- **Local:** These are defined inside the ‘DECLARE‘ block and can only be accessed or called within the same PL/SQL code.

Common Errors in Parameters

- You cannot use ‘TYPE‘ or ‘ROWTYPE‘ in parameter definitions.
- You cannot specify a size for ‘CHAR‘ or ‘VARCHAR2‘ in parameters.

6.11 Procedure

Syntax

Procedure Definition

Local Procedure

```
declare

-- local procedure
create or replace procedure procedureName (par_1 in dataType_1 ,..., par_n in dataType_n)

is

-- variables

begin

-- procedure body


end procedureName;


-- variables


begin

-- main pl/sql executable block

end;
/
```

Stand Alone Procedure

```
-- standalone procedure
create or replace procedure procedureName (par_1 in dataType_1 ,..., par_n in dataType_n)

is

-- variables

begin

-- procedure body


end procedureName;
/
```

Procedure Call

Inside Begin End Block

```
begin

   procedureName(par_1,...,par_n);

end;
/
```

Call Directly With Exec

```
exec procedureName(par_1,...,par_n);
```

7 View

7.1 What Are Views ?

View

Views are virtual tables that help simplify SELECT queries, allowing us to avoid retyping the same query repeatedly. They can also improve performance, especially when dealing with joins .There are two types of views:

- **External Views**
- **Materialized Views**

7.2 External

External View

External views have the same performance as normal tables. They simplify SELECT queries, eliminating the need to retype them each time we want to retrieve information from a table.

External views support data insertion. For a view to be considered external, it must meet all the following criteria:

- Include the primary key of the table.
- Must not contain any joins.
- Must not use aggregation functions.
- Must Contain All 'NOT NULL' attributes.

Syntax

```
create view viewName as [select query]
```

7.3 Materialized

Materialized View

Materialized views offer better performance and serve as read-only tables. Unlike external views, we cannot insert data into them , and they don't refresh automatically like external view

A view is considered materialized if any of the following conditions are met:

- It does not include the primary key.
- It does not include all 'NOT NULL' attributes.
- It contains joins.
- It uses aggregation functions.

Syntax

On Demand

```
1
2  -- create materialzed view
3
4  create materialized view viewName
5  refresh on demand
6  as
7  [select query]
8
9
10 -- to refresh it
11
12 exec dbms_mview.refresh('viewName');
```

On Commit

```
1
2  -- create materialzed view
3
4  create materialized view viewName
5  refresh on commit
6  as
7  [select query]
8
9
10 -- to refresh it
11
12 -- insert in table
13
14 insert into tableName values(val_1,...,val_2);
15
16 -- commit
17
18 commit;
```

Note

On Commit vs On Demand

- **On Demand** : default behaviour of materialized view , we have to refresh the view manually using the DBMS_MVIEW.REFRESH procedure
- **On Commit** : after each insert commit view automatically get refreshed

Insert On External View

Any insert on the external view will reflect on the source table