

DATE : 05/06/2024

BATCH : 250

LESSON : SQL

SUBJECT : SQL INTRO





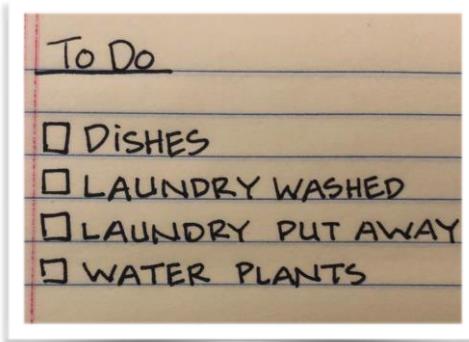
SQL

Structured Query Language

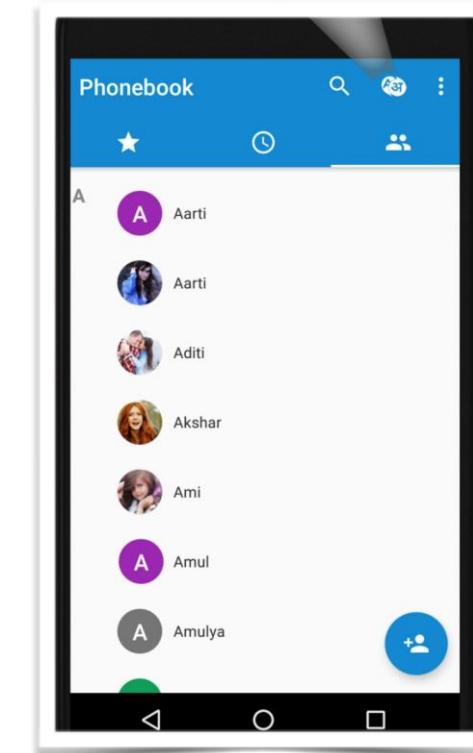
SQL TUTORING SUMMER 2021

WHAT IS DATABASE ?

- 1) Database is a collection of related information
- 2) Database can be stored in different ways
- 3) Most databases use Structured Query Language (SQL) for writing and querying data.



Todo List



Phone book



My 4 best friends



Names of Facebook users



Names of Students in a School

ADVANTAGES OF STORING DATA IN COMPUTER'S MEMORY OR CLOUD

- 1) Huge amount of data can be stored ✓
- 2) Easy to Create, Read, Update, Delete ✓
- 3) Easy to access ✓
- 4) Quick access ✓
- 5) Security ✓

no	adi	soyadi	dyeri	bid
1	Ali	Turan	İstanbul	1
2	Ahmet	Büyük	Ankara	1
3	Leyla	Şahin	İzmir	1
4	Can	Türkoğlu	Manisa	2
5	Aziz	Keskin	İstanbul	2
6	Talat	Şanlı	İzmir	3
7	Kamuran	Kece	Adana	3
8	Turgut	Cemal	Bursa	4



DATABASE VALIDATION TEST

Registration

[Register with Facebook](#) [Register with Twitter](#)

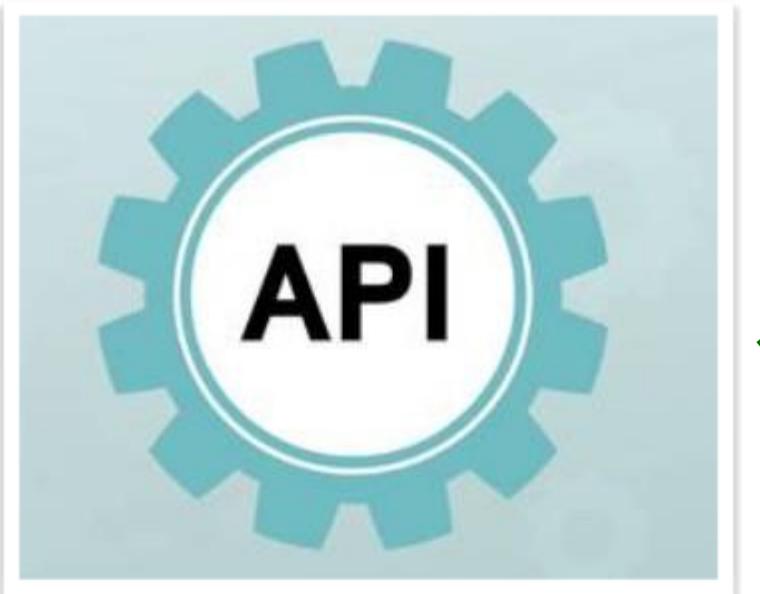
Main

User Name

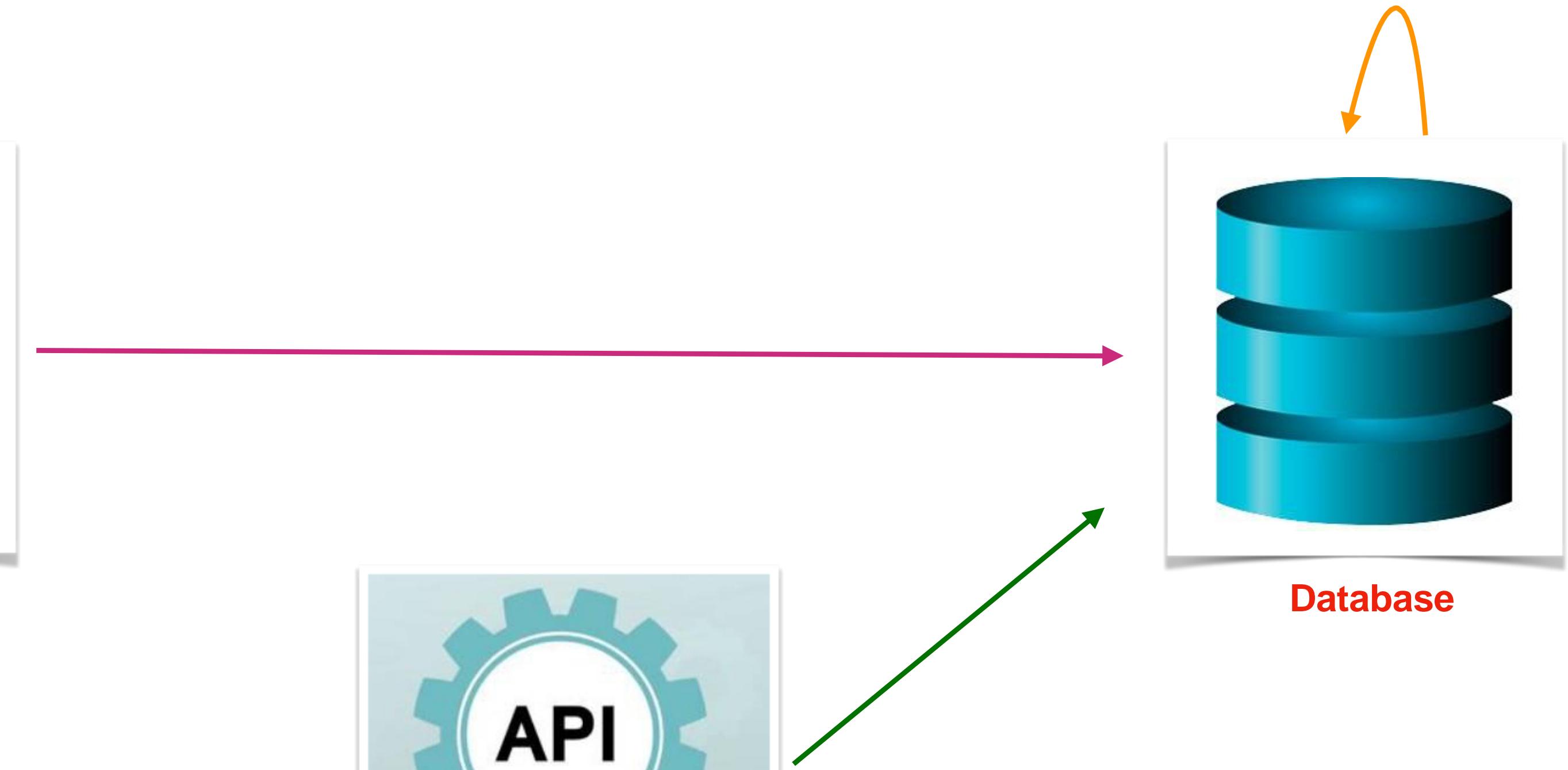
E-mail

Password

User Interface



API



```
public class facebook {  
  
    public static void main(String[] args) {  
        Scanner scan = new Scanner(System.in);  
        System.out.println("Enter your name");  
        String name = scan.nextLine();  
  
        System.out.println("Enter your surname");  
        String surname = scan.nextLine();  
  
        System.out.println("Enter your surname");  
        String email = scan.nextLine();  
  
        System.out.println("Enter your password");  
        String password = scan.nextLine();  
  
        scan.close();  
    }  
}
```

Kaydol

Hızlı ve kolaydır.

Adın Soyadın

Cep telefonu numarası veya e-posta

Yeni şifre

Doğum Tarihi 1 Eki 2020

Cinsiyet Kadın Erkek Özel

Kaydol

Kaydol düğmesine tıklayarak, Koşullarımıza, Veri İlkemizi ve Çerezler İlkemizi kabul etmiş olursun. Bizden SMS Bildirimleri alabilir ve bu bildirimleri istediği zaman durdurabilirsin.



DATA BASE MANAGEMENT SYSTEM (DBMS)

It is the software used to manage, use, develop and maintain databases.

A) DBMS is a special software program which enables its users

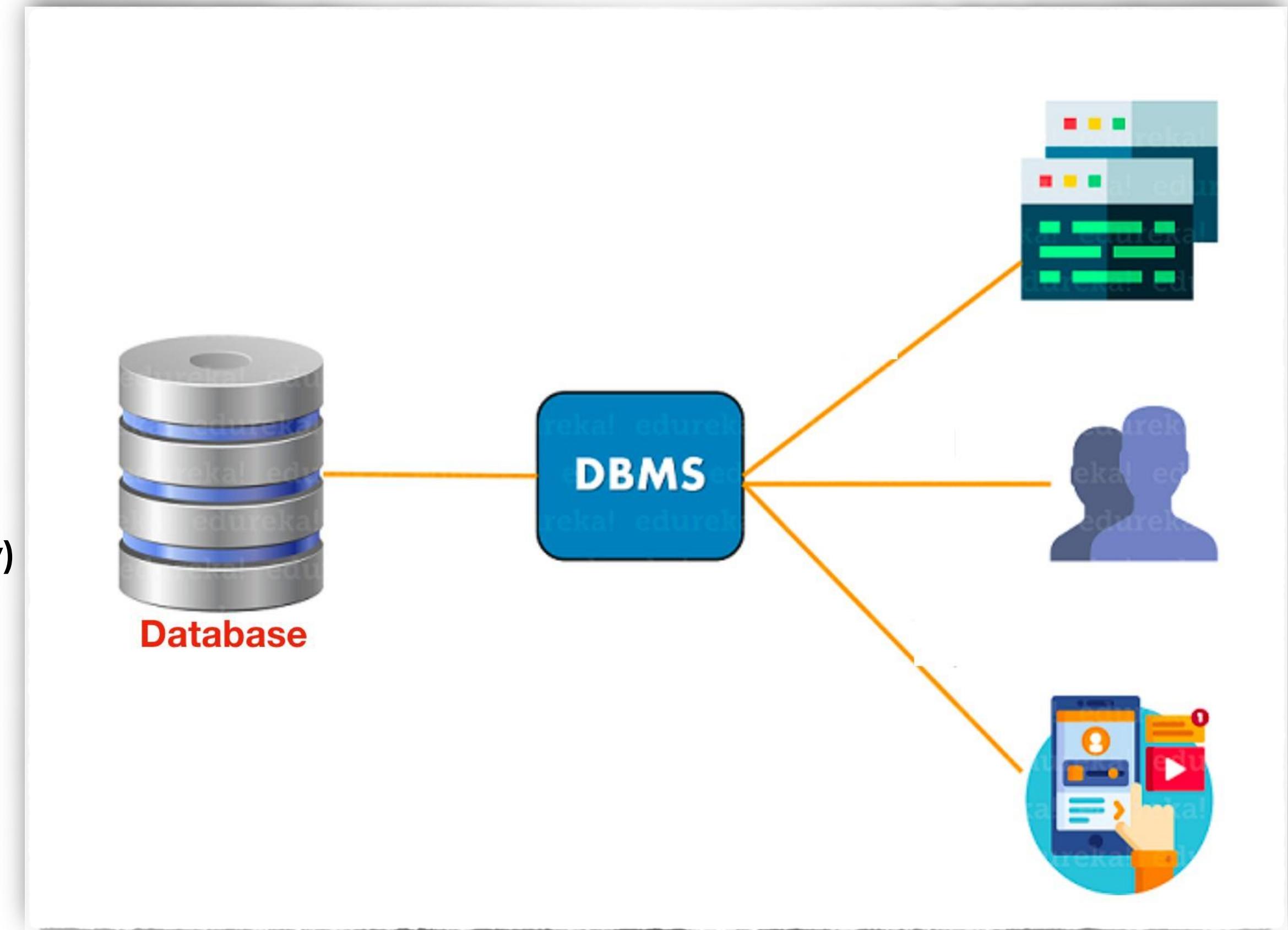
A) To access database,

B) To Create, Read, Update, Delete, (CRUD)

C) To get reports from database,

D) To control access to the database, (Security)

E) To interact with other applications



POPULAR RELATIONAL DATABASES(SQL DATABASE)



SQL Server : Developed by Microsoft

Cons: It can be **expensive** – with the Enterprise level costing thousands of dollars.

Pros: It has **rich user interface** and can **handle large quantities of data**.



MySQL Server : Created by a Swedish Company

Cons: Tends to **stop** working when it's given **too many operations** at a given time.

Pros: It's **free** and **open-source**. There's also **a lot of documentation** and **online support**.



PostgreSQL Server : Created by a computer science professor Michael

Stonebraker. **Cons:** Installation and configuration can be **difficult**.

Pros: If you need **additional features** in PostgreSQL, **you can add it yourself** – a difficult task in most databases.



PL/SQL is a procedural language designed specifically to embrace SQL statements within its syntax. **PL/SQL** program units are compiled by the Oracle Database server and stored inside the database.

Pros: PL/SQL provides high security level.

PL/SQL provides support for Object-Oriented Programming.

TABLES IN SQL (STRUCTURED QUERY LANGUAGE)

Headers	====>	contactID	name	company	email
Row (Record)	====>	1	Bill Gates	Microsoft	bill@XBoxOneRocks.com
Row (Record)	====>	2	Steve Jobs	Apple	steve@rememberNewton.com
Row (Record)	====>	3	Linus Torvalds	Linux Foundation	linus@gnuWho.org
Row (Record)	====>	4	Andy Harris	Wiley Press	andy@aharrisBooks.net

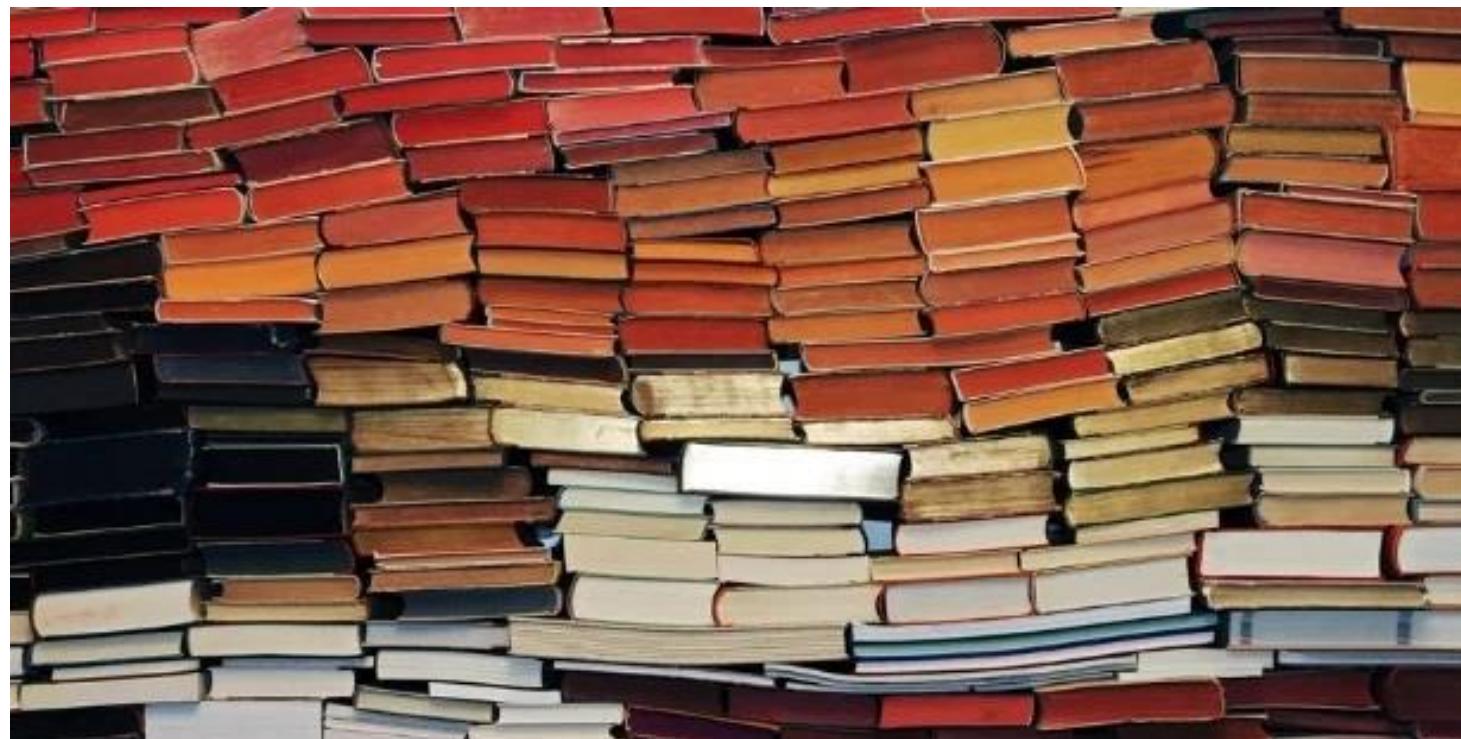
Column (Field) <-->

Column (Field) <-->

Column (Field) <-->

Column (Field) <-->

RELATIONAL DATABASES (SQL DATABASES)



RELATIONAL DATABASES (SQL DATABASES)

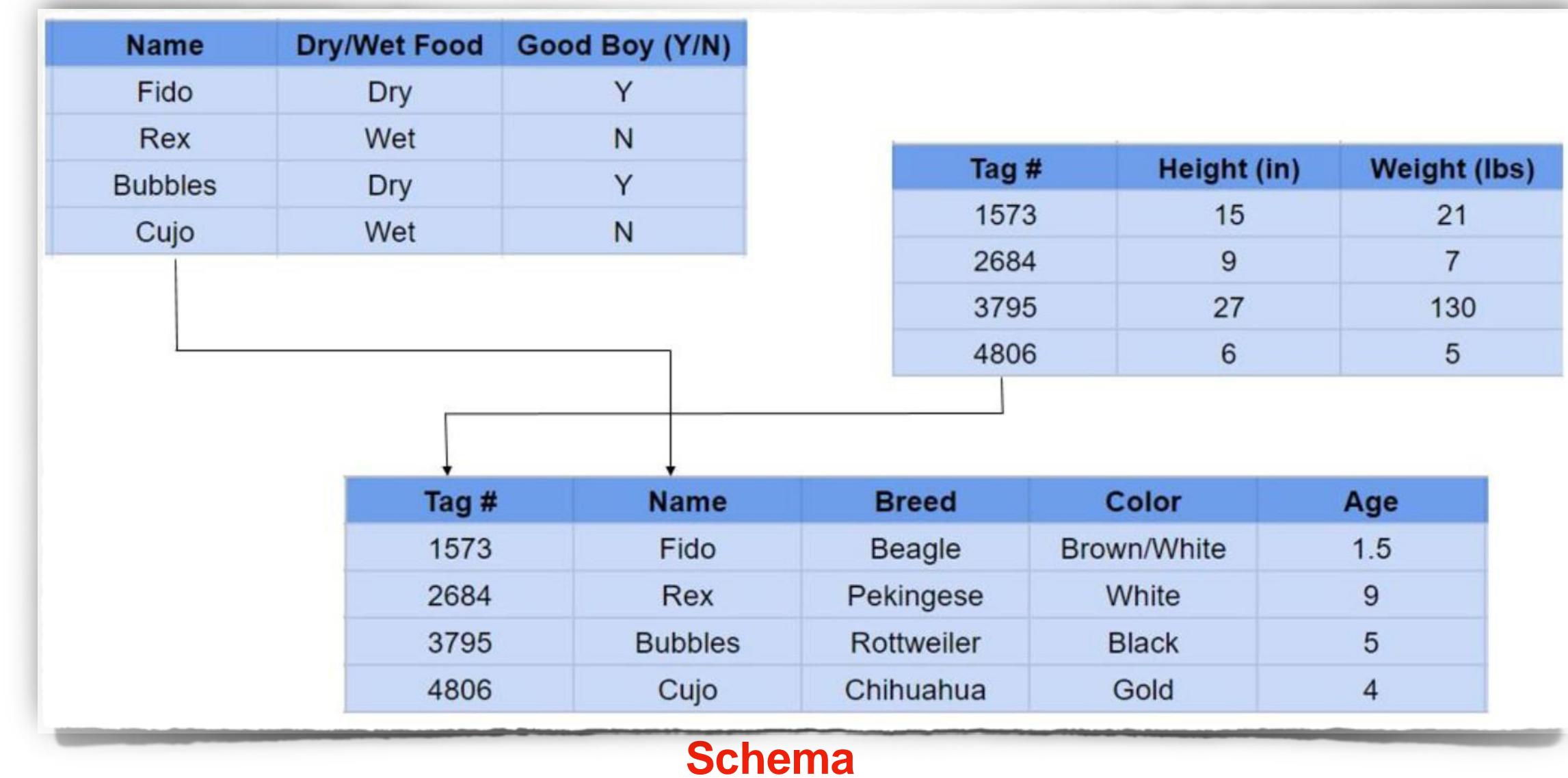
id	ogrenci_adi	ogrenci_soyadi
1	Elif	Türkmen
2	Ayşe	Sarı
3	Ender	Kaya
4	Ali	Demir
5	Adem	Salih

id	ogrenci_id	ders_id
1	1	3
2	1	5
3	2	1
4	3	4
5	4	2
6	4	3

id	ders_adi
1	Matematik
2	Tarih
3	Edebiyat
4	Yazılım
5	İstatistik

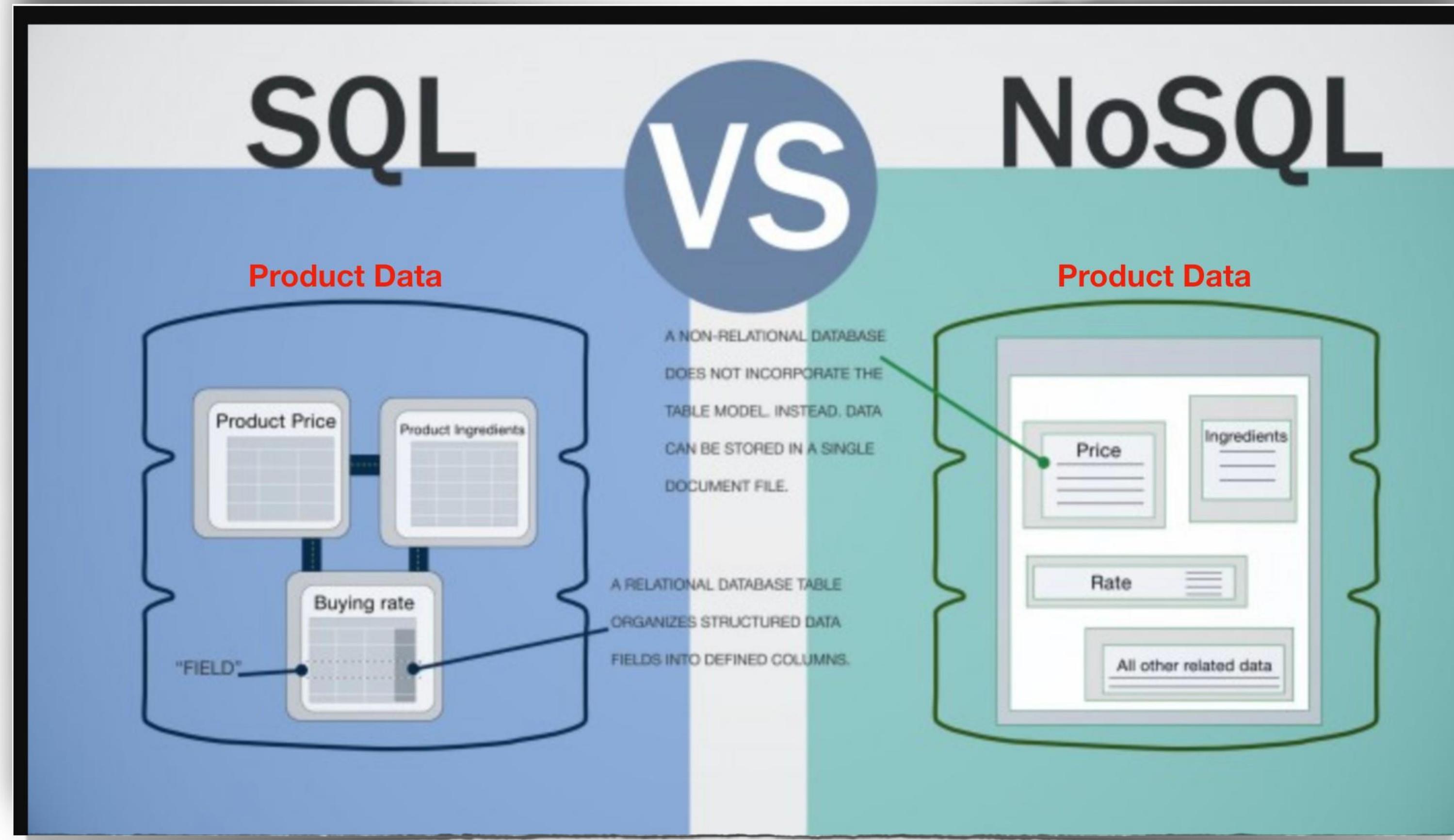
RELATIONAL DATABASES (SQL DATABASES)

