CREATE TABLE:

create table table_name(columnnName datatype constraints,columnname datatype references table2(columnname));

ADD COLUMN:

alter table tablename add column columnname datatype;

DROP COLUMN:

Alter table tablename drop columnname;

MODIFY DATATYPE:

Alter table tablename modify columnname datatype;

RENAME COLUMNNAME:

Alter table tablename rename column oldcolumnname to newcolumname;

TRUNCATE TABLE:

Truncate table table_name;

INSERT INTO TABLES:

Insert into table_name (columnname1, columnname2) values (column1value, colum2value);

Update DATA in table using condition:

Update tablename set columname =value where condition;

DELETE DATA in table using condition:

Delete from tablename where condition;

Order data in particular order:

Select * from tablename order by columnname desc;

-SELECT * FROM tablename WHERE LOWER(columnname) NOT LIKE '%@yahoo.com';

Here, **LOWER(colulmname)**: This function converts all characters in the columnname to lowercase making it easier to compare.
NOT LIKE '%@yahoo.com': The LIKE operator is used to search for a specified pattern in a column. The % symbol is a wildcard that matches zero or more characters. NOT LIKE means we are looking for rows where the email does **not** match the pattern '%@yahoo.com'. Essentially, it filters out all emails ending with '@yahoo.com'.

SELECTING DATA IN A RANGE:

Select * from tablename where columname **between** startingrange **and** ending range;

Checking for multiple queries all at once:

Select * from tablename where columnname = 'value1' or columnname = 'value2';

Or

Select * from tablename where columnname in ('value1' ,'value2');

comparing multiple numeric(int) column values all at once:

select * from tablename where column1 > column2;

getting a substring from one or more column and merging them:

select columnname, concat(substr(columnname,1,4), substr(columnname,1,5)) as something from tablename;

- **concat(...)**: This function combines (concatenates) two or more strings into one.
- **substr(username,1,4)**: This function extracts a substring from the username column, starting at the first character and taking the next four characters.
- **substr(phoneno,1,5)**: This function extracts a substring from the phoneno column, starting at the first character and taking the next five characters.

select user*id, username, case when phoneno is not null then phoneno when email email is not null then email else 'contact not provided' end as contact from users where to*char(dob,'mm')='02' order by username desc;
CASE Statement:

- The CASE statement checks the following conditions:
    - If phoneno is not NULL, it selects phoneno as contact.
    - If email is not NULL, it selects email as contact.
    - If neither phoneno nor email is provided, it sets contact to 'contact not provided'.

Filtering Rows:

- `where to_char(dob,'mm')='02'`: This filters the rows where the month of the date of birth (`dob`) is February.

select isbn, title, author, case when price < 300 then 'affordable price' when price >= 300 then 'moderate price' else 'very expensive' end as price_categorisation from book ;

Categorize prices under the heading PRICE_CATEGORIESATION: It also categorizes the price of each book into three categories:

- If the price is less than 300, it labels the book as having an "affordable price".
- If the price is 300 or more, it labels the book as having a "moderate price".
- If the price doesn't fit into the above categories (which is unlikely here), it labels the book as "very expensive".

select username
case
when address ='new york' then substr(email,1,instr(email,'@')-1) || '@Yahoo.com'
else email
end as email_details
from users;

Modify email addresses: It checks the address of each user:

- If the address is 'New York', it changes the user's email domain to '@Yahoo.com'. It does this by:
    - Extracting the part of the email before the '@' symbol using substr(email, 1, instr(email, '@') - 1).

    - Concatenating this part with '@Yahoo.com'.
- If the address is not 'New York', it keeps the email as it is.

From the users table: All this information is taken from the "users" table in the database.

So, the query helps you get a list of usernames with their email addresses, modifying the email domain to '@Yahoo.com' for users located in New York.

select username, round((sysdate-dob)/365) as age from users where round((sysdate-dob)/365)>35;

Calculate age: It calculates the age of each user by:

- Subtracting the date of birth (dob) from the current date (sysdate).
- Dividing the result by 365 to convert the difference from days to years.
- Rounding the result to the nearest whole number using the round function.

Filter users: It only includes users whose calculated age is greater than 35.

select user_id, count(borrowing_id) as book_count from borrowing where extract(year from borrow_date)=2024
group by user_id;

The COUNT function is used to count the number of rows that match a specified condition.

In the query, COUNT is used to count the number of times each user borrowed a book. The result is labeled as book_count.

The EXTRACT function is used to retrieve a specific part of a date, such as the year, month, or day. EXTRACT is used to get the year from the borrow_date. This helps in filtering the borrowings that happened in the year 2024.

The GROUP BY clause is used to group rows that have the same values in specified columns into summary rows.