- 1) A relational database stores data in tables.
- 2) The relationship between each data point is clear and searching through those relationships is easy.
- 3) The relationship between tables and field types is called a schema.
- 4) Relational Databases are also called SQL Databases. (Structured Query Language)



NON RELATIONAL DATABASES (NON-SQL DATABASES)

A **non-relational** database **does not use** the **tabular schema** of rows and columns like in relational databases



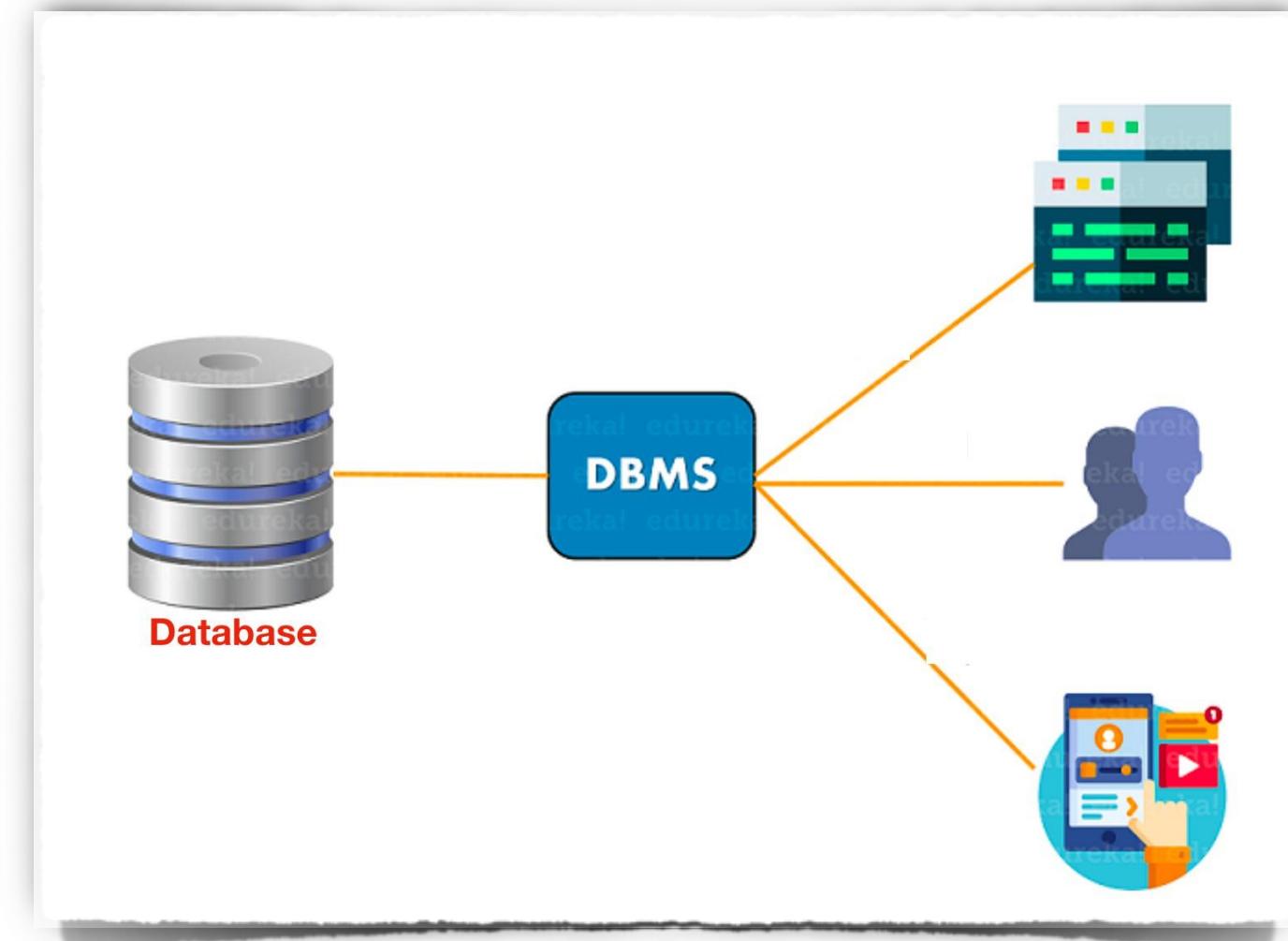
WHAT IS SQL ?

SQL stands for **Structured Query Language**

SQL is a language used for interacting with **Relational Data Base Management Systems (RDBMS)**

By using **SQL** we can;

- 1) Create and Manage databases
- 2) Create and Design database tables
- 3) Create, Read, Update, and Delete data (**CRUD**)
- 4) Perform administration tasks like security, user management etc.



We can use **SQL** for all RDBMS (MySQL, Microsoft SQL, PostgreSQL, Oracle SQL)

The concepts are same but implementation can be slightly different.

MORE ABOUT SQL

SQL is the **combination of 4** different languages;

1) Data Control Language (DCL)

DCL is used **to control privileges in Database**. To perform any operation in the database, such as for creating tables, sequences or views, a user needs privileges.

DCL manages users and permissions

GRANT: It is used to give authority

REVOKE: It is used to revoke permission from the user.

2) Data Definition Language (DDL)

DDL deals with descriptions of the **database schema** (tables, columns, rows) and is used to create and modify the **structure of database objects**

CREATE, ALTER(update), DROP(delete)

3) Data Manipulation Language (DML) ==> For Create, Update, Delete

DML deals with the **manipulation of data** present in the database. For example, insert, update, and delete data.

INSERT, UPDATE, DELETE

4) Data Query Language (DQL) ==> Read

DQL is used to **query the database for information**

DQL is used to **get information that is already stored in database**

SELECT command in DQL, it **retrieves** some or all of the existing records in the database, depending on the defined conditions.

PRIMARY KEY

Primary Keys



StudentId	firstName	lastName	courseId
L0002345	Jim	Black	C002
L0001254	James	Harradine	A004
L0002349	Amanda	Holland	C002
L0001198	Simon	McCloud	S042
L0023487	Peter	Murray	P301
L0018453	Anne	Norris	S042

PRIMARY KEY

Primary Key : Primary keys must contain **UNIQUE** values, and **cannot contain NULL** values.

For data whose all attributes are same, we need **primary key** to differentiate between them

A table can have only **one** primary key; and in the table, this primary key **can consist of multiple columns**

Note: Primary key can be anything, a number, String, character etc.

Note: If you use real values as a primary key like **SSN or email address**, it is called “**Natural Primary Key**” If

you use any values like 1, 2, 3, 4, ..., it is called “**Surrogate Primary Key**.”

Surrogate key values are just numbers.

StudentID	FirstName	LastName
10 ←	John	Walker
11	Tom	Hanks
12	Kevin	Star
13 ←	Carl	Wall
14	Andrei	Apazniak
15	Mark	High
16	Clara	Star
17	John	Ocean
18 ←	John	Walker
19	Pamela	Star
20 ←	Carl	Wall

Email	FirstName	LastName
JWalker@gmail.com	John	Walker
THanks@gmail.com	Tom	Hanks
KStar@gmail.com	Kevin	Star
CWall@gmail.com	Carl	Wall
AApazniak@gmail.com	Andrei	Apazniak
MHigh@gmail.com	Mark	High
CStar@gmail.com	Clara	Star
JOcean@gmail.com	John	Ocean
JWalker01@gmail.com	John	Walker
PStar@gmail.com	Pamela	Star
CWall01@gmail.com	Carl	Wall

primary key can consist of multiple columns

Job_ID	Recruiter	Company
2	Mark Eye	RCG
3	John Ted	RCG
1	Mark Eye	Signature
1	John Ted	InfoLog
1	Cory Al	InfoLog
2	Angela Star	Signature

FOREIGN KEY

A **Foreign Key** is a key used to create **link between two tables**.

A **Foreign Key** is a field (*or collection of columns*) in one table that **refers to the Primary Key in another table**. A table can have **multiple Foreign Keys**

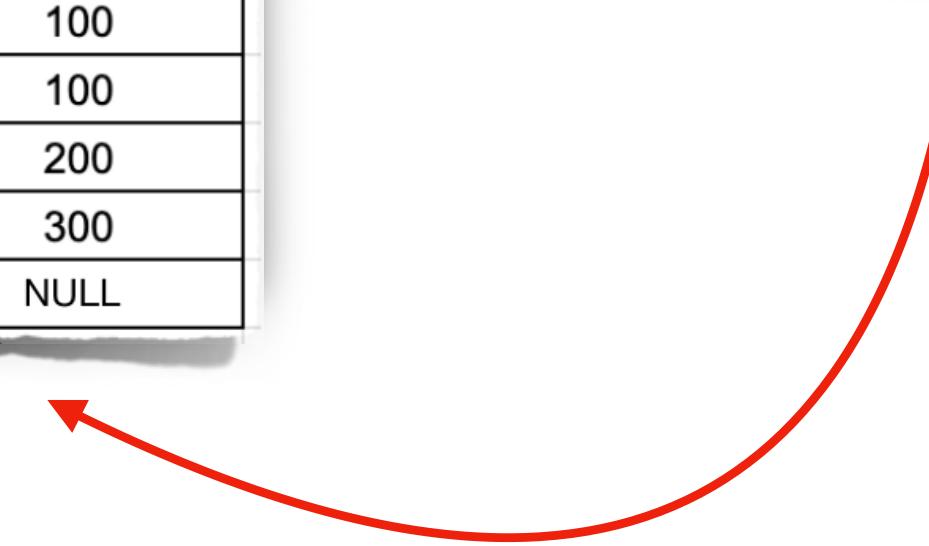
Foreign Key can have **NULL values** and **repeated values**

StudentID	FirstName	LastName	CourseID
10	John	Walker	200
11	Tom	Hanks	400
12	Kevin	Star	400
13	Carl	Wall	200
14	Andrei	Apazniak	300
15	Mark	High	400
16	Clara	Star	100
17	John	Ocean	100
18	John	Walker	200
19	Pamela	Star	300
20	Carl	Wall	NULL

Child Table

CourseID	CourseName	CourseCredit	CourseFee
100	Biology	3	1200
200	Math	3	1200
300	English	2	600
400	Selective	1	200

Parent Table



The "CourseID" column in the "Parent Table" table is the **primary key**.
The "CourseID" column in the "Child Table" table is a **foreign key**.

FOREIGN AND PRIMARY KEY

Note: Foreign key can create a relation between the table and the table itself.

- 1) Who is the Manager of Michael Scott ?
- 2) What is the job name of Angela Martin ?
- 3) What is the average salary of Manual Testers ?
- 4) What is the job name of the highest salary ?

Emp_ID	first_name	last_name	birth_date	Gender	salary	Job_ID	Manager_ID
100	Jan	Levinson	1961-05-11	F	110,000	1	NULL
101	Michael	Scott	1964-03-15	M	75,000	2	100
102	Josh	Porter	1969-09-05	M	78,000	3	100
103	Angela	Martin	1971-06-25	F	63,000	2	101
104	Andy	Bernard	1973-07-22	M	65,000	3	101

Job_ID	Job_Name
2	SDET
3	Manual Tester
1	QE Lead

SQL COMPOSITE KEY

A composite key is a **combination of two or more columns** in a table that can be used to uniquely identify each row in the table when the columns are combined **uniqueness is guaranteed**, but when it taken individually it does not guarantee uniqueness.

Note: Branch_ID and Recruiter are the primary keys for the Job and Recruiter tables; in addition, they are foreign key for the Company table.

The combination of Job_ID and Recruiter foreign keys in Company table is primary key for Company table.

Job_ID	Job_Name
2	SDET
3	Manual Tester
1	QE Lead

Job Table

Recruiter	NumberOfClient
Mark Eye	121
John Ted	283
Cory Al	67
Angela Star	301

Recruiter Table

Job_ID	Recruiter	Company
2	Mark Eye	RCG
3	John Ted	RCG
1	Mark Eye	Signature
1	John Ted	InfoLog
1	Cory Al	InfoLog
2	Angela Star	Signature

Company Table

Difference between “UNIQUE KEY” and “PRIMARY KEY”

Primary Key

Only one primary key is allowed to use in a table.

Primary key does not accept NULL values.

Unique Key

A table can have more than one unique key.

Unique key can accept **multiple NULL** values for column.

Common features of “UNIQUE KEY” and “PRIMARY KEY”

Primary Key

A primary key of one table can be referenced by the foreign key of another table.

Primary key does not allow duplication

Unique Key

Unique keys are also referenced by the foreign key of another table.

Unique key also does not allow duplication except null

WORKING WITH RELATED TABLES

ONE TO ONE

			===== One to One Relation =====				
1) Find the address of the Tom Hanks 2) Find the address of the John Walker 3) Find the address of the student whose ID is 17							
StudentID	FirstName	LastName	StudentID	Street	ZipCode	City	State
10	John	Walker	10	1234 W 23th Street	33018	Hialeah	Florida
11	Tom	Hanks	11	1235 N 3th Street	22145	Austwell	Texas
12	Kevin	Star	12	1236 SE 12th Street	54234	Orange	California
13	Carl	Wall	13	1237 N 5th Street	33018	Hialeah	Florida
14	Andrei	Apazniak	14	1238 SW 53th Street	33026	Miami	Florida
15	Mark	High	15	1239 S 123th Street	22314	Avery	Texas
16	Clara	Star	16	1240 N 1st Street	12345	Arlington	Virginia
17	John	Ocean	17	1241 NW 2nd Street	65432	Pittsburgh	Pensylvania
18	John	Walker	18	1242 W 5th Street	22133	Baytown	Texas
19	Pamela	Star	19	1243 SE 55th Street	74352	Beachwood	Ohio
20	Carl	Wall	20	1244 SW 17th Street	22314	Avery	Texas

WORKING WITH RELATED TABLES

ONE TO MANY

					===== One to Many Relation =====			
1) Find the names of the students who take Biology class 2) Find the names of the students who take Selective class 3) Find the names of the students who take the class whose course fee is 600								
CourseID	CourseName	CourseCredit	CourseFee	InstructorID	StudentID	FirstName	LastName	
100	Biology	3	1200	1	10	John	Walker	
200	Math	3	1200	2	11	Tom	Hanks	
300	English	2	600	3	12	Kevin	Star	
400	Selective	1	200	1	13	Carl	Wall	
					14	Andrei	Apazniak	
					15	Mark	High	
					16	Clara	Star	
					17	John	Ocean	
					18	John	Walker	
					19	Pamela	Star	
					20	Carl	Wall	

WORKING WITH RELATED TABLES

MANY TO MANY

			<==== Many to Many Relation =====>						
			To resolve Many to Many relation we need Linking Table						
1) Find the names of the students whose instructor is Mark Adam 2) Find the names of the instructors of Kevin Star 3) Find the names of the instructors of Pamela Star									
StudentID	FirstName	LastName	StudentID	InstructorID	InstructorID	FirstName	LastName	Phone	Department
10	John	Walker	12	1	1	Mark	Adam	1234567891	Science
11	Tom	Hanks	11	2	2	Eve	Sky	1239876543	Engineering
12	Kevin	Star	12	2	3	Leo	Ocean	1237845691	Language
13	Carl	Wall	13	1	4	Andy	Mark	1232134567	Health
14	Andrei	Apazniak	15	1					
15	Mark	High	17	3					
16	Clara	Star	15	4					
17	John	Ocean							
18	John	Walker							
19	Pamela	Star							
20	Carl	Wall							

SQL DATA TYPES

String Data Types

Data Type	Description
char(size)	<p>Maximum size of 2000 bytes. 1 character uses 1 byte.</p> <p>“size” is the number of characters to store. “char” is used to store character data.</p> <p>Fixed length Strings.</p> <p>The “char” is useful for expressions where the length of characters is always fix like SSN or ZipCode or State Abbreviations (FL, CA, ...)</p>
nchar(size)	<p>Maximum size of 2000 bytes. 1 character uses 2 bytes.</p> <p>“size” is the number of characters to store. “nchar” is used to store Unicode Data.</p> <p>It is often used to store data in different languages. Fixed length Strings.</p>
varchar2(size)	<p>Maximum size of 4000 bytes. 1 character uses 1 byte.</p> <p>“size” is the number of characters to store. Variable length string.</p>
nvarchar2(size)	<p>Maximum size of 8000 bytes. 1 character uses 2 byte.</p> <p>“size” is the number of characters to store. “varchar” uses Non-Unicode data while “nvarchar” uses Unicode Data Variable length string.</p>

Value	CHAR (4)	Storage Required	VARCHAR (4)	Storage Required
''	' '	4 bytes	''	1 byte
'ab'	'ab '	4 bytes	'ab'	3 bytes
'abcd'	'abcd'	4 bytes	'abcd'	5 bytes
'abcdefg'h	'abcd'	4 bytes	'abcd'	5 bytes

NUMERIC DATA TYPES

Data Type	Description
numeric(p, s)	<p>The “Precision” is a number of digits in a number. The “Scale” is the number of digits to the right of the decimal point in a number. For example, for 1234,56 ==> Precision is 6, and Scale is 2.</p> <p>1) “numeric(5, 2)” is a number that has 3 digits before the decimal and 2 digits after the decimal. ==> 123,45 is stored as 123,45</p> <p>2) “numeric(7)” defines a 7 digits number with scale zero. ==> 12345,67 is stored as 12345 Note: “number(7)” and “number(7, 0)” are same.</p> <p>3) “numeric(7, -2)” rounds the numeric value to hundreds. ==> 1234567,89 ==> 1234600</p> <p>4) “numeric(4, 2)” ==> 123,45 ==> Exceeds precision error Note: If the precision is exceeded, SQL will give error</p>

SQL Data Types

Numeric DataTypes

Data Type	Acıklama
SMALLINT	-32.768 ile 32.767 arasında değer alır. Hafızada kapladığı alan: 2 byte
INTEGER	Alabileceğiniz değerler -2147483648 ile 2147483647 arasındadır. Hafızada kapladığı alan: 4 byte.
BIGINT	-9.223.372.036.854.775.808 ile 9.223.372.036.854.775.807 arasında değer alır. “Boyut” ile alabileceğiniz sınırı belirtebiliriz. Hafızada kapladığı alan: 8 byte.
SERIAL	0'dan başlar. Tabloya bir veri girildiğinde bir artırılır, 1 olur. Diğer databaselerde bulunan Auto Increment özelliğinin karşılığıdır. Serial ifadeler eşsiz olacak diye bir kural yoktur.

DATE DATA TYPES

<i>Data Type</i>	<i>Description</i>
DATE	<p>“DATE” stores values that include both date and time with a precision of one second “DATE” stores the year, the month, the day, the hours, the minutes, and the seconds.</p> <p>The standard “Date Format” for input and output is “dd - MMM - yy” like 09 - Aug - 21</p> <p>We can change the format by using “ALTER SESSION SET NLS_DATE_FORMAT = “YYYY-MM-DD” The new date format is 2020 - 04 - 13</p>

BLOB DATA TYPES

<i>Data Type</i>	<i>Description</i>
BLOB	<p>“BLOB” stands for “Binary Large OBjects”</p> <p>“BLOB” is good to store digitized information like images, audios, and videos.</p>

HOW TO CREATE A TABLE

1) Create from Scratch

```
CREATE TABLE students
(
    id number(9),
    name varchar2(50),
    grade number(2),
    address varchar2(100),
    last_modification date
);
```



Columns								
#	Column	Type	Length	Precision	Scale	Nullable	Semantics	
1	ID	NUMBER	22	9	0	Yes		
2	NAME	VARCHAR2	50			Yes	Byte	
3	GRADE	NUMBER	22	2	0	Yes		
4	ADDRESS	VARCHAR2	100			Yes	Byte	
5	LAST_MODIFICATION	DATE	7			Yes		

2) Create from an Existing Table

```
CREATE TABLE studentsIdName
AS
SELECT id, name
FROM students;
```



Columns								
#	Column	Type	Length	Precision	Scale	Nullable	Semantics	
1	ID	NUMBER	22	9	0	Yes		
2	NAME	VARCHAR2	50			Yes	Byte	

Practice Exercise 1:

Create a table called “*suppliers*” that stores “supplier_ID”, “name”, address information which has “street”, “city”, “state”, and “zip_code” columns separately.

Practice Exercise 2:

Create a table called “*suppliers_id_name*” that stores “supplier ID”, “name” by using “*suppliers*” table

CONSTRAINTS

1) How to Enforce a Column not to Accept “repeated” Values

To make “**id**” column “not repeated”, we need to type “**UNIQUE**” after the **id** column data type

```
CREATE TABLE students
(
    id char(11),
    name varchar2(50),
    grade number(3),
    address varchar2(80),
    update_date date
);
```

```
CREATE TABLE students
(
    id char(11) UNIQUE,
    name varchar2(50),
    grade number(3),
    address varchar2(80),
    update_date date
);
```

2) How to Enforce a Column not to Accept “null” Values

To make “**id**” column “**not null**”, we need to type “**not null**” after the **id** column data type

```
CREATE TABLE students
(
    id number(9),
    name varchar2(50),
    grade number(2),
    address varchar2(100),
    last_modification date
);
```



```
CREATE TABLE students
(
    id number(9),
    name varchar2(50) NOT NULL,
    grade number(2),
    address varchar2(100),
    last_modification date
);
```

3) How to Add a “Primary Key” for a Table

- 1) A primary key is a single field or combination of fields that uniquely defines a record.
- 2) A table can have only one primary key.
- 3) None of the fields that are part of the primary key can contain a null value.

To Make “id” Column “primary key”

- 1) We can type “primary key” after the id column data type.
If you want to give a name to constraint you can type “CONSTRAINT constraintName PRIMARY KEY”

```
CREATE TABLE students
(
    id number(9),
    name varchar2(50),
    grade number(2),
    address varchar2(100),
    last_modification date
);
```

```
CREATE TABLE students
(
    id number(9) primary key,
    name varchar2(50),
    grade number(2),
    address varchar2(100),
    last_modification date
);
```

- 2) We can use “CONSTRAINT constraintName PRIMARY KEY(column1, column2, ... column_n)”

```
CREATE TABLE students
(
    id number(9),
    name varchar2(50),
    grade number(2),
    address varchar2(100),
    last_modification date
);
```

```
CREATE TABLE students
(
    id number(9),
    name varchar2(50),
    grade number(2),
    address varchar2(100),
    last_modification date,
    CONSTRAINT id_pk PRIMARY KEY(id)
);
```

Practice Exercise 3:

Create a table called “**cities**” that stores “area code”, “name”, “population”,

“**state**” The “area code” will be “**primary key**”

Add “**primary key**” by using first method.

Practice Exercise 4:

Create a table called “**teachers**” that stores “SSN”, “name”, “subject”,

“**gender**” The “SSN” will be “**primary key**”

Add “**primary key**” by using second method.

4) How to Add “foreign key” to a Table

A **Foreign Key** is a key used to create **link between two tables**.

A **Foreign Key** is a column (*or collection of column*) in one table that **refers to the Primary Key** in another table. The **referenced table** is called the **parent table** while the **table with the foreign key** is called the **child table**.

A table can have many **Foreign Keys** Foreign Key can have NULL value

Syntax: `CONSTRAINT constraintName FOREIGN KEY(column1, column2, ...) REFERENCES parentTableName(column1, column2, ...)`

```
CREATE TABLE students
(
    id number(9),
    name varchar2(50),
    grade number(2),
    address varchar2(100),
    last_modification date
);
```

Child
Table

```
CREATE TABLE studentPhoneNumber
(
    studentId number(9),
    PhoneNumber varchar2(10),
    CONSTRAINT studentId_fk FOREIGN KEY(studentId) REFERENCES students(id)
);
```

Parent
Table

Note 1: If “**Parent Table**” does not have same student id with the “**Child Table**” you cannot insert data

Note 2: You cannot drop “**Parent Table**” without dropping the “**Child Table**” You need to drop “**Child Table**” first, then you can drop “**Parent Table**”

Practice Exercise 5:

Create a table called “**supplier**” that stores “**supplier_id**”, “**supplier_name**”, “**contact_name**” and make “**supplier_id**” as **primary key**.

Create another table called “**products**” that stores “**supplier_id**” and “**product_id**” and make “**supplier_id**” as **foreign key**.

```
CREATE TABLE supplier
(
    supplier_id number(10) not null,
    supplier_name varchar2(50) not null,
    contact_name varchar2(50),
    CONSTRAINT supplier_pk PRIMARY KEY (supplier_id)
);
```

```
CREATE TABLE products
(
    supplier_id number(10),
    product_id number(10),
    CONSTRAINT fk_supplier FOREIGN KEY (supplier_id) REFERENCES supplier(supplier_id)
);
```

Practice Exercise 6:

Create a table called “**supplier**” that stores “**supplier_id**”, “**supplier_name**”, “**contact_name**” and make the combination of “**supplier_id**” and “**supplier_name**” as **primary key**.

Create another table called “**products**” that stores “**supplier_id**” and “**product_id**” and make the combination of “**supplier_id**” and “**supplier_name**” as **foreign key**.

```
CREATE TABLE supplier
(
    supplier_id number(10) not null,
    supplier_name varchar2(50) not null,
    contact_name varchar2(50),
    CONSTRAINT supplier_pk PRIMARY KEY (supplier_id, supplier_name)
);
```

```
CREATE TABLE products
(
    product_id number(10),
    supplier_id number(10),
    supplier_name varchar2(50) not null,
    CONSTRAINT fk_supplier FOREIGN KEY (supplier_id, supplier_name) REFERENCES supplier(supplier_id, supplier_name)
);
```

5) How to use Check Constraint

A **check constraint** allows you to specify a condition on each row in a table.
A **check constraint** can be defined in either a **SQL CREATE TABLE statement** or a **SQL ALTER TABLE statement**.

1) CREATE TABLE

```
suppliers (
    supplier_id numeric(4),
    supplier_name varchar2(50),
    CONSTRAINT
    check_supplier_id
    CHECK (supplier_id BETWEEN 100 AND 9999)
);
```

We've created a check constraint on the suppliers table called `check_supplier_id`.

This constraint ensures that the `supplier_id` field contains values **between 100 and 9999**.

2) CREATE TABLE

```
suppliers (
    supplier_id numeric(4),
    supplier_name varchar2(50),
    CONSTRAINT
    check_supplier_name
    CHECK (supplier_name = upper(supplier_name))
);
```

We've created a check constraint called `check_supplier_name`.

This constraint ensures that the `supplier_name` column always **contains uppercase characters**.

How to Insert Data Into a Table

The Oracle **INSERT INTO** statement is used to insert a single record or multiple records into a table in

```
CREATE TABLE students
(
    id number(9),
    name varchar2(50) NOT NULL,
    grade number(2),
    address varchar2(100),
    last_modification date
);
```

1) To insert values for all columns

```
INSERT INTO students VALUES(123456789, 'John Walker', 11, '1234 W 12th TER Addison Texas 75001', '14-Apr-2020');
```

ID	NAME	GRADE	ADDRESS	LAST_MODIFICATION
123456789	John Walker	11	1234 W 12th TER Addison Texas 75001	14-APR-20

2) To insert values for some selected columns

```
INSERT INTO students(id, name) VALUES(234567890, 'John Walker');
```

ID	NAME	GRADE	ADDRESS	LAST_MODIFICATION
234567890	John Walker	-	-	-

Note: When inserting records into a table using the **INSERT INTO** statement, you must provide a value for every NOT NULL column.

```
CREATE TABLE students  
(  
    id number(9),  
    name varchar2(50) NOT NULL,  
    grade number(2),  
    address varchar2(100),  
    last_modification date  
);
```



```
→ INSERT INTO students(id, grade) VALUES(123456789, 11);
```

```
ORA-01400: cannot insert NULL into ("SQL_LFGUHVRSSOWWDACLEMHRMQGCJQ"."STUDENTS"."NAME") ORA-06512: at "SYS.DBMS_SQL", line 1721
```

Practice Exercise 7:

Create an Oracle table called “**teachers**” that stores “SSN”, “name”, “subject”, “gender”

Based on the “**teachers**” table, insert a contact record whose SSN is 234 43 1223, name is Jane Smith, subject is Mathematics, and gender is female.

Practice Exercise 8:

Based on the “**teachers**” table, insert a contact record whose SSN is 567 59 7624, name is Leo Mark

HOW TO USE UPDATE SET

The **UPDATE SET** statement is used to update existing records in a table.

1)

```
CREATE TABLE supplier
(
    supplier_id number(10),
    supplier_name varchar2(50),
    contact_name varchar2(50),
    CONSTRAINT supplier_pk PRIMARY KEY (supplier_id, supplier_name)
);
```



```
INSERT INTO supplier VALUES(1, 'IBM', 'John Walker');
INSERT INTO supplier VALUES(2, 'APPLE', 'Steve Max');
INSERT INTO supplier VALUES(3, 'SAMSUNG', 'Tae Shaun');
```



SUPPLIER_ID	SUPPLIER_NAME	CONTACT_NAME
1	IBM	John Walker
2	APPLE	Steve Max
3	SAMSUNG	Tae Shaun

```
UPDATE supplier
SET supplier_name = 'LINUX',
    contact_name = 'Alex Leo'
WHERE supplier_id=1;
```



SUPPLIER_ID	SUPPLIER_NAME	CONTACT_NAME
1	LINUX	Alex Leo
2	APPLE	Steve Max
3	SAMSUNG	Tae Shaun

2)

```
UPDATE supplier
SET supplier_name = 'LG',
    contact_name = 'El Ci'
WHERE supplier_id<3;
```



SUPPLIER_ID	SUPPLIER_NAME	CONTACT_NAME
1	LG	El Ci
2	LG	El Ci
3	SAMSUNG	Tae Shaun

3)

```
CREATE TABLE workers
(
    id CHAR(5),
    name VARCHAR2(50),
    salary NUMBER(5),
    CONSTRAINT id3_pk PRIMARY KEY(id)
);
```

```
INSERT INTO workers VALUES('10001', 'Ali Can', 12000);
INSERT INTO workers VALUES('10002', 'Veli Han', 2000);
INSERT INTO workers VALUES('10003', 'Mary Star', 7000);
INSERT INTO workers VALUES('10004', 'Angie Ocean', 8500);
```

ID	NAME	SALARY
1 10001	Ali Can	12000
2 10002	Veli Han	2000
3 10003	Mary Star	7000
4 10004	Angie Ocean	8500

Example: Company decided to increase the salary of Veli Han. However, instead of setting a specific price, they want to make it 40% less than the highest salary.

```
UPDATE workers
SET salary = (SELECT MAX(salary)*0.6
               FROM
             workers) WHERE name =
'Veli Han';
```

ID	NAME	SALARY
1 10001	Ali Can	12000
2 10002	Veli Han	7200
3 10003	Mary Star	7000
4 10004	Angie Ocean	8500

4)

```
CREATE TABLE workers
(
    id CHAR(5),
    name VARCHAR2(50),
    salary NUMBER(5),
    CONSTRAINT id3_pk PRIMARY KEY(id)
);
```

```
INSERT INTO workers VALUES('10001', 'Ali Can', 12000);
INSERT INTO workers VALUES('10002', 'Veli Han', 2000);
INSERT INTO workers VALUES('10003', 'Mary Star', 7000);
INSERT INTO workers VALUES('10004', 'Angie Ocean', 8500);
```

ID	NAME	SALARY
10001	Ali Can	12000
10002	Veli Han	2000
10003	Mary Star	7000
10004	Angie Ocean	8500

```
CREATE TABLE parents
(
    id CHAR(5),
    num_of_kids NUMBER(2),
    total_income NUMBER(5),
    CONSTRAINT id3_fk FOREIGN KEY(id) REFERENCES workers(id)
);
```

```
INSERT INTO parents VALUES('10001', 4, 17000);
INSERT INTO parents VALUES('10002', 2, 5000);
INSERT INTO parents VALUES('10003', 1, 7000);
INSERT INTO parents VALUES('10004', 1, 10000);
```

Example: If the salary equals to total income increase the salary 20 percent.

```
UPDATE workers
SET salary = (SELECT total_income*1.2
               FROM parents
              WHERE workers.salary = parents.total_income
            );
```

```
WHERE salary = (SELECT total_income
                  FROM parents
                 WHERE workers.salary = parents.total_income
                );
```

ID	NAME	SALARY
10001	Ali Can	12000
10002	Veli Han	2000
10003	Mary Star	8400
10004	Angie Ocean	8500

Practice Exercise 9:

- a) Create a table called “*students*” that stores “*student_id*”, “*student_name*”, “*student_grade*”, “*student_gpa*”, “*school_name*”
- b) Insert 5 different data with 2.6, 1.9, 3.2, 3.8, 3.5 GPA scores.
- c) Update the student names whose GPAs are more than 3.0 to “*Gifted Student*”.

Practice Exercise 10:

- a) Create a table called “*students*” that stores “*student_id*”, “*student_name*”, “*student_grade*”, “*school_name*”
- b) Insert 5 different data with 2.6, 1.9, 3.2, 3.8, 3.5 GPA scores.
- c) Create a table called “*parents*” that stores “*student_id*”, “*parent_name*”, “*school_name*”
- d) Insert 5 different data with at least 2 same school names with the students table.
- e) Update the *student names* in the *students* table with the *parent name* in the *parents* table when the *school name* in the *students* table matches the *school name* in the *parents* table.

```
CREATE TABLE suppliers
(
supplier_id number(11) PRIMARY KEY,
supplier_name varchar2(50),
contact_name varchar2(50)
);
```



```
INSERT INTO suppliers VALUES(100, 'IBM', 'Ali Can');
INSERT INTO suppliers VALUES(101, 'APPLE', 'Merve Temiz');
INSERT INTO suppliers VALUES(102, 'SAMSUNG', 'Kemal Can');
INSERT INTO suppliers VALUES(103, 'LG', 'Ali Can');
```

```
CREATE TABLE products
(
supplier_id number(11),
product_id number(11),
product_name varchar2(50),
costumer_name varchar2(50),
CONSTRAINT supplier_id_fk FOREIGN KEY(supplier_id) REFERENCES suppliers(supplier_id)
);
```



```
INSERT INTO products VALUES(100, 1001,'Laptop', 'Suleyman');
INSERT INTO products VALUES(101, 1002,'iPad', 'Fatma');
INSERT INTO products VALUES(102, 1003,'TV', 'Ramazan');
INSERT INTO products VALUES(103, 1004,'Phone', 'Ali Can');
```

Practice Exercise 11:

According to the given tables do the followings

- Change the product which Ali Can purchased to the supplier name which Merve Temiz is contact person
- Change the customer name who purchased TV to the contact name of Apple

“IS NULL” Condition

```
CREATE TABLE  
people  
ssn char(9),  
name varchar2(50),  
address  
varchar2(50)  
);
```

```
INSERT INTO people VALUES(123456789, 'Mark Star', 'Florida');  
INSERT INTO people VALUES(234567890, 'Angie Way', 'Virginia');  
INSERT INTO people VALUES(345678901, 'Maryy Tien', 'New Jersey');  
INSERT INTO people(ssn, address) VALUES(456789012, 'Michigan');  
INSERT INTO people(ssn, address)  
VALUES(567890123, 'California');  
INSERT INTO people(ssn, name)  
VALUES(567890123, 'California');
```

SSN	NAME	ADDRESS
1 123456789	Mark Star	Florida
2 234567890	Angie Way	Virginia
3 345678901	Maryy Tien	New Jersey
4 456789012	(null)	Michigan
5 567890123	(null)	California
6 567890123	California	(null)

Table name is
“people”

Example: Return all records from the *people* table where the *name* contains a null value.

```
SELECT *  
FROM people  
WHERE name IS NULL OR address IS NULL;
```

SSN	NAME	ADDRESS
1 456789012	(null)	Michigan
2 567890123	(null)	California
3 567890123	California	(null)

1) **Example:** Update all null names to “No Name” from the *people* table

```
UPDATE people  
SET name = 'No  
Name' WHERE name  
IS NULL;
```

SSN	NAME	ADDRESS
1 123456789	Mark Star	Florida
2 234567890	Angie Way	Virginia
3 345678901	Maryy Tien	New Jersey
4 456789012	No Name	Michigan
5 567890123	No Name	California
6 567890123	California	(null)

```
CREATE TABLE  
people  
ssn char(9),  
name varchar2(50),  
address  
varchar2(50)  
);
```

```
INSERT INTO people VALUES(123456789, 'Mark Star',  
'Florida'); INSERT INTO people VALUES(234567890, 'Angie  
Way', null); INSERT INTO people VALUES(345678901, 'Maryy  
Tien', null); INSERT INTO people(ssn, address)  
VALUES(456789012, null);  
INSERT INTO people(ssn, address) VALUES(567890123,  
'California');
```

2) **Example:** Update all nulls in different columns to 'Not inserted yet!'

```
UPDATE people  
SET name = COALESCE(name, 'Name is not inserted yet!'),  
address = COALESCE(address, 'Address is not inserted  
yet');
```

HOW TO DELETE DATA FROM A TABLE

1) “**DELETE FROM** students” deletes all inserted data inside the table, but it does not delete the table.

After using “**DELETE FROM** students”, you will have an empty table.

```
CREATE TABLE students  
(  
    id number(9),  
    name varchar2(50),  
    state varchar2(50),  
    last_modification date  
);
```

```
INSERT INTO students VALUES(123456789, 'John Walker', 'Texas', '14-Apr-2020');  
INSERT INTO students VALUES(234567890, 'Eddie Murphy', 'Florida', '15-Apr-2020');  
INSERT INTO students VALUES(345678901, 'Adam Eve', 'New York', '16-Apr-2020');
```

ID	NAME	STATE	LAST_MODIFICATION
123456789	John Walker	Texas	14-APR-20
234567890	Eddie Murphy	Florida	15-APR-20
345678901	Adam Eve	New York	16-APR-20

```
DELETE FROM students;
```

ID	NAME	STATE	LAST_MODIFICATION
<i>Empty Table</i>			

2) “**DELETE FROM** students **WHERE** name = ‘John Walker’ ” deletes the data whose name is John Walker.

```
CREATE TABLE students  
(  
    id number(9),  
    name varchar2(50),  
    state varchar2(50),  
    last_modification date  
);
```

```
INSERT INTO students VALUES(123456789, 'John Walker', 'Texas', '14-Apr-2020');  
INSERT INTO students VALUES(234567890, 'Eddie Murphy', 'Florida', '15-Apr-2020');  
INSERT INTO students VALUES(345678901, 'Adam Eve', 'New York', '16-Apr-2020');
```

ID	NAME	STATE	LAST_MODIFICATION
123456789	John Walker	Texas	14-APR-20
234567890	Eddie Murphy	Florida	15-APR-20
345678901	Adam Eve	New York	16-APR-20

```
DELETE FROM students WHERE name = 'John Walker';
```

ID	NAME	STATE	LAST_MODIFICATION
234567890	Eddie Murphy	Florida	15-APR-20
345678901	Adam Eve	New York	16-APR-20

3) " DELETE FROM students WHERE name = 'John Walker' OR state = 'New York' " deletes the data whose name is John Walker.

```
CREATE TABLE students
(
    id number(9),
    name varchar2(50),
    state varchar2(50),
    last_modification date
);
```

```
INSERT INTO students VALUES(123456789, 'John Walker', 'Texas', '14-Apr-2020');
INSERT INTO students VALUES(234567890, 'Eddie Murphy', 'Florida', '15-Apr-2020');
INSERT INTO students VALUES(345678901, 'Adam Eve', 'New York', '16-Apr-2020');
```

ID	NAME	STATE	LAST_MODIFICATION
123456789	John Walker	Texas	14-APR-20
234567890	Eddie Murphy	Florida	15-APR-20
345678901	Adam Eve	New York	16-APR-20

```
DELETE FROM students WHERE name = 'John Walker' OR state = 'New York';
```

ID	NAME	STATE	LAST_MODIFICATION
234567890	Eddie Murphy	Florida	15-APR-20

4
CREATE TABLE people
(
 ssn char(9),
 name varchar2(50),
 address varchar2(50)
);

```
INSERT INTO people VALUES(123456789, 'Mark Star', 'Florida');
INSERT INTO people VALUES(234567890, 'Angie Way', null);
INSERT INTO people VALUES(345678901, 'Maryy Tien', null);
INSERT INTO people(ssn, address) VALUES(456789012, null);
INSERT INTO people(ssn, address) VALUES(567890123, 'California');
```

Example: Delete the records whose ADDRESSES are NOT null
DELETE FROM people
WHERE address IS NOT NULL;

Example: Delete all records whose names or addresses are null
DELETE FROM people
WHERE name IS NULL OR address IS NULL;

Review

Question

SSN	NAME	ADDRESS
123456789	Mark Star	Florida
234567890	Angie Way	Virginia
345678901	Maryy Tien	New Jersey

Table name is “people”

- 1) Create the given table by using SQL Queries
- 2) Update “Virginia” to “Pennsylvania ”
- 3) Delete 3rd row from the table
- 4) Drop the table

Note: Use **SELECT * FROM people;** to see the table on the console.

“Truncate” Statement

“Truncating” a table is a fast way to clear out records from a table if you don't need to worry about rolling back.

- 1) The **TRUNCATE TABLE** statement is used to remove all records from a table in Oracle.
- 2) We cannot use “**WHERE**” clause in **TRUNCATE**
- 3) It performs the same function as a **DELETE** statement without a “**WHERE**” clause.

TRUNCATE TABLE customers;

=

DELETE FROM customers;

Note: The main difference between the two is that you can roll back the **DELETE FROM** statement, but you can't roll back the **TRUNCATE TABLE** statement.

HOW TO DROP (DELETES TABLE CONTENTS AND TABLE STRUCTURE) A TABLE

```
CREATE TABLE students  
(  
    id number(9),  
    name varchar2(50),  
    grade number(2),  
    address varchar2(100),  
    last_modification date  
);
```

→ **DROP TABLE students;**

Table with all contents and structure moved to the trash

```
CREATE TABLE students  
(  
    id number(9),  
    name varchar2(50),  
    grade number(2),  
    address varchar2(100),  
    last_modification date  
);
```

→ **DROP TABLE students PURGE;**

The PURGE option will purge the table and its dependent objects so that they do not appear in the recycle bin.

Warning: The risk of specifying the PURGE option is that you will not be able to recover the table.

Benefit of PURGE: You can ensure that sensitive data will not be left sitting in the recycle bin.

OPERATORS TO USE IN WHERE FILTER

WHERE clause is used to filter the results from a **SELECT**, **INSERT**, **UPDATE**, or **DELETE** statement.