In the query, GROUP BY is used to group the results by user_id. This means that the count of borrowed books is calculated for each user separately.

select u.user_id, u.username, nvl(min(b.fine),0) as min_fine
from users u
left join borrowing b on u.user_id=b.user_id
group by u.user_id, u.username;

nvl(min(b.fine),0) as min_fine: This selects the minimum fine from the borrowing table for each user. If there are no fines, it returns 0.
left join borrowing b on u.user_id=b.user_id: This joins the users table with the borrowing table on the user_id column. A left join means all users will be included, even if they have no corresponding entries in the borrowing table.

group by u.user_id, u.username: This groups the results by user_id and username, ensuring that each user appears only once in the result set.

select distinct b1.title as title1,
b2.title as title2,
b1.author as author,
b1.publisher
from book b1
join book b2 on b1.author=b2.author
and b1.publisher=b2.publisher
and b1.title <> b2.title
order by b1.title asc;

Join Operation:

- join book b2 on b1.author=b2.author and b1.publisher=b2.publisher and b1.title <> b2.title: This joins the book table with itself. The join condition ensures that:
    - The authors are the same (b1.author = b2.author).
    - The publishers are the same (b1.publisher = b2.publisher).
    - The titles are different (b1.title <> b2.title).

Distinct:

- select distinct: This ensures that each combination of title1, title2, author, and publisher appears only once in the result set.

Ordering:

- order by b1.title asc: This sorts the results by the title of the first book (title1) in ascending order.

Limit and offset question 4

SQLBolt - Learn SQL - SQL Lesson 4: Filtering and sorting Query results

PL/SQL

datatypes in pl/sql
-scalar(Boolean, number, date)
-Large Object or LOB (video graphics, graohic image, sound waveforms)
-composite (collection or records like arrays individually access each item data in collection)
-reference (pointer to other data items)

basic pl/sql syntax

```
declare

begin

exception
end
/
```

if you declare something inside the declare statement it will be treated as a global variable whereas if you declare it inside a begin statement it will be an local variable meaning it can only be used inside that begin statement

declaring variable -
variablename datatype ;
variablename tablename.columnname%type;

declaring constants -
variablename constant datatype;

conditional statements

1. If

```
IF condition THEN
  //Statements;
END IF;
```

2. If else

```
IF condition THEN
  S1;
```

```
ELSE
  S2;
END IF;
```

3. If elsif

```
IF condition1 THEN
  S1;
ELSif condition2 then
        S2;
ELSif condition2 then
        S3;
else
        s4;
END IF;
```

4. case statements

Normal case syntax

```
DECLARE grade char(1) := 'A';
BEGIN CASE grade
when 'A' then dbms_output.put_line('Excellent');
when 'B' then dbms_output.put_line('Very good');
else dbms_output.put_line('No such grade');
 END CASE;
 END;
/
```

other way(searched case)syntax:

```
DECLARE grade char(1) := 'A';
BEGIN CASE
when grade= 'A' then dbms_output.put_line('Excellent');
when marks > 90 then dbms_output.put_line('Very good');
```

```
else dbms_output.put_line('No such grade');
 END CASE;
 END;
/
```

loops in pl/sql:
3 types of loops are there in pl/sql

1. Basic loop

```
LOOP
  Sequence of statements;
condition to exit;
END LOOP;
```

example

```
 BEGIN LOOP
dbms_output.put_line(x); --print statement
x := x + 10; --statements
IF x > 50 THEN
exit; -- exit condition
END IF;
END LOOP;
END;
```

2. For loop

```
DECLARE
a number(2);
BEGIN
FOR a in 10 .. 20 LOOP
dbms_output.put_line('value of a: ' || a);
```

END LOOP;
END;
/

for loop can only be used in range values and doesn't increment customized incrementations (like incrementing by 2 or 3 ...)
also for loop can be used in situation where we don't know when to end the loop like arrays **BEGIN FOR a in 10 .. LOOP**

example 2:

```
BEGIN
FOR a IN REVERSE 10 .. 20 LOOP
dbms_output.put_line('value of a: ' || a);
END LOOP;
 END;
/
```

3. While loop

```
WHILE a < 20 LOOP
dbms_output.put_line('value of a: ' || a);
END LOOP;
```

4. Nested loop
loop inside loop

STRING FUNCTION:
1. Concat(str1,str2)
2. Ascii(charactervalue)
3.chr(number or ascii value)
4. Lower(str)

5. Upper(str)
6. Substr(str, initialpos, finpos) eturn a new sub string
7.instr(str,chrval) return the index value starting from 1

arrays or varrays
syntax-

```
DECLARE
type namesarray IS VARRAY(5) OF VARCHAR2(10);
BEGIN
names := namesarray('Kavita', 'Pritam', 'Ayan', 'Rishav', 'Aziz');

type variabledt is varray(5) of number(5);
```

procedures in pl/sql
procedure is pl/sql is a sub program that performs a specific task
this procedure can be important when a procedure needs to be repeatedly called
as we can call out this object any number of time with different parameters

also this increases the execution speed compared to a fresh pl/sql block as the code is already compiled.