“ = “ ==> Equal to sign

“ > “ ==> Greater than sign

“ < “ ==> Less than sign

“ >= “ ==> Greater than or equal to sign

“ <= “ ==> Less than or equal to sign

“ < > “ ==> Not Equal to sign

“AND” ==> And operator

“OR” ==> Or operator

“SELECT” STATEMENT

CREATE TABLE workers

```
(  
    id CHAR(5),  
    name VARCHAR2(50),  
    salary NUMBER(5),  
    CONSTRAINT id4_pk PRIMARY  
    KEY(id)  
);
```

1) To get all records from a table **SELECT ***
FROM workers;

4) To get a specific record from a column **SELECT ***
FROM workers
WHERE id =
10002;

6) Select the record whose salary is the highest **SELECT ***
FROM workers
WHERE salary = (**SELECT MAX(salary)**
FROM workers
);

```
INSERT INTO workers VALUES(10001, 'Ali Can',  
12000); INSERT INTO workers VALUES(10002, 'Veli  
Han', 2000); INSERT INTO workers VALUES(10003,  
'Mary Star', 7000);  
INSERT INTO workers VALUES(10004, 'Angie Ocean',  
8500);
```

2) To get specific column/columns from a table **SELECT name**
FROM workers;

```
SELECT name,  
salary FROM  
workers;
```

5) Select all records whose names lengths are greater than 8 **SELECT ***
FROM workers
WHERE LENGTH(name)>8;

7) Select the name whose salary is the lowest **SELECT name**
FROM workers
WHERE salary = (**SELECT MIN(salary)**
FROM workers
);

3)To get a specific data from a column **SELECT name**
FROM workers
WHERE id =
10001;

8) Select name and salary of the workers whose salary is the lowest or the highest
highest **SELECT** name, salary

FROM workers

WHERE salary = (**SELECT MIN(salary)** **FROM** workers) **OR** salary = (**SELECT MAX(salary)** **FROM** workers);

9) Select id of the workers whose salary is 8500 and name is

Mary Star **SELECT** id

FROM workers

WHERE salary = 8500 **AND** name = 'Mary Star';

10) Select the second highest salary from workers

table **SELECT MAX(salary)** **AS**

second_highest_salary **FROM** workers

WHERE salary < (**SELECT MAX(salary)**

FROM workers

);

11) Select the second minimum salary from the workers

table **SELECT MIN(salary)** **AS** second_min_salary

FROM workers

WHERE salary > (**SELECT MIN (salary)**

FROM workers

);

12) Select the third highest salary from workers

```
table SELECT MAX(salary) AS  
third_highest_salary FROM workers  
WHERE salary < (SELECT MAX(salary)  
    FROM workers  
    WHERE salary < (SELECT MAX(salary)  
        FROM workers  
    )  
);
```

13) Select all records whose salary is the second highest salary from workers

```
table      SELECT *  
          FROM workers  
WHERE salary = (SELECT MAX(salary)  
    FROM workers  
    WHERE salary < (SELECT MAX(salary)  
        FROM workers  
    )  
);
```

14) Select all records whose salary is the third highest salary from workers

```
table      SELECT *  
          FROM workers  
WHERE salary = ( SELECT MAX(salary)  
    FROM workers  
    WHERE salary < (SELECT MAX(salary)  
        FROM workers  
    WHERE salary < (SELECT MAX(salary)  
        FROM workers  
    )  
);
```

Practice Exercise

11:

ID	NAME	STATE	GPA
123456789	John Walker	Texas	2.8
234567890	Eddie Murphy	Florida	3.2
345678901	Adam Eve	New York	3.5
456789012	Alex Tien	New York	3.8
567890123	Chris Matala	Virginia	4

- a) Create the given table
- b) Select all fields from the *students* table whose *gpa* is greater than 3.1 or school name is “Texas”
- c) Select students name from the *students* table whose *gpa* is less than 3.5 and state is “Florida”
- d) Select students names and student ids from the *students* table whose *gpa* is between 2.8 and 3.5
- e) Select all fields from the *students* table whose *state* is “New York”, and *gpa* is greater than 3.3, and *gpa* is less than 3.7
- f) Select all fields from the *students* table whose *state* is “New York” and *gpa* is greater than 3.7, or *gpa* is less than 3.3

“IN” CONDITION

IN condition is used to help **reduce** the need to use multiple **OR** conditions in a **SELECT**, **INSERT**, **UPDATE**, or **DELETE** statement.

CREATE TABLE

```
customers_products
product_id number(10),
customer_name
varchar2(50),
product_name
varchar2(50)
);
```

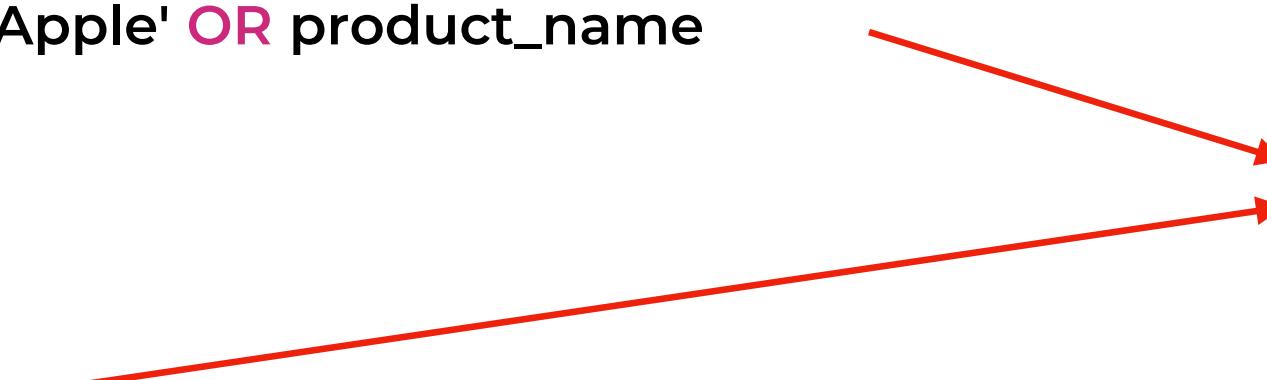


```
INSERT INTO customers_products VALUES (10, 'Mark',
'Orange'); INSERT INTO customers_products VALUES (10,
'Mark', 'Orange'); INSERT INTO customers_products VALUES
(20, 'John', 'Apple'); INSERT INTO customers_products VALUES
(30, 'Amy', 'Palm'); INSERT INTO customers_products VALUES
(20, 'Mark', 'Apple'); INSERT INTO customers_products
VALUES (10, 'Adem', 'Orange'); INSERT INTO
customers_products VALUES (40, 'John', 'Apricot'); INSERT
INTO customers_products VALUES (20, 'Eddie', 'Apple');
```

PRODUCT_ID	CUSTOMER_NAME	PRODUCT_NAME
10	Mark	Orange
10	Mark	Orange
20	John	Apple
30	Amy	Palm
20	Mark	Apple
10	Adem	Orange
40	John	Apricot
20	Eddie	Apple

```
SELECT *
FROM customers_products
WHERE product_name ='Orange' OR product_name ='Apple' OR product_name
='Apricot';
```

```
SELECT *
FROM customers_products
WHERE product_name IN ('Orange', 'Apple',
'Apricot');
```



PRODUCT_ID	CUSTOMER_NAME	PRODUCT_NAME
10	Mark	Orange
10	Mark	Orange
20	John	Apple
20	Mark	Apple
10	Adem	Orange
40	John	Apricot
20	Eddie	Apple

“NOT IN” CONDITION

Question: Select records whose product name is not Orange or Apple or Palm

SELECT *

FROM customers_products

WHERE product_name NOT IN('Orange', 'Apple', 'Palm');

	PRODUCT_ID	CUSTOMER_NAME	PRODUCT_NAME
1	40	John	Apricot

“BETWEEN” CONDITION

```
CREATE TABLE customers_likes  
(  
    product_id number(10),  
    customer_name  
    varchar2(50), liked_product  
    varchar2(50)  
);
```

```
INSERT INTO customers_likes VALUES (10, 'Mark', 'Orange');  
INSERT INTO customers_likes VALUES (50, 'Mark', 'Pineapple');  
INSERT INTO customers_likes VALUES (60, 'John', 'Avocado');  
INSERT INTO customers_likes VALUES (30, 'Lary', 'Cherries');  
INSERT INTO customers_likes VALUES (20, 'Mark', 'Apple');  
INSERT INTO customers_likes VALUES (10, 'Adem', 'Orange');  
INSERT INTO customers_likes VALUES (40, 'John', 'Apricot');  
INSERT INTO customers_likes VALUES (20, 'Eddie', 'Apple');
```

PRODUCT_ID	CUSTOMER_NAME	LIKED_PRODUCT
10	Mark	Orange
50	Mark	Pineapple
60	John	Avocado
30	Lary	Cherries
20	Mark	Apple
10	Adem	Orange
40	John	Apricot
20	Eddie	Apple

```
SELECT *  
FROM customers_likes  
WHERE product_id BETWEEN 20 AND  
40;
```

```
SELECT *  
FROM customers_likes  
WHERE product_id >= 20 AND product_id <=  
40;
```

PRODUCT_ID	CUSTOMER_NAME	LIKED_PRODUCT
30	Lary	Cherries
20	Mark	Apple
40	John	Apricot
20	Eddie	Apple

Note: 20 and 40 are inclusive for BETWEEN condition

“NOT BETWEEN” CONDITION

```
CREATE TABLE customers_likes  
(  
    product_id number(10), →  
    customer_name  
    varchar2(50), liked_product  
    varchar2(50)  
);
```

```
INSERT INTO customers_likes VALUES (10, 'Mark', 'Orange');  
INSERT INTO customers_likes VALUES (50, 'Mark', 'Pineapple');  
INSERT INTO customers_likes VALUES (60, 'John', 'Avocado');  
INSERT INTO customers_likes VALUES (30, 'Lary', 'Cherries');  
INSERT INTO customers_likes VALUES (20, 'Mark', 'Apple');  
INSERT INTO customers_likes VALUES (10, 'Adem', 'Orange');  
INSERT INTO customers_likes VALUES (40, 'John', 'Apricot');  
INSERT INTO customers_likes VALUES (20, 'Eddie', 'Apple');
```

PRODUCT_ID	CUSTOMER_NAME	LIKED_PRODUCT
10	Mark	Orange
50	Mark	Pineapple
60	John	Avocado
30	Lary	Cherries
20	Mark	Apple
10	Adem	Orange
40	John	Apricot
20	Eddie	Apple

```
SELECT *  
FROM customers_likes  
WHERE product_id NOT BETWEEN 20 AND  
40;
```

```
SELECT *  
FROM customers_likes  
WHERE product_id < 20 OR product_id >  
40;
```

PRODUCT_ID	CUSTOMER_NAME	LIKED_PRODUCT
10	Mark	Orange
50	Mark	Pineapple
60	John	Avocado
10	Adem	Orange

Note: 20 and 40 are exclusive for NOT BETWEEN condition

REVIEW QUESTIONS 10 MINUTES

- 1) What is the difference between “DELETE” and “TRUNCATE”**
- 2) What is the difference between “DELETE” and “DROP”**
- 3) What is the difference between “DROP” and “DROP PURGE”**
- 4) Type a query which gives the same result with the following query**

```
SELECT *  
FROM students  
WHERE age>=8 AND age<=17;
```

- 5) Type a query which gives the same result with the following query**

```
SELECT *  
FROM students  
WHERE age<8 OR age>17;
```

- 6) Type a query which gives the same result with the following query**

```
SELECT *  
FROM students  
WHERE grade = 6 OR grade = 7 OR grade = 8 OR grade = 9;
```

ANSWERS OF REVIEW QUESTIONS

1) What is the difference between “DELETE” and “TRUNCATE”

- A) TRUNCATE removes **all rows** from a table. DELETE command is used to remove **all or specific rows** from a table based on WHERE condition.
- B) If you use TRUNCATE rollback is **not** possible. For DELETE rollback is possible.
- C) We **cannot** use WHERE clause with TRUNCATE but we can use WHERE with DELETE.

2) What is the difference between “DELETE” and “DROP”

- A) DROP command removes a table from the database while DELETE removes records from a table.

3) What is the difference between “DROP” and “DROP PURGE”

- A) The **DROP** will drop the table and place it into the recycle bin.
DROP with **PURGE** will drop the table and flush it out from the recycle bin also.

4) SELECT *

```
FROM students  
WHERE age>=8 AND age<=17;
```



```
SELECT *  
FROM students  
WHERE age BETWEEN 8 AND  
17;
```

5) SELECT *

```
FROM students  
WHERE age<8 OR age>17;
```



```
SELECT *  
FROM students  
WHERE age NOT BETWEEN 8 AND  
17;
```

6) SELECT *

```
FROM students  
WHERE grade = 6 OR grade = 7 OR grade = 8 OR grade =  
9;
```

```
SELECT *  
FROM students  
WHERE grade IN (6, 7, 8,  
9);
```

“EXISTS” CONDITION

EXISTS condition is used in combination with a subquery and is considered "to be met" if the subquery returns at least one row. It can be used in a **SELECT**, **INSERT**, **UPDATE**, or **DELETE** statement.

```
CREATE TABLE customers_products
(
    product_id number(10),
    customer_name
    varchar2(50),
    product_name
    varchar2(50)
);
```

```
INSERT INTO customers_products VALUES (10, 'Mark', 'Orange');
INSERT INTO customers_products VALUES (10, 'Mark', 'Orange');
INSERT INTO customers_products VALUES (20, 'John', 'Apple');
INSERT INTO customers_products VALUES (30, 'Amy', 'Palm');
INSERT INTO customers_products VALUES (20, 'Mark', 'Apple');
INSERT INTO customers_products VALUES (10, 'Adem', 'Orange');
INSERT INTO customers_products VALUES (40, 'John', 'Apricot');
INSERT INTO customers_products VALUES (20, 'Eddie', 'Apple');
```

PRODUCT_ID	CUSTOMER_NAME	PRODUCT_NAME
10	Mark	Orange
10	Mark	Orange
20	John	Apple
30	Amy	Palm
20	Mark	Apple
10	Adem	Orange
40	John	Apricot
20	Eddie	Apple

```
CREATE TABLE customers_likes
(
    product_id number(10),
    customer_name
    varchar2(50), liked_product
    varchar2(50)
);
```

```
INSERT INTO customers_likes VALUES (10, 'Mark', 'Orange');
INSERT INTO customers_likes VALUES (50, 'Mark', 'Pineapple');
INSERT INTO customers_likes VALUES (60, 'John', 'Avocado');
INSERT INTO customers_likes VALUES (30, 'Lary', 'Cherries');
INSERT INTO customers_likes VALUES (20, 'Mark', 'Apple');
INSERT INTO customers_likes VALUES (10, 'Adem', 'Orange');
INSERT INTO customers_likes VALUES (40, 'John', 'Apricot');
INSERT INTO customers_likes VALUES (20, 'Eddie', 'Apple');
```

PRODUCT_ID	CUSTOMER_NAME	LIKED_PRODUCT
10	Mark	Orange
50	Mark	Pineapple
60	John	Avocado
30	Lary	Cherries
20	Mark	Apple
10	Adem	Orange
40	John	Apricot
20	Eddie	Apple

```
SELECT customer_name
FROM
customers_products
WHERE EXISTS (SELECT product_id FROM customers_likes WHERE customers_products.product_id =
customers_likes.product_id);
```

CUSTOMER_NAME
Mark
Mark
Adem
Amy
John
Mark
Eddie
John

“SUBQUERIES”

SUBQUERY IS A QUERY WITHIN A QUERY

```
CREATE TABLE employees
(
    id number(9),
    name varchar2(50),
    state varchar2(50),
    salary number(20),
    company
    varchar2(20)
);
```

```
INSERT INTO employees VALUES(123456789, 'John Walker', 'Florida', 2500, 'IBM');
INSERT INTO employees VALUES(234567890, 'Brad Pitt', 'Florida', 1500, 'APPLE');
INSERT INTO employees VALUES(345678901, 'Eddie Murphy', 'Texas', 3000, 'IBM');
INSERT INTO employees VALUES(456789012, 'Eddie Murphy', 'Virginia', 1000, 'GOOGLE');
INSERT INTO employees VALUES(567890123, 'Eddie Murphy', 'Texas', 7000, 'MICROSOFT');
INSERT INTO employees VALUES(456789012, 'Brad Pitt', 'Texas', 1500, 'GOOGLE');
INSERT INTO employees VALUES(123456710, 'Mark Stone', 'Pennsylvania', 2500, 'IBM');
```

ID	NAME	STATE	SALARY	COMPANY
123456789	John Walker	Florida	2500	IBM
234567890	Brad Pitt	Florida	1500	APPLE
345678901	Eddie Murphy	Texas	3000	IBM
456789012	Eddie Murphy	Virginia	1000	GOOGLE
567890123	Eddie Murphy	Texas	7000	MICROSOFT
456789012	Brad Pitt	Texas	1500	GOOGLE
123456710	Mark Stone	Pennsylvania	2500	IBM

```
CREATE TABLE companies
(
    company_id number(9),
    company varchar2(20),
    number_of_employees
);number(20)
```

```
INSERT INTO companies VALUES(100, 'IBM', 12000);
INSERT INTO companies VALUES(101, 'GOOGLE', 18000);
INSERT INTO companies VALUES(102, 'MICROSOFT', 10000);
INSERT INTO companies VALUES(100, 'APPLE', 21000);
```

COMPANY_ID	COMPANY	NUMBER_OF_EMPLOYEES
100	IBM	12000
101	GOOGLE	18000
102	MICROSOFT	10000
100	APPLE	21000

Example: Find the employee and company names whose company has more than 15000 employees

```
SELECT name, company
FROM employees
WHERE company IN (SELECT company
                   FROM companies
                   WHERE number_of_employees > 15000);
```

NAME	COMPANY
Eddie Murphy	GOOGLE
Brad Pitt	GOOGLE
Brad Pitt	APPLE

2) SUBQUERY in the SELECT clause

A **SUBQUERY** in the select clause must return a single value.

Therefore, an aggregate function such as **SUM**, **COUNT**, **MIN**, or **MAX** is commonly used in the subquery.

```
CREATE TABLE employees
(
    id number(9),
    name varchar2(50),
    state varchar2(50),
    salary number(20),
    company varchar2(20)
);
```

```
INSERT INTO employees VALUES(123456789, 'John Walker', 'Florida', 2500, 'IBM');
INSERT INTO employees VALUES(234567890, 'Brad Pitt', 'Florida', 1500, 'APPLE');
INSERT INTO employees VALUES(345678901, 'Eddie Murphy', 'Texas', 3000, 'IBM');
INSERT INTO employees VALUES(456789012, 'Eddie Murphy', 'Virginia', 1000, 'GOOGLE');
INSERT INTO employees VALUES(567890123, 'Eddie Murphy', 'Texas', 7000, 'MICROSOFT');
INSERT INTO employees VALUES(456789012, 'Brad Pitt', 'Texas', 1500, 'GOOGLE');
INSERT INTO employees VALUES(123456710, 'Mark Stone', 'Pennsylvania', 2500, 'IBM');
```

ID	NAME	STATE	SALARY	COMPANY
123456789	John Walker	Florida	2500	IBM
234567890	Brad Pitt	Florida	1500	APPLE
345678901	Eddie Murphy	Texas	3000	IBM
456789012	Eddie Murphy	Virginia	1000	GOOGLE
567890123	Eddie Murphy	Texas	7000	MICROSOFT
456789012	Brad Pitt	Texas	1500	GOOGLE
123456710	Mark Stone	Pennsylvania	2500	IBM

```
CREATE TABLE companies
(
    company_id number(9),
    company varchar2(20),
    number_of_employees
);number(20)
```

```
INSERT INTO companies VALUES(100, 'IBM', 12000);
INSERT INTO companies VALUES(101, 'GOOGLE', 18000);
INSERT INTO companies VALUES(102, 'MICROSOFT', 10000);
INSERT INTO companies VALUES(100, 'APPLE', 21000);
```

COMPANY_ID	COMPANY	NUMBER_OF_EMPLOYEES
100	IBM	12000
101	GOOGLE	18000
102	MICROSOFT	10000
100	APPLE	21000

Example: Find the company number of employees and average salary for every company.

SELECT company, number_of_employees,

```
(SELECT AVG(salary) FROM  
employees  
WHERE companies.company = employees.company) Average_Salary_Per_Company
```

FROM companies;

Example: Find the name of the companies, company ids, and the number of states for every company

```
SELECT company, company_id, (SELECT COUNT(state)
    FROM employees
    WHERE companies.company = employees.company
    ) number_of_states
FROM companies;
```

Example: Find the name of the companies, company ids, maximum and minimum salaries per company.

```
SELECT company, company_id, (SELECT MAX(salary)
    FROM employees
    WHERE companies.company = employees.company
    ) max_salary,
    (SELECT MIN(salary) FROM
    employees
    WHERE companies.company = employees.company
    ) min_salary
FROM companies;
```

LIKE CONDITION

LIKE condition allows wildcards to be used in the WHERE Clause of a SELECT, INSERT, UPDATE, or DELETE statement. This allows you to perform pattern matching.

Wildcard Characters

- 1) % => Represents zero or more characters

WHERE customer_name LIKE 'J%' ==> Finds any values that starts with "J"

WHERE customer_name LIKE '%e' ==> Finds any values that ends with "e"

WHERE customer_name LIKE '%an%' ==> Finds any values that have "an" in any position

CREATE TABLE customers

```
(  
customer_id number(10) UNIQUE,  
customer_name varchar2(50) NOT  
NULL, income number(6)  
);
```

INSERT INTO customers (customer_id, customer_name, income) VALUES (1001, 'John', 62000);

INSERT INTO customers (customer_id, customer_name, income) VALUES (1002, 'Jane', 57500);

INSERT INTO customers (customer_id, customer_name, income) VALUES (1003, 'Brad', 71000);

INSERT INTO customers (customer_id, customer_name, income) VALUES (1004, 'Manse', 42000);



CUSTOMER_ID	CUSTOMER_NAME	INCOME
1001	John	62000
1002	Jane	57500
1003	Brad	71000
1004	Manse	42000

2) _ => Represents just one character

WHERE CustomerName **LIKE** '_ohn' ==> Finds all customer's name starting with any character, followed by "ohn"

WHERE CustomerName **LIKE** '_a_e' ==> Finds all sized 4 customer's name whose 2nd character is "a", and 4th character is "e" **WHERE** CustomerName **LIKE** '_r%' ==> Finds any values that have "r" in the second position

WHERE CustomerName **LIKE** 'M_%_%_%' ==> Finds any values that starts with "M" and are at least 4 characters in length

WHERE CustomerName **LIKE** 'B%d' ==> Finds any values that starts with "B" and ends with "d"

```
CREATE TABLE customers
(
customer_id number(10) UNIQUE,
customer_name varchar2(50) NOT
NULL, income number(6)
);
```



```
INSERT INTO customers (customer_id, customer_name,
income) VALUES (1001, 'John', 62000);
```

```
INSERT INTO customers (customer_id, customer_name,
income) VALUES (1002, 'Jane', 57500);
```

```
INSERT INTO customers (customer_id, customer_name,
income) VALUES (1003, 'Brad', 71000);
```

```
INSERT INTO customers (customer_id, customer_name,
income) VALUES (1004, 'Manse', 42000);
```

CUSTOMER_ID	CUSTOMER_NAME	INCOME
1001	John	62000
1002	Jane	57500
1003	Brad	71000
1004	Manse	42000

3) REGEXP_LIKE Condition

WHERE REGEXP_LIKE(word, 'h[oa]t') ==> Finds "hot" and "hat", but **not** "hit"

WHERE REGEXP_LIKE(word, 'h(o|a)t') ==> Finds "hot" and "hat", but **not** "hit"

WHERE REGEXP_LIKE(word,'h[a-c]t') ==> Finds "hat" and "hbt" and "hct"

WHERE REGEXP_LIKE(word,'h(a|b|c)t') ==> Finds "hat" and "hbt" and "hct"

WHERE REGEXP_LIKE(word,['au'](*)) ==> Finds all contains "a" or "u"
"hat" and "selena" and "yusuf" and "adem"

WHERE REGEXP_LIKE(word,'^asy') ==> Finds all start with "a" or "s" or "y"
"adem" and "selena" and "yusuf"

WHERE REGEXP_LIKE(word,'(*) f \$') ==> Finds all end with "f" ==> "yusuf"

```
CREATE TABLE words
(
    word_id      number(10)
    UNIQUE, word  varchar2(50)
    NOT          NULL,
    number_of_letters
    number(6)
);
```

```
INSERT INTO words VALUES (1001, 'hot', 3);
INSERT INTO words VALUES (1002, 'hat', 3);
INSERT INTO words VALUES (1003, 'hit', 3);
INSERT INTO words VALUES (1004, 'hbt', 3);
INSERT INTO words VALUES (1008, 'hct', 3);
INSERT INTO words VALUES (1005, 'adem',
4); INSERT INTO words VALUES (1006,
'selena', 6); INSERT INTO words VALUES
(1007, 'yusuf', 5);
```

WORD_ID	WORD	NUMBER_OF_LETTERS
1001	hot	3
1002	hat	3
1003	hit	3
1004	hbt	3
1006	selena	6
1007	yusuf	5
1005	adem	4
1008	hct	3

WHERE REGEXP_LIKE(word,'a') ==> Retrieve all of the names that contain the letter 'a'

WHERE REGEXP_LIKE(word,'a', 'i') ==> By specifying the letter 'i' (as the third argument of the REGEXP_LIKE function) we can make a **case insensitive** search. It is case sensitive as default in Oracle

WHERE REGEXP_LIKE(word,'at') ==> Retrieve all of the names that contain the letter 'at'

WHERE REGEXP_LIKE(word,'[a-i]d') ==> Retrieve all words that contain a letter in the range of 'a' and 'i', followed by the letter 'd'.

WHERE REGEXP_LIKE(word, '[a-i].[e]') ==> Retrieve all words that contain a letter in the range of 'a' and 'i', followed by any character, followed by the letter 'e'.

WHERE REGEXP_LIKE(word,'[a-i]..[m]') ==> Retrieve all words that contain a letter in the range of 'a' and 'i', followed by any two characters, followed by the letter 'm'.

WHERE REGEXP_LIKE(word,'[e{2}]') ==> The curly brackets are used to specify an exact number of occurrences, for example display all words that contain double 'e' letters.

NOT LIKE Condition

WHERE word NOT LIKE 'h%' ==> Finds all words which do

NOT start with 'h'. **WHERE word NOT LIKE '%t'** ==> Finds all words which do **NOT** end with 't'.

WHERE word NOT LIKE '%a%' ==> Finds all words which do **NOT** contain "a" in any position

WHERE word NOT LIKE '_us%' ==> Finds all customer's name starting with any character, **NOT** followed by "us"

WHERE NOT REGEXP_LIKE(word, '[_ead](*)'); ==> Find all words starting with any character, **NOT** followed by 'e' or 'a' or 'd'

(

CREATE TABLE words

```
word_id      number(10)
UNIQUE, word varchar2(50)
NOT          NULL,
number_of_letters
number(6)
);
```

```
INSERT INTO words VALUES (1001, 'hot', 3);
INSERT INTO words VALUES (1002, 'hat', 3);
INSERT INTO words VALUES (1003, 'hit', 3);
INSERT INTO words VALUES (1004, 'hbt', 3);
INSERT INTO words VALUES (1004, 'hct', 3);
INSERT INTO words VALUES (1005, 'adem', 4);
INSERT INTO words VALUES (1006, 'selena', 6);
INSERT INTO words VALUES (1007, 'yusuf', 5);
```

WORD_ID	WORD	NUMBER_OF_LETTERS
1001	hot	3
1002	hat	3
1003	hit	3
1004	hbt	3
1006	selena	6
1007	yusuf	5
1005	adem	4
1008	hct	3

“ORDER BY” Clause

The **ORDER BY** clause is used to sort the records in result set. The **ORDER BY** clause can only be used in **SELECT statements**.

1)

ID	NAME	STATE	GPA
123456789	John Walker	Texas	2.8
234567890	Eddie Murphy	Florida	3.2
345678901	Adam Eve	New York	3.5
456789012	Alex Tien	New York	3.8
567890123	Chris Matala	Virginia	4



```
SELECT *  
FROM students  
ORDER BY  
name;
```

ID	NAME	STATE	GPA
345678901	Adam Eve	New York	3.5
456789012	Alex Tien	New York	3.8
567890123	Chris Matala	Virginia	4
234567890	Eddie Murphy	Florida	3.2
123456789	John Walker	Texas	2.8

2)

ID	NAME	STATE	GPA
123456789	John Walker	Texas	2.8
234567890	Eddie Murphy	Florida	3.2
345678901	Adam Eve	New York	3.5
456789012	Alex Tien	New York	3.8
567890123	Chris Matala	Virginia	4

A red arrow pointing from the left side of the second table towards the center, indicating the flow from the data to the query.

```
SELECT name  
FROM students  
WHERE gpa = 2.8 OR state  
='Florida'  
ORDER BY name;
```

NAME
Eddie Murphy
John Walker

3)

ID	NAME	STATE	GPA
123456789	John Walker	Texas	2.8
234567890	Eddie Murphy	Florida	3.2
345678901	Adam Eve	New York	3.5
456789012	Alex Tien	New York	3.8
567890123	Chris Matala	Virginia	4

SELECT *
FROM students
ORDER BY name
DESC;

ID	NAME	STATE	GPA
123456789	John Walker	Texas	2.8
234567890	Eddie Murphy	Florida	3.2
567890123	Chris Matala	Virginia	4
456789012	Alex Tien	New York	3.8
345678901	Adam Eve	New York	3.5

4)

ID	NAME	STATE	GPA
123456789	John Walker	Texas	2.8
234567890	Eddie Murphy	Florida	3.2
345678901	Adam Eve	New York	3.5
456789012	Alex Tien	New York	3.8
567890123	Chris Matala	Virginia	4

SELECT *
FROM students
ORDER BY 3
DESC;

ID	NAME	STATE	GPA
567890123	Chris Matala	Virginia	4
123456789	John Walker	Texas	2.8
345678901	Adam Eve	New York	3.5
456789012	Alex Tien	New York	3.8
234567890	Eddie Murphy	Florida	3.2

Number of columns

5)

ID	NAME	STATE	GPA
123456789	John Walker	Texas	2.8
234567890	Eddie Murphy	Florida	3.2
345678901	Adam Eve	New York	3.5
456789012	Zeyna Rose	New York	3.8
567890123	Chris Matala	Virginia	4
456789012	Brad Pitt	New York	3.8



```
SELECT *  
FROM students  
ORDER BY 3 DESC, 2  
ASC;
```



ID	NAME	STATE	GPA
567890123	Chris Matala	Virginia	4
123456789	John Walker	Texas	2.8
345678901	Adam Eve	New York	3.5
456789012	Brad Pitt	New York	3.8
456789012	Zeyna Rose	New York	3.8
234567890	Eddie Murphy	Florida	3.2

ORDER BY will return all records sorted by the *3rd field in descending order*, with a secondary sort by *2nd field in ascending order*.

“ALIASES”

```
CREATE TABLE employees  
(  
    employee_id number(9),  
    employee_first_name  
    varchar2(20)  
    employee_last_name  
    varchar2(20)  
);
```

```
INSERT INTO employees VALUES(14, 'Chris', 'Tae');  
INSERT INTO employees VALUES(11, 'John', 'Walker');  
INSERT INTO employees VALUES(12, 'Amy', 'Star');  
INSERT INTO employees VALUES(13, 'Brad', 'Pitt');  
INSERT INTO employees VALUES(15, 'Chris', 'Way');
```

EMPLOYEE_ID	EMPLOYEE_FIRST_NAME	EMPLOYEE_LAST_NAME
14	Chris	Tae
11	John	Walker
12	Amy	Star
13	Brad	Pitt
15	Chris	Way

1) **SELECT employee_id AS id, employee_first_name AS first_name, employee_last_name AS last_name FROM employees;**

ID	FIRST_NAME	LAST_NAME
14	Chris	Tae
11	John	Walker
12	Amy	Star
13	Brad	Pitt
15	Chris	Way

2) **SELECT employee_id AS id, employee_first_name || employee_last_name AS full_name FROM employees;**

ID	FULL_NAME
14	ChrisTae
11	JohnWalker
12	AmyStar
13	BradPitt
15	ChrisWay

```
CREATE TABLE employees
(
    employee_id number(9),
    employee_first_name
    varchar2(20),
    employee_last_name
    varchar2(20)
);
```

```
CREATE TABLE addresses
(
    employee_id
    number(9), street →
    varchar2(20),
    city
    varchar2(20),
    state char(2),
    zipcode char(5)
);
```

3) **SELECT e.employee_first_name, e.employee_last_name, a.city
FROM employees e, addresses a
WHERE e.employee_id = a.employee_id;**

```
INSERT INTO employees VALUES(14, 'Chris', 'Tae');
INSERT INTO employees VALUES(11, 'John', 'Walker');
INSERT INTO employees VALUES(12, 'Amy', 'Star');
INSERT INTO employees VALUES(13, 'Brad', 'Pitt');
INSERT INTO employees VALUES(15, 'Chris', 'Way');
```

EMPLOYEE_ID	EMPLOYEE_FIRST_NAME	EMPLOYEE_LAST_NAME
14	Chris	Tae
11	John	Walker
12	Amy	Star
13	Brad	Pitt
15	Chris	Way

```
INSERT INTO addresses VALUES(11, '32nd Star 1234', 'Miami', 'FL', '33018');
INSERT INTO addresses VALUES(12, '23rd Rain 567', 'Jacksonville', 'FL', '32256');
INSERT INTO addresses VALUES(13, '5th Snow 765', 'Hialeah', 'VA', '20121');
INSERT INTO addresses VALUES(14, '3rd Man 12', 'Weston', 'MI', '12345');
INSERT INTO addresses VALUES(15, '11th Chris 12', 'St. Johns', 'FL', '32259');
```

EMPLOYEE_ID	STREET	CITY	STATE	ZIPCODE
11	32nd Star 1234	Miami	FL	33018
12	23rd Rain 567	Jacksonville	FL	32256
13	5th Snow 765	Hialeah	VA	20121
14	3rd Man 12	Weston	MI	12345
15	11th Chris 12	St. Johns	FL	32259

EMPLOYEE_FIRST_NAME	EMPLOYEE_LAST_NAME	CITY
John	Walker	Miami
Amy	Star	Jacksonville
Brad	Pitt	Hialeah
Chris	Tae	Weston
Chris	Way	St. Johns

“GROUP BY” CLAUSE

GROUP BY clause is used in a **SELECT** statement to collect data across multiple records and group the results by one or more columns.

```
( CREATE TABLE workers  
id number(9), name  
varchar2(50), state  
varchar2(50), salary  
number(20),  
company varchar2(20)  
);
```

```
INSERT INTO workers VALUES(123456789, 'John Walker', 'Florida', 2500, 'IBM');  
INSERT INTO workers VALUES(234567890, 'Brad Pitt', 'Florida', 1500, 'APPLE');  
INSERT INTO workers VALUES(345678901, 'Eddie Murphy', 'Texas', 3000, 'IBM');  
INSERT INTO workers VALUES(456789012, 'Eddie Murphy', 'Virginia', 1000, 'GOOGLE');  
INSERT INTO workers VALUES(567890123, 'Eddie Murphy', 'Texas', 7000, 'MICROSOFT');  
INSERT INTO workers VALUES(456789012, 'Brad Pitt', 'Texas', 1500, 'GOOGLE');  
INSERT INTO workers VALUES(123456710, 'Mark Stone', 'Pennsylvania', 2500, 'IBM');
```

ID	NAME	STATE	SALARY	COMPANY
123456789	John Walker	Florida	2500	IBM
234567890	Brad Pitt	Florida	1500	APPLE
345678901	Eddie Murphy	Texas	3000	IBM
456789012	Eddie Murphy	Virginia	1000	GOOGLE
567890123	Eddie Murphy	Texas	7000	MICROSOFT
456789012	Brad Pitt	Texas	1500	GOOGLE
123456710	Mark Stone	Pennsylvania	2500	IBM

1) **Example:** Find the total salary for every employee

```
SELECT name, SUM(salary) AS "Total Salary"  
FROM workers  
GROUP BY name;
```

NAME	Total Salary
Brad Pitt	3000
Mark Stone	2500
John Walker	2500
Eddie Murphy	11000

2) **Example:** Find the number of employees per state

```
SELECT state, COUNT(state) AS "Number Of Workers"  
FROM workers  
GROUP BY state;
```

STATE	Number Of Employees
Virginia	1
Florida	2
Pennsylvania	1
Texas	3

CREATE TABLE workers

```
(  
    id number(9), name  
    varchar2(50), state  
    varchar2(50), salary  
    number(20),  
    company varchar2(20)  
);
```

```
INSERT INTO workers VALUES(123456789, 'John Walker', 'Florida', 2500, 'IBM');  
INSERT INTO workers VALUES(234567890, 'Brad Pitt', 'Florida', 1500, 'APPLE');  
INSERT INTO workers VALUES(345678901, 'Eddie Murphy', 'Texas', 3000, 'IBM');  
INSERT INTO workers VALUES(456789012, 'Eddie Murphy', 'Virginia', 1000, 'GOOGLE');  
INSERT INTO workers VALUES(567890123, 'Eddie Murphy', 'Texas', 7000, 'MICROSOFT');  
INSERT INTO workers VALUES(456789012, 'Brad Pitt', 'Texas', 1500, 'GOOGLE');  
INSERT INTO workers VALUES(123456710, 'Mark Stone', 'Pennsylvania', 2500, 'IBM');
```

ID	NAME	STATE	SALARY	COMPANY
123456789	John Walker	Florida	2500	IBM
234567890	Brad Pitt	Florida	1500	APPLE
345678901	Eddie Murphy	Texas	3000	IBM
456789012	Eddie Murphy	Virginia	1000	GOOGLE
567890123	Eddie Murphy	Texas	7000	MICROSOFT
456789012	Brad Pitt	Texas	1500	GOOGLE
123456710	Mark Stone	Pennsylvania	2500	IBM

3) Example: Find the number of the employees whose salary is more than \$2000 per company

```
SELECT company, COUNT(*) AS "Number Of Workers"  
FROM workers  
WHERE salary > 2000  
GROUP BY company;
```

COMPANY	Number Of Employees
MICROSOFT	1
IBM	3

4) Example: Find the minimum and maximum salary for every company

```
SELECT company, MIN(salary) AS "Min Salary", MAX(salary) AS "Max Salary"  
FROM workers  
GROUP BY company;
```

COMPANY	Min Salary	Max Salary
GOOGLE	1000	1500
MICROSOFT	7000	7000
APPLE	1500	1500
IBM	2500	3000

“HAVING” CLAUSE

HAVING clause is used in combination with the **GROUP BY** clause to restrict the groups of returned rows to only those whose the condition is TRUE. This is a further condition applied only to the aggregated results to restrict the groups of returned rows.

CREATE TABLE

```
workers
id number(9),
name varchar2(50),
state varchar2(50),
salary number(20),
company varchar2(20)
);
```

```
INSERT INTO workers VALUES(123456789, 'John Walker', 'Florida', 2500, 'IBM');
INSERT INTO workers VALUES(234567890, 'Brad Pitt', 'Florida', 1500, 'APPLE');
INSERT INTO workers VALUES(345678901, 'Eddie Murphy', 'Texas', 3000, 'IBM');
INSERT INTO workers VALUES(456789012, 'Eddie Murphy', 'Virginia', 1000, 'GOOGLE');
INSERT INTO workers VALUES(567890123, 'Eddie Murphy', 'Texas', 7000, 'MICROSOFT');
INSERT INTO workers VALUES(456789012, 'Brad Pitt', 'Texas', 1500, 'GOOGLE');
INSERT INTO workers VALUES(123456710, 'Mark Stone', 'Pennsylvania', 2500, 'IBM');
```

ID	NAME	STATE	SALARY	COMPANY
123456789	John Walker	Florida	2500	IBM
234567890	Brad Pitt	Florida	1500	APPLE
345678901	Eddie Murphy	Texas	3000	IBM
456789012	Eddie Murphy	Virginia	1000	GOOGLE
567890123	Eddie Murphy	Texas	7000	MICROSOFT
456789012	Brad Pitt	Texas	1500	GOOGLE
123456710	Mark Stone	Pennsylvania	2500	IBM

- 1) Example: Find the total salary if it is greater than 2500 for every employee

```
SELECT name, SUM(salary) AS "Total Salary"
FROM workers
GROUP BY name
HAVING SUM(salary) >= 2500;
```

NAME	Total Salary
Brad Pitt	3000
Mark Stone	2500
John Walker	2500
Eddie Murphy	11000

- 2) Example: Find the number of employees if it is more than 1 per state

```
SELECT state, COUNT(state) AS "Number Of Employees"  
FROM workers  
GROUP BY state  
HAVING COUNT(state) > 1;
```

STATE	Number Of Employees
Florida	2
Texas	3

- 3) Example: Find the minimum salary if it is more than 2000 for every company

```
SELECT company, MIN(salary) AS "Min Salary"  
FROM workers GROUP  
BY company  
HAVING MIN(salary) > 2000;
```

COMPANY	Min Salary
MICROSOFT	7000
IBM	2500

- 4) Example: Find the maximum salary if it is less than 3000 for every state

```
SELECT state, MAX(salary) AS "Max  
Salary" FROM workers  
GROUP BY state  
HAVING MAX(salary) < 3000;
```

STATE	Max Salary
Florida	2500
Pennsylvania	2500
Virginia	1000

“UNION” OPERATOR

UNION operator is used to combine the result sets of two or more **SELECT** statements.

It removes duplicate rows between the **SELECT** statements.

Each **SELECT** statement within the **UNION** operator must have the same number of fields in the result sets with similar data types.

```
CREATE TABLE employees
(
    id number(9),
    name varchar2(50),
    state varchar2(50),
    salary number(20),
    company varchar2(20)
);
```

```
INSERT INTO employees VALUES(123456789, 'John Walker', 'Florida', 2500, 'IBM');
INSERT INTO employees VALUES(234567890, 'Brad Pitt', 'Florida', 1500, 'APPLE');
INSERT INTO employees VALUES(345678901, 'Eddie Murphy', 'Texas', 3000, 'IBM');
INSERT INTO employees VALUES(456789012, 'Eddie Murphy', 'Virginia', 1000, 'GOOGLE'); →
INSERT INTO employees VALUES(567890123, 'Eddie Murphy', 'Texas', 7000, 'MICROSOFT');
INSERT INTO employees VALUES(456789012, 'Brad Pitt', 'Texas', 1500, 'GOOGLE');
INSERT INTO employees VALUES(123456710, 'Mark Stone', 'Pennsylvania', 2500, 'IBM');
```

ID	NAME	STATE	SALARY	COMPANY
123456789	John Walker	Florida	2500	IBM
234567890	Brad Pitt	Florida	1500	APPLE
345678901	Eddie Murphy	Texas	3000	IBM
456789012	Eddie Murphy	Virginia	1000	GOOGLE
567890123	Eddie Murphy	Texas	7000	MICROSOFT
456789012	Brad Pitt	Texas	1500	GOOGLE
123456710	Mark Stone	Pennsylvania	2500	IBM

1) Example: Find the state or employee names whose salary is greater than 3000, less than 2000 without duplication.

```
SELECT state AS "State or Employee Name", salary
FROM employees
WHERE salary >3000
UNION
SELECT name AS "State or Employee Name", salary
FROM employees
WHERE salary < 2000;
```

State or Employee Name	SALARY
Brad Pitt	1500
Eddie Murphy	1000
Texas	7000

Note: If you add **ORDER BY 2** after the last **WHERE** statement, you get the salary in ascending order.

“UNION ALL” OPERATOR

UNION operator is used to combine the result sets of two or more **SELECT** statements.

It returns all rows from the query and does not remove duplicate rows between the SELECT statements.

Each **SELECT** statement within the **UNION ALL** operator must have the **same number of fields** in the result sets **with similar data types.**

```
CREATE TABLE employees
(
    id number(9),
    name varchar2(50),
    state varchar2(50),
    salary number(20),
    company varchar2(20)
);
```

```
INSERT INTO employees VALUES(123456789, 'John Walker', 'Florida', 2500, 'IBM');
INSERT INTO employees VALUES(234567890, 'Brad Pitt', 'Florida', 1500, 'APPLE');
INSERT INTO employees VALUES(345678901, 'Eddie Murphy', 'Texas', 3000, 'IBM');
INSERT INTO employees VALUES(456789012, 'Eddie Murphy', 'Virginia', 1000, 'GOOGLE');
INSERT INTO employees VALUES(567890123, 'Eddie Murphy', 'Texas', 7000, 'MICROSOFT');
INSERT INTO employees VALUES(456789012, 'Brad Pitt', 'Texas', 1500, 'GOOGLE');
INSERT INTO employees VALUES(123456710, 'Mark Stone', 'Pennsylvania', 2500, 'IBM');
```

ID	NAME	STATE	SALARY	COMPANY
123456789	John Walker	Florida	2500	IBM
234567890	Brad Pitt	Florida	1500	APPLE
345678901	Eddie Murphy	Texas	3000	IBM
456789012	Eddie Murphy	Virginia	1000	GOOGLE
567890123	Eddie Murphy	Texas	7000	MICROSOFT
456789012	Brad Pitt	Texas	1500	GOOGLE
123456710	Mark Stone	Pennsylvania	2500	IBM

1) Example: Find all state or employee names whose salary is greater than 3000, less than 2000

```
SELECT state AS "State or Employee Name", salary
FROM employees
WHERE salary >3000
UNION ALL
SELECT name AS "State or Employee Name", salary
FROM employees
WHERE salary < 2000;
```

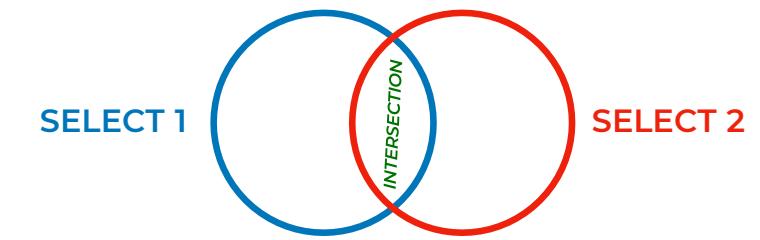
State or Employee Name	SALARY
Texas	7000
Brad Pitt	1500
Eddie Murphy	1000
Brad Pitt	1500

Note: When you use “**UNION**”, Brad Pitt is printed just once as you can see in the previous slide.

Note: If you add **ORDER BY 1** after the last **WHERE** statement, you get the salary in ascending order.