syntax
```
CREATE [OR REPLACE] PROCEDURE procedure_name
[(parameter_name [IN | OUT | IN OUT]
type [, ...])]
{IS | AS}
BEGIN < procedure_body >
END procedure_name;
```

functions are similar to procedure but they are used in cases where a value need to be returned

```
DECLARE
FUNCTION findMax(x IN number, y IN number)
RETURN number IS
z number;
BEGIN
IF x > y THEN
z:= x;
ELSE
Z:= y;
END IF;
RETURN z;
end;

BEGIN
a:= 23;
b:= 45;
c := findMax(a, b);
dbms_output.put_line(' Maximum of (23,45): ' || c);
END;
/
```

cursor in pl/sql
A **cursor** is a pointer to this context area. A cursor holds the rows (one or more) returned by a SQL statement.

there are 4 function that can be performed in cursors
-%rowcount
-%found
-%notfound

-%isopen

there are 2 types of cursors
1. Implicit
Implicit cursors are automatically created by Oracle whenever an SQL statement is executed.
this is represented by i.e. sql%found

example:

```
DECLARE
total_rows number(2);
BEGIN
UPDATE customers SET salary = salary + 500;
IF sql%notfound THEN
dbms_output.put_line('no customers selected');
ELSIF sql%found THEN
total_rows := sql%rowcount;
dbms_output.put_line( total_rows || ' customers selected ');
END IF;
END;
/
```

2. Explicit cursors
Explicit cursors are programmer-defined cursors for gaining more control over the **context area.**

note: while declaring a explicit cursor it should have select statement only, as its work is to only select some statement

```
DECLARE c_id customers.id%type;
c_name customers.name%type;
c_addr customers.address%type;
CURSOR c_customers is
SELECT id, name, address FROM customers;
BEGIN
OPEN c_customers;
```

LOOP
FETCH c_customers into c_id, c_name, c_addr;
EXIT WHEN c_customers%notfound;
dbms_output.put_line(c_id || ' ' || c_name || ' ' || c_addr);
END LOOP;
CLOSE c_customers;
END;
/

RECORDS
A **record** is a data structure that can hold data items of different kinds. Records consist of different fields, similar to a row of a database table.
types (3)

-table based
DECLARE
customer_rec customers%rowtype;

- cursor based
DECLARE
CURSOR customer_cur
is SELECT id, name, address FROM customers;
customer_rec customer_cur%rowtype;

- user defined
```
   DECLARE

  type books is record

    (title varchar(50),

    author varchar(50),

    subject varchar(100),
    book_id number);
  book1 books;
  book2 books;
```

EXCEPTIONS
An exception is an error condition during a program execution.
types:
1. System defined

```
BEGIN
//Statements
EXCEPTION
  WHEN no_data_found THEN
  statement;
end;
```

2. User defined

```
DECLARE
  ex_invalid_id  EXCEPTION;
BEGIN
  IF c_id <= 0 THEN
    RAISE ex_invalid_id;
  END IF;


EXCEPTION
  WHEN ex_invalid_id THEN
    dbms_output.put_line('ID must be greater than zero!');
```

```
END;
/
```

PACKAGES

are schema objects that has collection of variables, subprograms(procedures, functions) that groups logically related data.

```
CREATE OR REPLACE PACKAGE BODY c_package AS
  PROCEDURE addCustomer(c_id  customers.id%type,
    c_name customers.Name%type,
  IS
  BEGIN
    INSERT INTO customers (id,name)
      VALUES(c_id, c_name);
  END addCustomer;
END c_package;
/

BEGIN
  c_package.addcustomer(7, 'Rajnish');
  c_package.addcustomer(8, 'Subham');
END;
```

/

packages helps in situation where similar programs(procedures or functions) can be called
as every procedure is holded inside the same package and is given to the memory in the same time It givesperformance benefit when we run or call any procedure inside the package for the second time.

TRIGGERS:

Triggers are stored programs, which are automatically executed or fired when some events occur.
Triggers can be executed in response to any of the following events −

- A **database manipulation (DML)** statement (DELETE, INSERT, or UPDATE)
- A **database definition (DDL)** statement (CREATE, ALTER, or DROP).

A **database operation** (SERVERERROR, LOGON, LOGOFF, STARTUP, or SHUTDOWN).

syntax

```
CREATE [OR REPLACE ] TRIGGER trigger_name
        BEFORE |/AFTER /INSTEAD OF
        INSERT OR / UPDATE OR/ DELETE
        [OF col_name]
        ON table_name
[FOR EACH ROW]
WHEN (condition)
DECLARE
  Declaration-statements

BEGIN
  Executable-statements
```

```
EXCEPTION
  Exception-handling-statements
END;
```

//declaring trigger example

```
CREATE OR REPLACE TRIGGER display_salary_changes
    BEFORE DELETE OR INSERT OR UPDATE ON customers
    FOR EACH ROW
    WHEN (NEW.ID > 0)
    DECLARE
    sal_diff number;
    BEGIN
      sal_diff := :NEW.salary  - :OLD.salary;
      dbms_output.put_line('Old salary: ' || :OLD.salary);
      dbms_output.put_line('New salary: ' || :NEW.salary);
      dbms_output.put_line('Salary difference: ' || sal_diff);
    END;
    /
```

this trigger above will be called whenever an update, insert or delete query will be called.