“INTERSECT” OPERATOR

INTERSECT operator is used to return the common results of 2 or more **SELECT** statements.



```
CREATE TABLE employees
(
    id number(9),
    name varchar2(50),
    state varchar2(50),
    salary number(20),
    company varchar2(20)
);
```

```
INSERT INTO employees VALUES(123456789, 'John Walker', 'Florida', 2500, 'IBM');
INSERT INTO employees VALUES(234567890, 'Brad Pitt', 'Florida', 1500, 'APPLE');
INSERT INTO employees VALUES(345678901, 'Eddie Murphy', 'Texas', 3000, 'IBM');
INSERT INTO employees VALUES(456789012, 'Eddie Murphy', 'Virginia', 1000, 'GOOGLE'); →
INSERT INTO employees VALUES(567890123, 'Eddie Murphy', 'Texas', 7000, 'MICROSOFT');
INSERT INTO employees VALUES(456789012, 'Brad Pitt', 'Texas', 1500, 'GOOGLE');
INSERT INTO employees VALUES(123456710, 'Mark Stone', 'Pennsylvania', 2500, 'IBM');
```

ID	NAME	STATE	SALARY	COMPANY
123456789	John Walker	Florida	2500	IBM
234567890	Brad Pitt	Florida	1500	APPLE
345678901	Eddie Murphy	Texas	3000	IBM
456789012	Eddie Murphy	Virginia	1000	GOOGLE
567890123	Eddie Murphy	Texas	7000	MICROSOFT
456789012	Brad Pitt	Texas	1500	GOOGLE
123456710	Mark Stone	Pennsylvania	2500	IBM

1) Example: Find all common employee names whose salary is greater than 1000, less than 2000

```
SELECT name FROM
employees
WHERE salary < 2000
```

NAME
Brad Pitt
Eddie Murphy
Brad Pitt

INTERSECT

```
SELECT name FROM
employees
WHERE salary > 1000;
```

NAME
John Walker
Brad Pitt
Eddie Murphy
Eddie Murphy
Brad Pitt
Mark Stone

NAME
Brad Pitt
Eddie Murphy

```
CREATE TABLE employees
(
    id number(9),
    name varchar2(50),
    state varchar2(50),
    salary number(20),
    company varchar2(20)
);
```

```
INSERT INTO employees VALUES(123456789, 'John Walker', 'Florida', 2500, 'IBM');
INSERT INTO employees VALUES(234567890, 'Brad Pitt', 'Florida', 1500, 'APPLE');
INSERT INTO employees VALUES(345678901, 'Eddie Murphy', 'Texas', 3000, 'IBM');
INSERT INTO employees VALUES(456789012, 'Eddie Murphy', 'Virginia', 1000, 'GOOGLE'); →
INSERT INTO employees VALUES(567890123, 'Eddie Murphy', 'Texas', 7000, 'MICROSOFT');
INSERT INTO employees VALUES(456789012, 'Brad Pitt', 'Texas', 1500, 'GOOGLE');
INSERT INTO employees VALUES(123456710, 'Mark Stone', 'Pennsylvania', 2500, 'IBM');
```

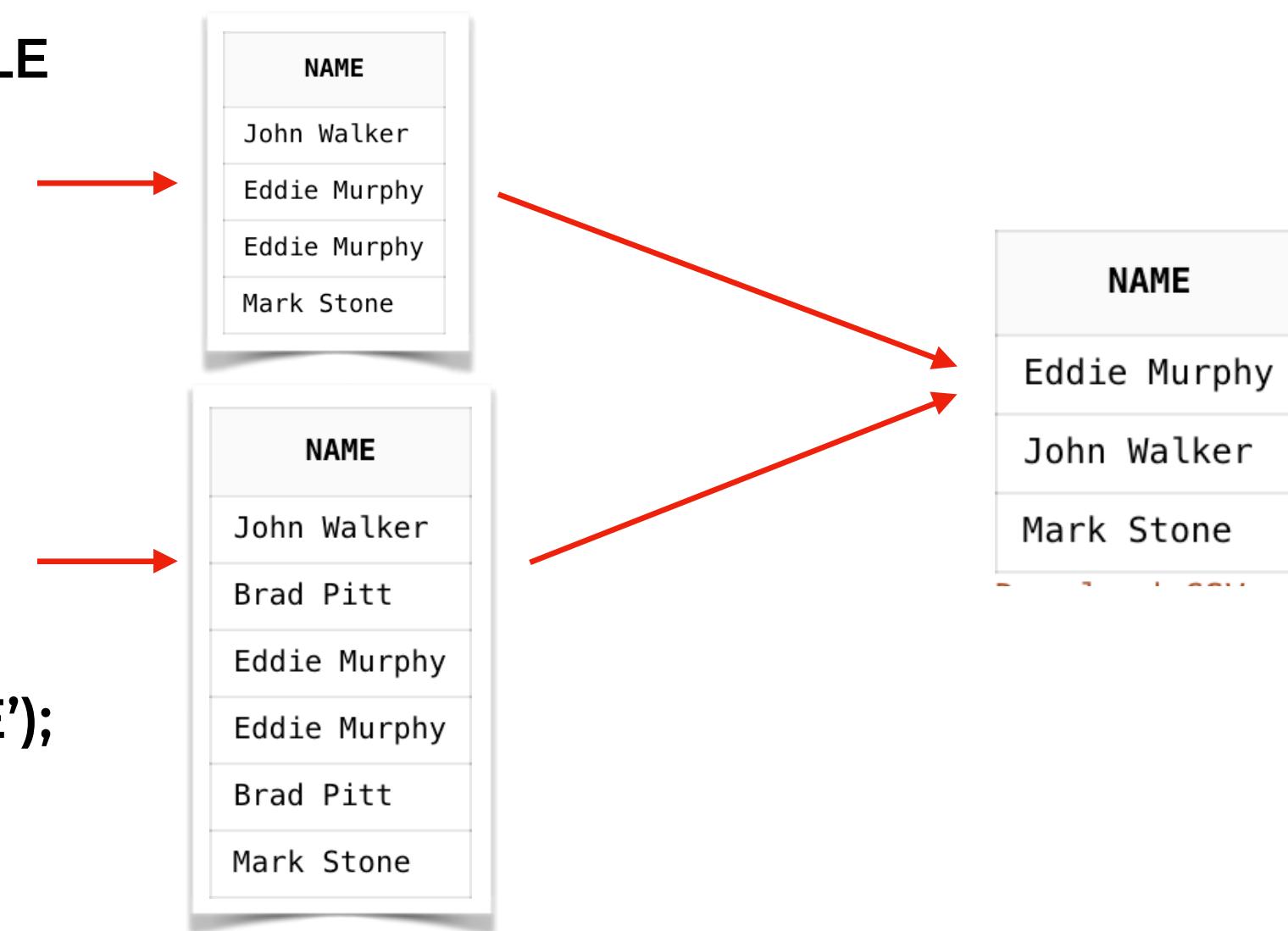
ID	NAME	STATE	SALARY	COMPANY
123456789	John Walker	Florida	2500	IBM
234567890	Brad Pitt	Florida	1500	APPLE
345678901	Eddie Murphy	Texas	3000	IBM
456789012	Eddie Murphy	Virginia	1000	GOOGLE
567890123	Eddie Murphy	Texas	7000	MICROSOFT
456789012	Brad Pitt	Texas	1500	GOOGLE
123456710	Mark Stone	Pennsylvania	2500	IBM

2) Example: Find all common employee names whose salary is greater than 2000 and company name is IBM, APPLE or GOOGLE

```
SELECT name
FROM employees
WHERE salary > 2000
```

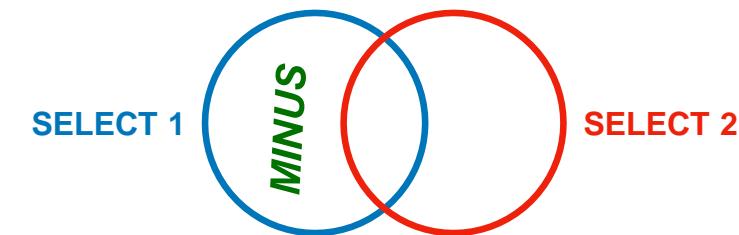
INTERSECT

```
SELECT name
FROM employees
WHERE company in ('IBM', 'APPLE', 'GOOGLE');
```



“MINUS” OPERATOR

MINUS operator is used to return all rows in the first SELECT statement that are not returned by the second SELECT statement.



```
CREATE TABLE employees
(
    id number(9),
    name varchar2(50),
    state varchar2(50),
    salary number(20),
    company varchar2(20)
);
```

```
INSERT INTO employees VALUES(123456789, 'John Walker', 'Florida', 2500, 'IBM');
INSERT INTO employees VALUES(234567890, 'Brad Pitt', 'Florida', 1500, 'APPLE');
INSERT INTO employees VALUES(345678901, 'Eddie Murphy', 'Texas', 3000, 'IBM');
INSERT INTO employees VALUES(456789012, 'Eddie Murphy', 'Virginia', 1000, 'GOOGLE');
INSERT INTO employees VALUES(567890123, 'Eddie Murphy', 'Texas', 7000, 'MICROSOFT');
INSERT INTO employees VALUES(456789012, 'Brad Pitt', 'Texas', 1500, 'GOOGLE');
INSERT INTO employees VALUES(123456710, 'Mark Stone', 'Pennsylvania', 2500, 'IBM');
```

ID	NAME	STATE	SALARY	COMPANY
123456789	John Walker	Florida	2500	IBM
234567890	Brad Pitt	Florida	1500	APPLE
345678901	Eddie Murphy	Texas	3000	IBM
456789012	Eddie Murphy	Virginia	1000	GOOGLE
567890123	Eddie Murphy	Texas	7000	MICROSOFT
456789012	Brad Pitt	Texas	1500	GOOGLE
123456710	Mark Stone	Pennsylvania	2500	IBM

2) Example: Find the employee names whose salary is less than 3000 and not working in GOOGLE

```
SELECT name, company
FROM employees
WHERE salary < 3000
```

NAME	COMPANY
Brad Pitt	APPLE
Eddie Murphy	GOOGLE
Brad Pitt	GOOGLE

MINUS

```
SELECT name, company
FROM employees
WHERE company IN ('GOOGLE');
```

NAME	COMPANY
Eddie Murphy	GOOGLE
Brad Pitt	GOOGLE

NAME	COMPANY
Brad Pitt	APPLE

REVIEW QUESTION

COMPANY_ID	COMPANY_NAME
100	IBM
101	GOOGLE
102	MICROSOFT
103	APPLE

Companies

ORDER_ID	COMPANY_ID	ORDER_DATE
11	101	17-APR-20
22	102	18-APR-20
33	103	19-APR-20
44	104	20-APR-20
55	105	21-APR-20

Orders

- 1) Create the given tables and insert data
- 2) Find the common company ids

NOTE: When you use SET OPERATIONS(Union, Union ALL, Intersect, Minus) both queries must return

1) Same number of columns. Following example gives error due to number of columns

```
SELECT name, state  
FROM employees1 WHERE salary  
< 2800
```

UNION

```
SELECT name, state, salary FROM  
employees1  
WHERE salary < 3000;
```

2) Corresponding columns must have same data types. Following example gives error due to data type

```
SELECT name, state  
FROM employees1 WHERE salary  
< 2800
```

INTERSECT

```
SELECT name, salary FROM  
employees1 WHERE salary <  
3000;
```

--Following example does not give error because number of columns and data types are fine.

--But SQL mixes the names and states, it is not good but no error SELECT
name, state
FROM employees1 WHERE salary
< 2800

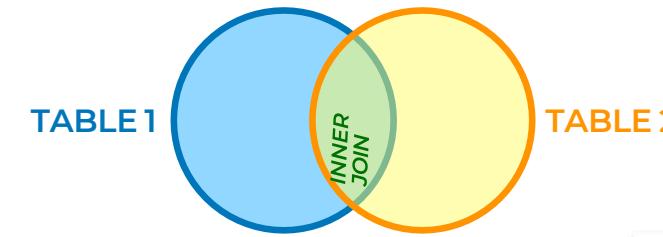
UNION

```
SELECT state, name FROM  
employees1 > 3000;
```

“JOINS”

1) INNER JOIN

The **INNER JOIN** would return the common records of two tables



```
CREATE TABLE my_companies
(
    company_id number(9),
    company_name
    varchar2(20)
);
```

```
INSERT INTO my_companies VALUES(100, 'IBM');
INSERT INTO my_companies VALUES(101, 'GOOGLE');
INSERT INTO my_companies VALUES(102, 'MICROSOFT');
INSERT INTO my_companies VALUES(103, 'APPLE');
```

COMPANY_ID	COMPANY_NAME
100	IBM
101	GOOGLE
102	MICROSOFT
103	APPLE

```
CREATE TABLE orders
(
    order_id number(9),
    company_id
    number(9),
    order_date date
);
```

```
INSERT INTO orders VALUES(11, 101, '17-Apr-2020');
INSERT INTO orders VALUES(22, 102, '18-Apr-2020');
INSERT INTO orders VALUES(33, 103, '19-Apr-2020');
INSERT INTO orders VALUES(44, 104, '20-Apr-2020');
INSERT INTO orders VALUES(55, 105, '21-Apr-2020');
```

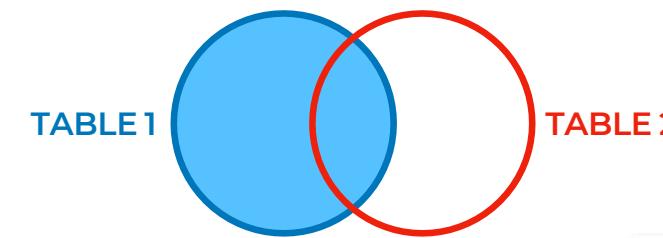
ORDER_ID	COMPANY_ID	ORDER_DATE
11	101	17-APR-20
22	102	18-APR-20
33	103	19-APR-20
44	104	20-APR-20
55	105	21-APR-20

```
SELECT my_companies.company_name, orders.order_id,
orders.order_date FROM my_companies INNER JOIN orders
ON my_companies.company_id = orders.company_id;
```

COMPANY_NAME	ORDER_ID	ORDER_DATE
GOOGLE	11	17-APR-20
MICROSOFT	22	18-APR-20
APPLE	33	19-APR-20

2) LEFT JOIN

The **LEFT JOIN** returns **all rows** from the **LEFT-Hand table**



```
CREATE TABLE companies
(
    company_id number(9), →
    company_name
    varchar2(20)
);
```

```
INSERT INTO companies VALUES(100, 'IBM');
INSERT INTO companies VALUES(101, 'GOOGLE');
INSERT INTO companies VALUES(102, 'MICROSOFT');
INSERT INTO companies VALUES(103, 'APPLE');
```

COMPANY_ID	COMPANY_NAME
100	IBM
101	GOOGLE
102	MICROSOFT
103	APPLE

```
CREATE TABLE orders
(
    order_id number(9), →
    company_id
    number(9),
    order_date date
);
```

```
INSERT INTO orders VALUES(11, 101, '17-Apr-2020');
INSERT INTO orders VALUES(22, 102, '18-Apr-2020');
INSERT INTO orders VALUES(33, 103, '19-Apr-2020');
INSERT INTO orders VALUES(44, 104, '20-Apr-2020');
INSERT INTO orders VALUES(55, 105, '21-Apr-2020');
```

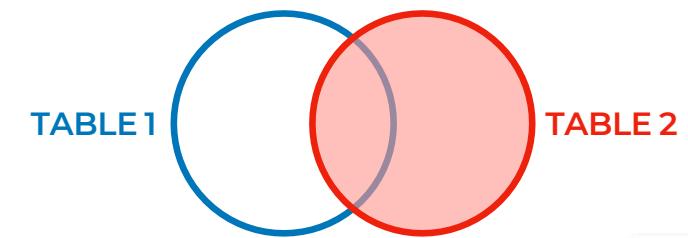
ORDER_ID	COMPANY_ID	ORDER_DATE
11	101	17-APR-20
22	102	18-APR-20
33	103	19-APR-20
44	104	20-APR-20
55	105	21-APR-20

```
SELECT companies.company_name, orders.order_id,
orders.order_date FROM companies LEFT JOIN orders
ON companies.company_id = orders.company_id;
```

COMPANY_NAME	ORDER_ID	ORDER_DATE
GOOGLE	11	17-APR-20
MICROSOFT	22	18-APR-20
APPLE	33	19-APR-20
IBM	-	-

3) RIGHT JOIN

The **RIGHT JOIN** returns **all rows** from the **RIGHT-Hand table**



```
CREATE TABLE companies
(
    company_id number(9),
    company_name
    varchar2(20)
);
```

```
INSERT INTO companies VALUES(100, 'IBM');
INSERT INTO companies VALUES(101, 'GOOGLE');
INSERT INTO companies VALUES(102, 'MICROSOFT');
INSERT INTO companies VALUES(103, 'APPLE');
```

COMPANY_ID	COMPANY_NAME
100	IBM
101	GOOGLE
102	MICROSOFT
103	APPLE

```
CREATE TABLE
orders
order_id number(9),
company_id
number(9),
order_date date
);
```

```
INSERT INTO orders VALUES(11, 101, '17-Apr-2020');
INSERT INTO orders VALUES(22, 102, '18-Apr-2020');
INSERT INTO orders VALUES(33, 103, '19-Apr-2020');
INSERT INTO orders VALUES(44, 104, '20-Apr-2020');
INSERT INTO orders VALUES(55, 105, '21-Apr-2020');
```

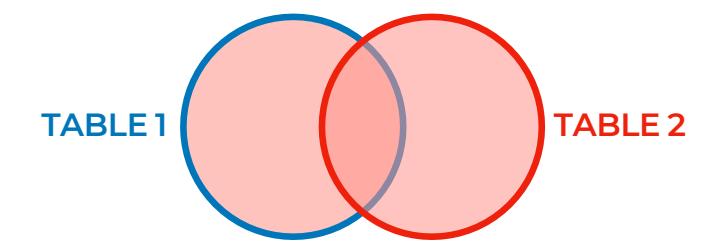
ORDER_ID	COMPANY_ID	ORDER_DATE
11	101	17-APR-20
22	102	18-APR-20
33	103	19-APR-20
44	104	20-APR-20
55	105	21-APR-20

```
SELECT companies.company_name, orders.order_id,
orders.order_date FROM companies RIGHT JOIN orders
ON companies.company_id = orders.company_id;
```

COMPANY_NAME	ORDER_ID	ORDER_DATE
GOOGLE	11	17-APR-20
MICROSOFT	22	18-APR-20
APPLE	33	19-APR-20
-	55	21-APR-20
-	44	20-APR-20

4) FULL JOIN

The **FULL JOIN** returns **all rows** from the LEFT-Hand table and RIGHT-Hand table with nulls in place where the join condition is not met.



```
CREATE TABLE companies
(
    company_id number(9),
    company_name
    varchar2(20)
);
```

```
INSERT INTO companies VALUES(100, 'IBM');
INSERT INTO companies VALUES(101, 'GOOGLE');
INSERT INTO companies VALUES(102, 'MICROSOFT');
INSERT INTO companies VALUES(103, 'APPLE');
```

COMPANY_ID	COMPANY_NAME
100	IBM
101	GOOGLE
102	MICROSOFT
103	APPLE

```
CREATE TABLE orders
(
    order_id number(9),
    company_id
    number(9),
    order_date date
);
```

```
INSERT INTO orders VALUES(11, 101, '17-Apr-2020');
INSERT INTO orders VALUES(22, 102, '18-Apr-2020');
INSERT INTO orders VALUES(33, 103, '19-Apr-2020');
INSERT INTO orders VALUES(44, 104, '20-Apr-2020');
INSERT INTO orders VALUES(55, 105, '21-Apr-2020');
```

ORDER_ID	COMPANY_ID	ORDER_DATE
11	101	17-APR-20
22	102	18-APR-20
33	103	19-APR-20
44	104	20-APR-20
55	105	21-APR-20

```
SELECT companies.company_name, orders.order_id,
orders.order_date FROM companies FULL JOIN orders
ON companies.company_id = orders.company_id;
```

COMPANY_NAME	ORDER_ID	ORDER_DATE
GOOGLE	11	17-APR-20
MICROSOFT	22	18-APR-20
APPLE	33	19-APR-20
-	44	20-APR-20
-	55	21-APR-20
IBM	-	-

5) SELF JOIN

```
CREATE TABLE workers
(
    id number(2),
    name varchar2(20),
    title varchar2(60),
    manager_id number(2)
);
```

→
INSERT INTO workers VALUES(1, 'Ali Can', 'SDET', 2);
INSERT INTO workers VALUES(2, 'John Walker', 'QA', 3);
INSERT INTO workers VALUES(3, 'Angie Star', 'QA Lead', 4);
INSERT INTO workers VALUES(4, 'Amy Sky', 'CEO', 5);

ID	NAME	TITLE	BOSS_ID
1	Ali Can	SDET	2
2	John Walker	QA	3
3	Angie Star	QA Lead	4
4	Amy Sky	CEO	5

```
SELECT e1.name AS employee_name, e2.name AS
manager_name FROM my_employees
JOIN my_employees e2
ON e1.manager_id = e2.id;
```

e1 INNER
→

EMPLOYEE_NAME	BOSS_NAME
Ali Can	John Walker
John Walker	Angie Star
Angie Star	Amy Sky

“PIVOT” CLAUSE

PIVOT Clause allows you to aggregate your results and rotate rows into columns

```
CREATE TABLE customers_products
(
    product_id number(10),
    customer_name
    varchar2(50),
    product_name
    varchar2(50)
);
```



```
INSERT INTO customers_products VALUES (10, 'Mark', 'Orange');
INSERT INTO customers_products VALUES (10, 'Mark', 'Orange');
INSERT INTO customers_products VALUES (20, 'John', 'Apple');
INSERT INTO customers_products VALUES (30, 'Amy', 'Palm');
INSERT INTO customers_products VALUES (20, 'Mark', 'Apple');
INSERT INTO customers_products VALUES (10, 'Adem', 'Orange');
INSERT INTO customers_products VALUES (40, 'John', 'Apricot');
INSERT INTO customers_products VALUES (20, 'Eddie', 'Apple');
```

PRODUCT_ID	CUSTOMER_NAME	PRODUCT_NAME
10	Mark	Orange
10	Mark	Orange
20	John	Apple
30	Amy	Palm
20	Mark	Apple
10	Adem	Orange
40	John	Apricot
20	Eddie	Apple

```
SELECT * FROM (SELECT product_name, customer_name FROM
customers_products) PIVOT
(COUNT(product_name) FOR product_name IN ('Orange','Apple','Apricot','Palm'));
```

CUSTOMER_NAME	'Orange'	'Apple'	'Apricot'	'Palm'
Amy	0	0	0	1
Mark	2	1	0	0
Adem	1	0	0	0
Eddie	0	1	0	0
John	0	1	1	0

“ALTER TABLE” STATEMENT

ALTER TABLE statement is used to **add, modify, or drop/delete columns** in a table. **ALTER TABLE** statement is also used to **rename a table**.

```
CREATE TABLE employees  
id number(9),  
name varchar2(50),  
state varchar2(50),  
salary number(20),  
company varchar2(20)  
);
```

```
INSERT INTO employees VALUES(123456789, 'John Walker', 'Florida', 2500, 'IBM');  
INSERT INTO employees VALUES(234567890, 'Brad Pitt', 'Florida', 1500, 'APPLE');  
INSERT INTO employees VALUES(345678901, 'Eddie Murphy', 'Texas', 3000, 'IBM');  
INSERT INTO employees VALUES(456789012, 'Eddie Murphy', 'Virginia', 1000, 'GOOGLE');  
INSERT INTO employees VALUES(567890123, 'Eddie Murphy', 'Texas', 7000, 'MICROSOFT');  
INSERT INTO employees VALUES(456789012, 'Brad Pitt', 'Texas', 1500, 'GOOGLE');  
INSERT INTO employees VALUES(123456710, 'Mark Stone', 'Pennsylvania', 2500, 'IBM');
```

ID	NAME	STATE	SALARY	COMPANY
123456789	John Walker	Florida	2500	IBM
234567890	Brad Pitt	Florida	1500	APPLE
345678901	Eddie Murphy	Texas	3000	IBM
456789012	Eddie Murphy	Virginia	1000	GOOGLE
567890123	Eddie Murphy	Texas	7000	MICROSOFT
456789012	Brad Pitt	Texas	1500	GOOGLE
123456710	Mark Stone	Pennsylvania	2500	IBM

1) ADD a column into a table with a default value

```
ALTER TABLE employees  
ADD country_name varchar2(20) DEFAULT 'USA';
```

ID	NAME	STATE	SALARY	COMPANY	COUNTRY_NAME
123456789	John Walker	Florida	2500	IBM	USA
234567890	Brad Pitt	Florida	1500	APPLE	USA
345678901	Eddie Murphy	Texas	3000	IBM	USA
456789012	Eddie Murphy	Virginia	1000	GOOGLE	USA
567890123	Eddie Murphy	Texas	7000	MICROSOFT	USA
456789012	Brad Pitt	Texas	1500	GOOGLE	USA
123456710	Mark Stone	Pennsylvania	2500	IBM	USA

2) ADD multiple columns into a table

```
ALTER TABLE  
employees ADD  
(gender varchar2(6),  
age number(3))
```

ID	NAME	STATE	SALARY	COMPANY	COUNTRY_NAME	GENDER	AGE
123456789	John Walker	Florida	2500	IBM	USA	-	-
234567890	Brad Pitt	Florida	1500	APPLE	USA	-	-
345678901	Eddie Murphy	Texas	3000	IBM	USA	-	-
456789012	Eddie Murphy	Virginia	1000	GOOGLE	USA	-	-
567890123	Eddie Murphy	Texas	7000	MICROSOFT	USA	-	-
456789012	Brad Pitt	Texas	1500	GOOGLE	USA	-	-
123456710	Mark Stone	Pennsylvania	2500	IBM	USA	-	-

3)DROP COLUMN in a table

ID	NAME	STATE	SALARY	COMPANY	COUNTRY_NAME	GENDER	AGE
123456789	John Walker	Florida	2500	IBM	USA	-	-
234567890	Brad Pitt	Florida	1500	APPLE	USA	-	-
345678901	Eddie Murphy	Texas	3000	IBM	USA	-	-
456789012	Eddie Murphy	Virginia	1000	GOOGLE	USA	-	-
567890123	Eddie Murphy	Texas	7000	MICROSOFT	USA	-	-
456789012	Brad Pitt	Texas	1500	GOOGLE	USA	-	-
123456710	Mark Stone	Pennsylvania	2500	IBM	USA	-	-

ALTER TABLE employees DROP COLUMN age;

ID	NAME	STATE	SALARY	COMPANY	COUNTRY_NAME	GENDER
123456789	John Walker	Florida	2500	IBM	USA	-
234567890	Brad Pitt	Florida	1500	APPLE	USA	-
345678901	Eddie Murphy	Texas	3000	IBM	USA	-
456789012	Eddie Murphy	Virginia	1000	GOOGLE	USA	-
567890123	Eddie Murphy	Texas	7000	MICROSOFT	USA	-
456789012	Brad Pitt	Texas	1500	GOOGLE	USA	-
123456710	Mark Stone	Pennsylvania	2500	IBM	USA	-

4)RENAME COLUMN in a table

ID	NAME	STATE	SALARY	COMPANY	COUNTRY_NAME	GENDER
123456789	John Walker	Florida	2500	IBM	USA	-
234567890	Brad Pitt	Florida	1500	APPLE	USA	-
345678901	Eddie Murphy	Texas	3000	IBM	USA	-
456789012	Eddie Murphy	Virginia	1000	GOOGLE	USA	-
567890123	Eddie Murphy	Texas	7000	MICROSOFT	USA	-
456789012	Brad Pitt	Texas	1500	GOOGLE	USA	-
123456710	Mark Stone	Pennsylvania	2500	IBM	USA	-

ALTER TABLE employees RENAME COLUMN company TO company_name;

ID	NAME	STATE	SALARY	COMPANY_NAME	COUNTRY_NAME	GENDER
123456789	John Walker	Florida	2500	IBM	USA	-
234567890	Brad Pitt	Florida	1500	APPLE	USA	-
345678901	Eddie Murphy	Texas	3000	IBM	USA	-
456789012	Eddie Murphy	Virginia	1000	GOOGLE	USA	-
567890123	Eddie Murphy	Texas	7000	MICROSOFT	USA	-
456789012	Brad Pitt	Texas	1500	GOOGLE	USA	-
123456710	Mark Stone	Pennsylvania	2500	IBM	USA	-

5) RENAME table name in a table

```
ALTER TABLE  
employees RENAME  
TO workers;
```

```
SELECT * FROM  
workers;
```

ID	NAME	STATE	SALARY	COMPANY_NAME	COUNTRY_NAME	GENDER
123456789	John Walker	Florida	2500	IBM	USA	-
234567890	Brad Pitt	Florida	1500	APPLE	USA	-
345678901	Eddie Murphy	Texas	3000	IBM	USA	-
456789012	Eddie Murphy	Virginia	1000	GOOGLE	USA	-
567890123	Eddie Murphy	Texas	7000	MICROSOFT	USA	-
456789012	Brad Pitt	Texas	1500	GOOGLE	USA	-
123456710	Mark Stone	Pennsylvania	2500	IBM	USA	-

Schema

Search Database Obj

Schema: My Schema

Sort By: Name

WORKERS

Table Status: Valid Created 23 minutes ago

6) MODIFY column or columns in a table

Columns								
#	Column	Type	Length	Precision	Scale	Nullable	Semantics	
1	ID	NUMBER	22	9	0	Yes		
2	NAME	VARCHAR2	50			Yes	Byte	
3	STATE	VARCHAR2	50			Yes	Byte	
4	SALARY	NUMBER	22	20	0	Yes		
5	COMPANY_NAME	VARCHAR2	20			Yes	Byte	
6	COUNTRY_NAME	VARCHAR2	20			Yes	Byte	
7	GENDER	VARCHAR2	11			Yes	Byte	



**ALTER TABLE workers
MODIFY state varchar2(70) NOT
NULL;**



Columns								
#	Column	Type	Length	Precision	Scale	Nullable	Semantics	
1	ID	NUMBER	22	9	0	Yes		
2	NAME	VARCHAR2	50			Yes	Byte	
3	STATE	VARCHAR2	70			No	Byte	
4	SALARY	NUMBER	22	20	0	Yes		
5	COMPANY_NAME	VARCHAR2	20			Yes	Byte	
6	COUNTRY_NAME	VARCHAR2	20			Yes	Byte	
7	GENDER	VARCHAR2	11			Yes	Byte	

To modify multiple columns



**ALTER TABLE workers
MODIFY (state varchar2(70) NOT
NULL, id number(11)
NULL);**

EXCEPTIONS IN ORACLE SQL

1) CREATE TABLE students
(
 id char(11),
 name VARCHAR2(50),
 grade number(3),
 address
 VARCHAR2(80),
 update_date date
);

4) DECLARE
 employee_id char(11);
 employee_name
 VARCHAR2(50);
BEGIN
 SELECT id, name
 INTO employee_id,
 employee_name FROM students;
END;
/

2) INSERT INTO students VALUES(123456789, 'John Walker', 11, '1234 Texas', '14-Apr-2020');
 INSERT INTO students VALUES(223456789, 'Johnny Walker', 12, '1234 Florida', '14-Apr-2020');

3) SELECT * FROM

ID	NAME	GRADE	ADDRESS	UPDATE_DATE
1 123456789	John Walker	11	1234 Texas	14-APR-20
2 223456789	Johnny Walker	12	1234 Florida	14-APR-20

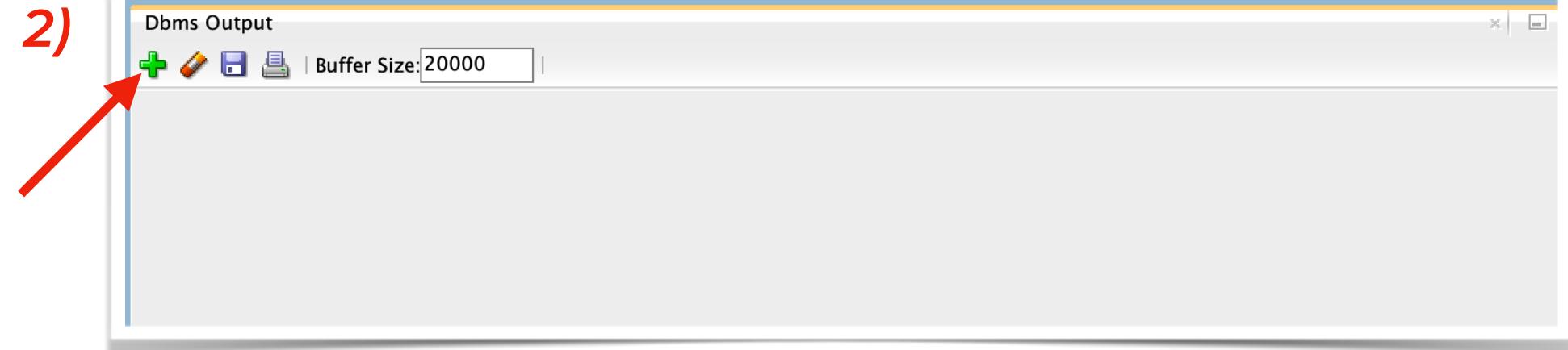
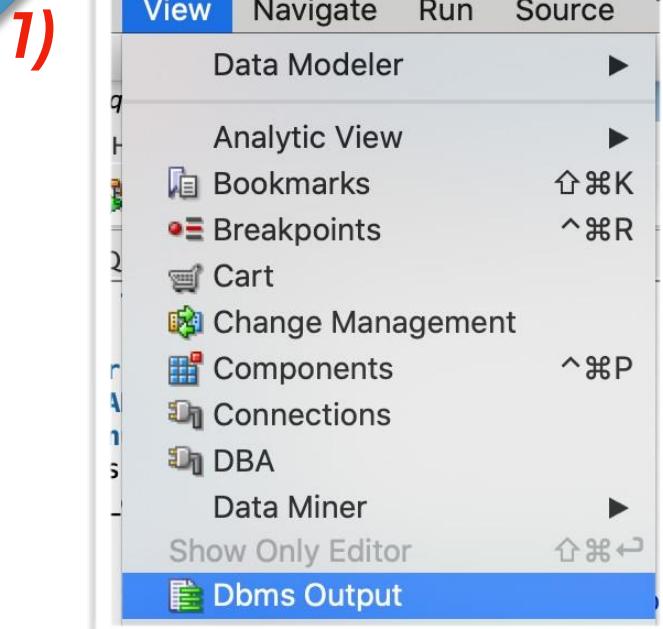
5) Error starting at line : 15 in command -
DECLARE
 employee_id char(11);
 employee_name VARCHAR2(50);
BEGIN
 SELECT id, name
 INTO employee_id, employee_name
 FROM students;
END;
Error report -
ORA-01422: exact fetch returns more than requested number of rows
ORA-06512: at line 5
01422. 00000 - "exact fetch returns more than requested number of rows"
*Cause: The number specified in exact fetch is less than the rows returned.
*Action: Rewrite the query or change number of rows requested

Note: If we run the 4th code we get the 5th Error Message on the console.

Because, **employee_id** and **employee_name** can store just a single value but more than one data is coming from table. SQL cannot store multiple data into a single variable so it throws Exception.

How can we handle Exception in SQL?

HOW TO TURN ON DBMS_OUTPUT IN ORACLE SQL



Using Predefined Exception to Handle Exceptions

1) TOO_MANY_ROWS Example:

```
DECLARE
  std_id char(11);
  std_name
  VARCHAR2(50); BEGIN
  SELECT id, name
  INTO std_id,
  std_name FROM
  students;
  DBMS_OUTPUT.PUT_LINE('Student id: '||std_id);
  DBMS_OUTPUT.PUT_LINE('Student name: '||std_name);
  DBMS_OUTPUT.PUT_LINE('Query is completed
successfully!'); DBMS_OUTPUT.PUT_LINE(std_id || ' ' ||
  std_name);
EXCEPTION
  WHEN TOO_MANY_ROWS THEN
    DBMS_OUTPUT.PUT_LINE('Too many
rows!!!'); END;
```

How to Handle Exceptions in Oracle SQL

2) NO_DATA_FOUND Example:

```
DECLARE
  std_id char(11);
  std_name
  varchar2(50);
BEGIN
  SELECT id, name
  INTO std_id,
  std_name FROM
  students WHERE id =
  '13579';
  DBMS_OUTPUT.PUT_LINE('Employee id: ' || std_id);
  DBMS_OUTPUT.PUT_LINE('Employee name: ' ||
  std_name);
EXCEPTION
  WHEN NO_DATA_FOUND THEN
    DBMS_OUTPUT.PUT_LINE('Query did not return any
```

3) ZERO_DIVIDE Example:

```
DECLARE
    result number(4, 2);
    num1 number(4) :=
    12; num2 number(4)
    := 0;
BEGIN
    result := num1 / num2;
    DBMS_OUTPUT.PUT_LINE(num1||'/'||num2||='||resu
    lt);
EXCEPTION
    WHEN ZERO_DIVIDE THEN
        DBMS_OUTPUT.PUT_LINE('Do not divide by
zero'); END;
```

4) INVALID_NUMBER Example:

```
DECLARE
    new_grade
    number(3); BEGIN
        INSERT INTO students(id, grade) VALUES('523456789', 'A');
        SELECT grade
        INTO
        new_grade
        FROM students
        WHERE id = '323456789';
        DBMS_OUTPUT.PUT_LINE('Grade is
'||new_grade); EXCEPTION
    WHEN INVALID_NUMBER THEN
        DBMS_OUTPUT.PUT_LINE('Conversion of String to number is
failed!'); END;
```

PRE-DEFINED EXCEPTIONS IN PL/SQL

Exception	Oracle Error	SQLCODE	Description
ACCESS_INTO_NULL	06530	-6530	It is raised when a null object is automatically assigned a value.
CASE_NOT_FOUND	06592	-6592	It is raised when none of the choices in the WHEN clause of a CASE statement is selected, and there is no ELSE clause.
COLLECTION_IS_NULL	06531	-6531	It is raised when a program attempts to apply collection methods other than EXISTS to an uninitialized nested table or varray, or the program attempts to assign values to the elements of an uninitialized nested table or varray.
DUP_VAL_ON_INDEX	00001	-1	It is raised when duplicate values are attempted to be stored in a column with unique index.
INVALID_CURSOR	01001	-1001	It is raised when attempts are made to make a cursor operation that is not allowed, such as closing an unopened cursor.
INVALID_NUMBER	01722	-1722	It is raised when the conversion of a character string into a number fails because the string does not represent a valid number.
LOGIN_DENIED	01017	-1017	It is raised when a program attempts to log on to the database with an invalid username or password.
NO_DATA_FOUND	01403	+100	It is raised when a SELECT INTO statement returns no rows.
NOT_LOGGED_ON	01012	-1012	It is raised when a database call is issued without being connected to the database.
PROGRAM_ERROR	06501	-6501	It is raised when PL/SQL has an internal problem.
ROWTYPE_MISMATCH	06504	-6504	It is raised when a cursor fetches value in a variable having incompatible data type.
SELF_IS_NULL	30625	-30625	It is raised when a member method is invoked, but the instance of the object type was not initialized.
STORAGE_ERROR	06500	-6500	It is raised when PL/SQL ran out of memory or memory was corrupted.
TOO_MANY_ROWS	01422	-1422	It is raised when a SELECT INTO statement returns more than one row.
VALUE_ERROR	06502	-6502	It is raised when an arithmetic, conversion, truncation, or sizeconstraint error occurs.
ZERO_DIVIDE	01476	1476	It is raised when an attempt is made to divide a number by zero.

USING USER-DEFINED EXCEPTION TO HANDLE EXCEPTIONS

1) Example:

```
DECLARE
    employee_id NUMBER;
    wrong_entry
    EXCEPTION;
BEGIN
    employee_id := '&Three_Digits_Number';
    IF(employee_id>999 or employee_id<100)
    THEN
        RAISE
    wrong_entry; ELSE
        DBMS_OUTPUT.PUT_LINE(employee_i
d); END IF;
EXCEPTION
    WHEN wrong_entry THEN
        DBMS_OUTPUT.put_line('Employee id must have 3
digits!!!'); END;
```

Note: This user-defined exception is throwing exception if the employee_id has no 3 digits.

USING USER-DEFINED EXCEPTION AND PRE-DEFINED EXCEPTION TO HANDLE EXCEPTIONS

2) Example: How to use multiple Exceptions...

DECLARE

```
new_grade students.grade%TYPE;  
new_name students.name%TYPE;
```

```
invalid_grade EXCEPTION; BEGIN
```

```
new_grade := '&std_grade';  
new_name := '&std_name';
```

```
IF(new_grade <=0 or new_grade >=12) THEN RAISE  
    invalid_grade;
```

```
ELSE
```

```
DBMS_OUTPUT.put_line(new_grade); END IF;
```

```
EXCEPTION
```

```
WHEN invalid_grade THEN DBMS_OUTPUT.put_line('Employee id  
must be from 0 to 12');
```

```
WHEN value_error THEN
```

```
DBMS_OUTPUT.put_line('Do not use characters different from 0 to 4'); END;
```

3) Example: How to use multiple Exceptions...

```
DECLARE
    std_id char(11);
    std_name
    varchar2(50); result
    number(4,2);
    num1 number(4,2) :=
    '&num1'; num2 number(4,2)
    := '&num2';
BEGIN
    SELECT id, name
    INTO std_id,
    std_name FROM
    students WHERE
    grade = 21;
    DBMS_OUTPUT.put_line('ID: ' || std_id);
    DBMS_OUTPUT.put_line('Name: ' ||
    std_name); result := num1 / num2;
    DBMS_OUTPUT.put_line(num1 ||'/'|| num2
    ||'=||result); EXCEPTION
    WHEN NO_DATA_FOUND THEN
        DBMS_OUTPUT.put_line('Query did not return any
        data'); WHEN ZERO_DIVIDE THEN
        DBMS_OUTPUT.put_line('Divisor cannot be
        zero!'); WHEN OTHERS THEN
        DBMS_OUTPUT.put_line('Undefined error!');
END;
```

USER DEFINED FUNCTIONS(PL/SQL BLOCK) IN ORACLE SQL

Note: Functions are created for 2 main reasons

- 1) Calculations
- 2) To fetch data from database

Advantage: Reusability

Functions : Functions must return a value; mainly used to compute and return a value.

Functions can have just input parameters

We can use functions in SQL statements like SELECT, INSERT, UPDATE, DELETE etc. We cannot call Procedures from inside the Functions

DUAL is a 1 row 1 column table automatically created by Oracle Database along with the data dictionary. DUAL is in the schema of the user SYS but is accessible by the name DUAL to all users.

It has one column whose name is DUMMY, defined to be VARCHAR2(1), and contains one row with a value X. Selecting from the DUAL table is useful for computing a constant expression with the SELECT statement because DUAL has only one row, the constant is returned only once.

Usage Examples of DUAL Table:

- 1) SELECT 3+5 FROM DUAL;
- 2) SELECT sysdate FROM DUAL;
- 3) SELECT user FROM DUAL;

REALIFE EXAMPLE OF DUAL TABLE USAGE

```
CREATE TABLE students (
    name VARCHAR2(50),
    avg_score NUMBER(4,2),
    modify_date DATE, modifier
    VARCHAR2(50)
);
```

```
DECLARE
cur_date DATE;
modifer_name VARCHAR2(50);
```

```
BEGIN
    SELECT sysdate, user
    INTO cur_date, modifer_name
    FROM DUAL;
```

```
INSERT INTO students VALUES('John Walker', 87.23, cur_date, modifer_name);
INSERT INTO students VALUES('Mary Star', 95.12, cur_date, modifer_name);
INSERT INTO students VALUES('Angie Ocean', 68.54, cur_date,
                           modifer_name);
```

```
END;
```

```
SELECT * FROM students;
```



NAME	AVG_SCORE	MODIFY_DATE	MODIFIER
1 John Walker	87.23	16-DEC-20	HR
2 Mary Star	95.12	16-DEC-20	HR
3 Angie Ocean	68.54	16-DEC-20	HR

Example 1:

1) Syntax to Create a Function

```
CREATE OR REPLACE FUNCTION addf(a number, b  
number) RETURN NUMBER IS  
BEGIN  
RETURN  
a+b; END;
```

2) Syntax to Call the Function

1. Way: Syntax to Call the

```
Function SELECT addf(10, 20)  
FROM DUAL;
```

2. Way: Syntax to Call the Function

```
EXECUTE DBMS_OUTPUT.PUT_LINE(addf(10, 20));
```

3. Way: Syntax to Call the

```
Function VARIABLE K NUMBER;  
EXECUTE :K := addf(10,  
20); PRINT K;
```

Example 2:

1) Syntax to Create a Function

```
CREATE OR REPLACE FUNCTION calcf(a number, b number, opr char) RETURN NUMBER
IS
BEGIN
  IF opr = '+' THEN RETURN a+b;
  ELSIF opr = '-' THEN RETURN a-b;
  ELSIF opr = '*' THEN RETURN a*b;
  ELSIF opr = '/' THEN RETURN a/b;
  ELSE
    DBMS_OUTPUT.PUT_LINE('Select one of the +, -, *, /');
  END IF;
END;
```

```
RETURN      ==> Functions must return value, we must put that
0; END IF; over here
EXCEPTION
WHEN ZERO_DIVIDE THEN
DBMS_OUTPUT.PUT_LINE('Do not divide by
zero!!!');

RETURN 0; ==> Functions must return value, we must put that over here
END;
```

2) Syntax to Call the Function

1. Way: Syntax to Call the Function

SELECT calcf(30, 20, 'm') FROM DUAL; ==> This gives error sometimes

2. Way: Syntax to Call the Function

EXECUTE DBMS_OUTPUT.PUT_LINE(calcfc(10, 20, 'm'));

3. Way: Syntax to Call the Function

VARIABLE K NUMBER;
EXECUTE :K := calcf(400, 20, '+'); PRINT K;

Example 3:

1) Syntax to Create and call a Function with DECLARE part

```
DECLARE  
  a number := '&numA';  
  b number := '&numB';
```

```
FUNCTION findMax(x number, y  
number) RETURN NUMBER IS
```

```
BEGIN  
  IF x > y  
  THEN  
    RETURN  
    x;  
  ELSE  
    RETURN y;  
  END IF;  
END;
```

- - The following part is to call function
- - When you use **DECLARE** in the method use the following way to call method
- - When you use **DECLARE** do not type “CREATE OR REPLACE FUNCTION” type just “FUNCTION”

```
BEGIN  
  dbms_output.put_line(' Maximum value of '||a||' and '||b||' is ' ||  
                      findMax(a,  
b)); END;
```

Example 4:

1) Syntax to Create and call a Function with DECLARE part

```
DECLARE
    num number := '&num';

FUNCTION fact(x
number) RETURN
NUMBER IS

BEGIN
    IF x=0 THEN
        RETURN 1;
    ELSE
        RETURN x * fact(x-
1); END IF;
END;

BEGIN
    dbms_output.put_line(num|| '! = ' ||
fact(num)); END;
```

HOW TO DROP A FUNCTION FROM DATABASE

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

DROP FUNCTION function_name;

Example: Let us drop **DivNumber** function **DROP FUNCTION** **DivNumber;**

USER DEFINED PROCEDURES(PL/SQL BLOCK) IN ORACLE SQL

Procedures : Procedures **may or may not return a value**; mainly used to perform an action.

Procedures can have **input, output, and input/output parameters**

We **cannot use** procedures **in SQL statements** like SELECT, INSERT, UPDATE, DELETE, MERGE etc. We can call functions from inside the procedures

Parameters are in 3 types in PL/SQL

IN: An IN parameter lets you pass a value to the subprogram.

It is a read-only parameter.

Inside the subprogram, an IN parameter acts like a constant. It cannot be assigned a value.

OUT: An OUT parameter returns a value to the calling program.

Inside the subprogram, an OUT parameter acts like a variable.

You can change its value and reference the value after assigning it.

IN OUT: An IN OUT parameter passes an initial value to a subprogram and returns an updated value to the caller.

It can be assigned a value and the value can be read.

Example 1:

—How to create a procedure

```
CREATE OR REPLACE PROCEDURE findMin(x IN number, y IN number, z OUT  
number) IS BEGIN  
    IF x < y THEN  
        z:= x;  
    ELSE  
        z:= y;  
    END  
    IF;  
END;
```

—How to call a procedure

```
DECLARE
```

```
a number;  
b number;  
c  
number;  
BEGIN  
    a:= 23;  
    b:= 45;  
    findMin(a, b, c);  
    dbms_output.put_line(' Minimum is ' ||  
c); END;
```

Example 2:

```
DECLARE  
  a number;
```

```
PROCEDURE squareNum(x IN OUT  
  number) IS BEGIN  
  x := x*x;  
END;
```

```
BEGIN  
  a:= 23;  
  squareNum(a);  
  dbms_output.put_line('Square is ' ||  
    a);  
END;  
/
```

Example 3:

```
CREATE TABLE
workers
  id CHAR(5),
  name VARCHAR2(50),
  salary NUMBER(5),
  CONSTRAINT id4_pk PRIMARY
  KEY(id)
);

DECLARE
  a number := '&emp_id';
  b number := '&raise_amount';
  c number;

PROCEDURE Raise_Salary(w_id IN number, amount IN number, raisedSalary OUT number) IS BEGIN

  UPDATE workers
  SET salary = salary + amount WHERE id =
  w_id;

  COMMIT      -- If you want you can remove COMMIT;
  ;          -- DBMS does it automatically to make sure you can type
  SELECT salary INTO raisedSalary FROM workers WHERE id = w_id;

END; BEGIN
  Raise_Salary(a, b, c); dbms_output.put_line('Raised salary is ' || c);
END;
```

INSERT INTO workers VALUES(10001, 'Ali Can', 12000); INSERT INTO workers VALUES(10002, 'Veli Han', 2000); INSERT INTO workers VALUES(10003, 'Mary Star', 7000);
INSERT INTO workers VALUES(10004, 'Angie Ocean', 8500);

Example 4:

--Create a procedure to withdraw money from an account.
--Account id and withdraw amount will be given by user.
--If the withdraw amount is greater than the balance withdraw cannot be done
--otherwise do withdraw and display the remaining balance on the output console.

CREATE TABLE accounts

```
(  
    id CHAR(3),  
    name VARCHAR2(50),  
    balance  
    NUMBER(10,2)  
);
```

```
INSERT INTO accounts VALUES(101, 'Ali Can', 12000);  
INSERT INTO accounts VALUES(102, 'Veli Han', 2000);  
INSERT INTO accounts VALUES(103, 'Mary Star', 7000);  
INSERT INTO accounts VALUES(104, 'Angie Ocean', 8500);
```

```
DECLARE  
    a_id accounts.id%TYPE := '&account_id';  
    w_amount accounts.balance%TYPE :=  
        '&withdraw_amount'; a_balance  
        accounts.balance%TYPE;  
    r_balance accounts.balance%TYPE;  
PROCEDURE withdrawP(account_id IN CHAR, withdraw_amount IN NUMBER, remaining_balance OUT NUMBER) IS  
    BEGIN  
        SELECT balance INTO  
            a_balance FROM accounts  
            WHERE id = account_id;  
  
        IF withdraw_amount <= a_balance THEN UPDATE  
            accounts  
            SET balance = balance - withdraw_amount WHERE id  
            = account_id;  
        ELSE  
            DBMS_OUTPUT.put_line('Insufficient balance!'); END IF;  
  
        SELECT balance  
        INTO remaining_balance  
        FROM accounts  
        WHERE id = account_id;  
  
    END; BEGIN  
    withdrawP(a_id, w_amount, r_balance);  
    DBMS_OUTPUT.put_line('Remaining balance is ' || r_balance);  
    END;
```

HOW TO DROP A PROCEDURE FROM DATABASE

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

Syntax

```
DROP PROCEDURE procedure_name;
```

Example: Let us drop **Raise_Salary** procedure **DROP PROCEDURE Raise_Salary;**

SEQUENCES IN ORACLE SQL

You can create an **auto-number 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.

For example; when you create a student list you may want to create student ids automatically, or when you order products from an online website, the order numbers are created automatically in your cart.

Example 1:

```
CREATE SEQUENCE sequence1;  
CREATE TABLE  
STUDENTS (  
    student_id NUMBER(4),  
    student_name  
    VARCHAR2(50)  
);
```



Sequence SEQUENCE1
created.



After creating sequence you can use it in any
query.



```
INSERT INTO students VALUES(sequence1.NEXTVAL,  
'&student_name');
```

Example 2:

```
CREATE SEQUENCE sequence1  
START WITH 101;  
  
CREATE TABLE  
STUDENTS (  
    student_id NUMBER(4),  
    student_name  
    VARCHAR2(50)  
);
```



Sequence SEQUENCE1
created.



After creating sequence you can use it in any
query.



```
INSERT INTO students VALUES(sequence1.NEXTVAL,  
'&student_name');
```

Example 3:

```
CREATE SEQUENCE  
sequence1 START WITH 101  
INCREMENT BY 2;
```



Sequence SEQUENCE1 created.



After creating sequence you can use it in any query.

```
CREATE TABLE  
STUDENTS (  
    student_id NUMBER(4),  
    student_name  
    VARCHAR2(50)  
);
```



```
INSERT INTO students VALUES(sequence1.NEXTVAL,  
'&student_name');
```

Example 4:

```
CREATE SEQUENCE  
sequence1 START WITH 101  
INCREMENT BY 2  
MAXVALUE 105;
```



Sequence SEQUENCE1 created.



After creating sequence you can use it in any query.

```
CREATE TABLE  
STUDENTS (  
    student_id NUMBER(4),  
    student_name  
    VARCHAR2(50)  
);
```



```
INSERT INTO students VALUES(sequence1.NEXTVAL,  
'&student_name');
```

Example 5:

```
CREATE SEQUENCE sequence2  
START WITH 101  
INCREMENT BY 2  
MAXVALUE 105  
MINVALUE 99  
CYCLE CACHE  
3;
```

```
INSERT INTO bank_customers VALUES(sequence2.NEXTVAL,  
'&customer_name');
```

```
CREATE TABLE  
bank_customers  
customer_order_number  
NUMBER(3), customer_name  
VARCHAR2(50)  
);
```



	CUSTOMER_ORDER_NUMBER	CUSTOMER_NAME
1	101	Ali Can
2	102	Veli Han
3	103	Ayse Tan
4	100	Mary Star
5	101	Angie Ocean

Note: Cache is a memory which holds 20 numbers as default.

If the Cycle is less than 20 you have to declare.

Note: Using Cache improves the performance

HOW TO ALTER SEQUENCES

1) CREATE TABLE bank_customers

```
(  
    customer_order_number NUMBER(3),  
    customer_name VARCHAR2(50)  
);
```

2) CREATE SEQUENCE seq1

```
START WITH 101  
INCREMENT BY 2  
MAXVALUE 105;
```

3) Following code updates the Max Value from 107 to 103

```
ALTER SEQUENCE seq1  
MAXVALUE 103;
```

4) Following code updates the increment value from

```
2 to 1 ALTER SEQUENCE seq1  
INCREMENT BY 1;
```

5) Following code opens a CYCLE

```
ALTER SEQUENCE seq1  
CYCLE  
CACHE 3;
```

Note: START WITH cannot be altered.

If you want to change START WITH value, you should drop then recreate the sequence with a new

Triggers(PL/SQL Block) in Oracle SQL

Basically, there are 2 types of Triggers

- 1) DML Triggers: They are used **before** or **after** **INSERT INTO, UPDATE or DELETE (Table Level)**
- 2) DDL Triggers: They are used **before** or **after** **DROP, TRUNCATE (Database Level)**

Note: **Procedures** are executed manually but **Triggers** are executed automatically by Oracle Server implicitly

Difference Between Before and After Triggers

- 1) Before Triggers are executed before DML Commands **Trigger is Executed**
DML Command is executed
- 2) After Triggers are executed after DML Commands **DML Command is executed**
Trigger is Executed

Levels of DML Triggers

- 1) **Statement Level Triggers:** They are executed just once for the statement.
For example; when you create a trigger for “update” statement even it updates 5 records, the trigger is executed just once.
Note: Triggers are statement level as default
- 2) **Record Level Triggers:** They are executed once for every record.
For example; when you create a record level trigger for “update” statement and if the “update” statement updates 5 records, the trigger is executed five times.

EXAMPLES

```
CREATE TABLE workers
(
    id CHAR(5),
    name VARCHAR2(50),
    salary NUMBER(5)
);
```

```
INSERT INTO workers VALUES(10001, 'Ali Can', 12000);
INSERT INTO workers VALUES(10002, 'Veli Han', 2000);
INSERT INTO workers VALUES(10003, 'Mary Star', 7000);
INSERT INTO workers VALUES(10004, 'Angie Ocean', 8500);
```

Example 1 (DML Trigger):

Create Statement Level Trigger

```
CREATE OR REPLACE TRIGGER workers_trigger BEFORE UPDATE ON
workers BEGIN
    DBMS_OUTPUT.PUT_LINE('Record updated message from
trigger!'); END;
/
```

Create Update
Query UPDATE
workers
SET salary = salary +
100 WHERE salary <
10000;

Note: After running update command you will get “Record updated message from trigger!” Message on DBMS Output console just once even 2 records are updated because Triggers are statement level as default

Example 2 (DML Trigger):

Create Record Level Trigger

```
CREATE OR REPLACE TRIGGER workers_trigger BEFORE UPDATE ON workers FOR EACH
ROW BEGIN
    DBMS_OUTPUT.PUT_LINE('Record updated message from
trigger!'); END;
/
```

Create Update

Query UPDATE

workers

```
SET salary = salary +
100 WHERE salary <
10000;
```

Note: After running update command you will get “Record updated message from trigger!” message on DBMS Output console twice because I added “FOR EACH ROW” in the first row of trigger creation

Example 3 (DML Trigger):

1) **CREATE TABLE** workers
(
 id **CHAR**(5),
 name
 VARCHAR2(50),
 salary **NUMBER**(5)
);

```
INSERT INTO workers VALUES(10001, 'Ali Can', 12000);
INSERT INTO workers VALUES(10002, 'Veli Han', 2000);
INSERT INTO workers VALUES(10003, 'Mary Star', 7000);
INSERT INTO workers VALUES(10004, 'Angie Ocean', 8500);
```

2) **CREATE OR REPLACE TRIGGER** display_salary_changes
BEFORE INSERT OR UPDATE ON workers **FOR EACH**
ROW

```
DECLARE
    salary_diff
    number; BEGIN
        salary_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: ' ||
        salary_diff);
    END;
```

3) **INSERT INTO** workers **VALUES** (10005, 'Kemal Man',
8000);



Old salary:
New salary:
8000 Salary
difference:

4) **UPDATE** workers
SET salary = salary +
500 **WHERE** id = 10002;



Old salary: 9200
New salary: 9700
Salary difference:
500

Example 4 (DDL Trigger):

Create Trigger: BE CAREFUL ==> In the first line at the end you have to type “database name” and “.schema” CREATE OR REPLACE TRIGGER workers_drop_trigger BEFORE DROP ON hr.schema
BEGIN
DBMS_OUTPUT.PUT_LINE('Table dropped message from trigger!'); END;
/

Create Drop Query

DROP TABLE

workers;

Note: After running drop command you will get “Table dropped message from trigger!” message on DBMS Output console

HOW TO DISABLE AND ENABLE TRIGGERS

How to alter a specific TRIGGER

```
ALTER TRIGGER workers_trigger DISABLE; ALTER  
TRIGGER workers_trigger ENABLE;
```

How to alter ALL TRIGGERS on a TABLE

```
ALTER TABLE workers DISABLE ALL  
TRIGGERS; ALTER TABLE workers ENABLE  
ALL TRIGGERS;
```

Example 5 (DML Trigger) Real Time:

When you update salary of a worker, trigger will store old data in workers_history table

- 1)

```
CREATE TABLE workers
(
    id CHAR(5),
    name VARCHAR2(50),
    salary NUMBER(5)
);
```



```
INSERT INTO workers VALUES(10001, 'Ali Can', 12000);
INSERT INTO workers VALUES(10002, 'Veli Han', 2000);
INSERT INTO workers VALUES(10003, 'Mary Star', 7000);
INSERT INTO workers VALUES(10004, 'Angie Ocean', 8500);
```
- 2)

```
CREATE TABLE workers_history
(
    id CHAR(5),
    name VARCHAR2(50),
    salary NUMBER(5),
    m_date DATE
);
```
- 3)

```
CREATE or REPLACE TRIGGER
w_h_trigger BEFORE UPDATE OF salary
ON workers FOR EACH ROW
BEGIN
    INSERT INTO workers_history VALUES      (:old.id,    :old.name,
                                              :old.salary, sysdate); END;
```
- 4)

```
UPDATE workers
SET salary = salary +
500 WHERE id = 10002;
```
- 5)

```
SELECT * FROM
workers_history;
```

Example 6 (DML Trigger) Real Time:

When you update the salary of a worker, insert the following data into salary_changes table Name, Old Salary, New Salary, Raise Amount, date

```
CREATE TABLE workers
(
    id CHAR(5),
    name VARCHAR2(50),
    salary NUMBER(5)
);
INSERT INTO workers VALUES(10001, 'Ali Can', 12000);
INSERT INTO workers VALUES(10002, 'Veli Han', 2000);
INSERT INTO workers VALUES(10003, 'Mary Star', 7000);
INSERT INTO workers VALUES(10004, 'Angie Ocean', 8500);
```

```
CREATE TABLE
salary_changes (
    name VARCHAR2(50),
    old_salary NUMBER(5),
    new_salary NUMBER(5),
    raise_amount
    NUMBER(5),
    update_date DATE
);
```

```
SELECT * FROM salary_changes;
```

```
CREATE OR REPLACE TRIGGER w_c_trigger BEFORE UPDATE OF salary ON workers FOR EACH
ROW BEGIN
    INSERT INTO salary_changes VALUES(:old.name, :old.salary, :new.salary, :new.salary - :old.salary,
SYSDATE); END;
```

```
SELECT * FROM salary_changes;
```

Example 7 (DML Trigger) Real Time:

Note: In that example, we will make payment to a worker and the Trigger will save, payment amount, payment date, worker id etc. into Payment table automatically

Create Payment Table

```
CREATE TABLE  
payments (  
    payment_id  
    CHAR(5), id  
    CHAR(5),  
    payment_date DATE,  
    payment_amount  
    NUMBER(5), salary  
    NUMBER(5)  
);
```

Create a Sequence

```
CREATE SEQUENCE payment_seq;
```

Create Trigger

```
CREATE OR REPLACE TRIGGER payment_trigger BEFORE UPDATE ON workers FOR EACH  
ROW BEGIN  
    INSERT INTO payments VALUES(payment_seq.NEXTVAL, :new.id, SYSDATE,  
                                ABS(:old.salary - :new.salary),  
                                :new.salary); DBMS_OUTPUT.PUT_LINE('Payment is done  
message from trigger!');  
END;
```

Look at the payments table there is nothing
in it **SELECT * FROM payments;**

Create Update
Query UPDATE
workers
SET salary = salary -
100 WHERE id = 10003;

Look at the payments table there is nothing
in it **SELECT * FROM payments;**

Example 8 (DML Trigger) Real Time:

Create a trigger like

When you UPDATE any record or INSERT any record it will be triggered It will store updated or inserted records into "changes" table

```
CREATE TABLE changes (
    order_number NUMBER(3), id CHAR(5),
    name VARCHAR2(50),
    salary NUMBER(5),
    change_date DATE
);
```

```
CREATE SEQUENCE seq_changes;
```

```
CREATE OR REPLACE TRIGGER w_ui_trigger BEFORE INSERT OR UPDATE ON workers FOR EACH ROW BEGIN
    IF :old.id IS NULL THEN
        INSERT INTO changes VALUES(seq_changes.NEXTVAL, :new.id, :new.name, :new.salary, sysdate); ELSE
        INSERT INTO changes VALUES(seq_changes.NEXTVAL, :old.id, :old.name, :old.salary, sysdate); END IF;
END;
```

```
SELECT * FROM changes;
```

```
SELECT * FROM workers;
```

```
INSERT INTO workers VALUES(10005, 'XXX YYY', 6600);
```

```
UPDATE workers
SET name = 'Angie Star' WHERE id
< 10003;
```

Example 9 (DML Trigger) Real Time:

Create a record level trigger like

When you DELETE or INSERT any record it will be triggered

It will store deleted or inserted records with the statement name into "changes" table Use sequence and date in records table

```
CREATE TABLE workers
(
    id CHAR(5),
    name VARCHAR2(50),
    salary NUMBER(5)
);
```

```
INSERT INTO workers VALUES(10001, 'Ali Can', 12000);
INSERT INTO workers VALUES(10002, 'Veli Han', 2000);
INSERT INTO workers VALUES(10003, 'Mary Star', 7000);
INSERT INTO workers VALUES(10004, 'Angie Ocean', 8500);
```

```
CREATE TABLE changes
(
    order_number NUMBER(3),
    type CHAR(20),
    id CHAR(5),
    name VARCHAR2(50),
    salary NUMBER(5),
    change_date DATE
);
```

```
CREATE SEQUENCE seq_changes;
```

```
CREATE OR REPLACE TRIGGER w_di_trigger BEFORE DELETE OR INSERT ON workers FOR EACH
ROW BEGIN
```

```
IF :old.id IS NULL THEN
    INSERT INTO changes VALUES(seq_changes.NEXTVAL, 'Inserted', :new.id, :new.name, :new.salary,
    sysdate); ELSE
    INSERT INTO changes VALUES(seq_changes.NEXTVAL, 'Deleted', :old.id, :old.name, :old.salary,
    sysdate); END IF;
END;
```

```
SELECT * FROM changes;
```

```
DELETE FROM
workers WHERE id =
10001;
```

```
INSERT INTO workers VALUES(10005, 'Ali Can', 8800);
```

CURSORS

Context Area: Oracle creates a memory area, known as the **context area**, for processing an SQL statement, which **contains all the information needed for processing the statement**

Cursor: A cursor is a **pointer to this context area**. PL/SQL controls the context area through a cursor.

A cursor holds the rows (one or more) returned by an SQL statement. The set of rows the cursor holds is called as the **active set**.

What are the advantages of Cursors?

- 1) Cursors can provide the first few rows before the whole result set is assembled.
Without using cursors, the entire result set must be delivered before any rows are displayed by the application. So using cursor, better response time is achieved.
- 2) Cursors can be faster than a loop

What are the disadvantages of Cursors?

- 1) Cursor in SQL is temporary work area created in the system memory, thus it occupies memory from your system that may be available for other processes. So occupies more resources and temporary storage.

THERE ARE TWO TYPES OF CURSORS

1)IMPLICIT CURSORS

2)EXPLICIT CURSORS

Implicit cursors:

Implicit cursors are automatically created by Oracle whenever an SQL statement is executed, when there is no explicit cursor for the statement. Programmers cannot control the implicit cursors and the information in it.

Whenever a DML statement (INSERT, UPDATE and DELETE) is issued, an implicit cursor is associated with this statement.

For INSERT operations, the cursor holds the data that needs to be inserted.

For UPDATE and DELETE operations, the cursor identifies the rows that would be affected.

Note: Implicit cursors have attributes such as **%FOUND**, **%NOTFOUND**, and **%ROWCOUNT**

%FOUND

It returns TRUE if an INSERT, UPDATE, or DELETE statement affected one or more rows or a SELECT INTO statement returned one or more rows. Otherwise, it returns FALSE.

%NOTFOUND

The logical opposite of %FOUND.

It returns TRUE if an INSERT, UPDATE, or DELETE statement affected no rows, or a SELECT INTO statement returned no rows. Otherwise, it returns FALSE.

%ROWCOUNT

Returns the number of rows affected by an INSERT, UPDATE, or DELETE statement, or returned by a SELECT INTO statement.

Example 1 (%FOUND):

```
CREATE TABLE workers
(
    id CHAR(5),
    name VARCHAR2(50),
    salary NUMBER(5)
);
```

```
DECLARE
    emplId workers.id%type;
BEGIN
    emplId := '&id';
    DELETE FROM workers WHERE id = emplId;
    COMMIT;
END;
```

If we type the code like on the right, we can get message together with the above console like "Record is deleted" if the record is found and we get "Record could not be found" message if the record could not be found.

```
INSERT INTO workers VALUES(10001, 'Ali Can', 12000);
INSERT INTO workers VALUES(10002, 'Veli Han', 2000);
INSERT INTO workers VALUES(10003, 'Mary Star', 7000);
INSERT INTO workers VALUES(10004, 'Angie Ocean', 8500);
```

```
old:
DECLARE
    emplId
    workers.id%type;
BEGIN
    emplId := &id;
    DELETE FROM workers WHERE id =
    emplId; COMMIT;
END;
new:
DECLARE
    emplId
    workers.id%type;
BEGIN
    emplId := 10001;
    DELETE FROM workers WHERE id =
    emplId; COMMIT;
    PLS SQL procedure successfully
    completed.
```

When we run the code on the left the console will be like the right part. As you see there is no any information about the data is deleted or not. It says just the procedure successfully completed but user should get a message about the result. To do that we will use "implicit cursor"

```
DECLARE
    emplId workers.id%type; BEGIN
    emplId := &id;
    DELETE FROM workers WHERE id = emplId; IF
    SQL%FOUND THEN
        DBMS_OUTPUT.PUT_LINE('Record is deleted'); ELSE
        DBMS_OUTPUT.PUT_LINE('Record could not be found'); END IF;
    COMMIT; END;
```

Example 2 (%ROWCOUNT):

```
CREATE TABLE workers
(
    id CHAR(5),
    name VARCHAR2(50),
    salary NUMBER(5)
);
INSERT INTO workers VALUES(10001, 'Ali Can', 12000);
INSERT INTO workers VALUES(10002, 'Veli Han', 2000);
INSERT INTO workers VALUES(10003, 'Mary Star', 7000);
INSERT INTO workers VALUES(10004, 'Angie Ocean', 8500);
```

The program will update the table and increase the salary of each customer by 500 and use the **SQL%ROWCOUNT** attribute to determine the number of rows affected

```
DECLARE
    totalRows number(2);
BEGIN
    UPDATE workers
    SET salary = salary + 500;
    IF SQL%NOTFOUND THEN
        dbms_output.put_line('No workers selected');
    ELSIF SQL%FOUND THEN
        totalRows := SQL%ROWCOUNT;
        dbms_output.put_line(totalRows || ' workers selected');
    END IF;
END;
```



4 workers
selected

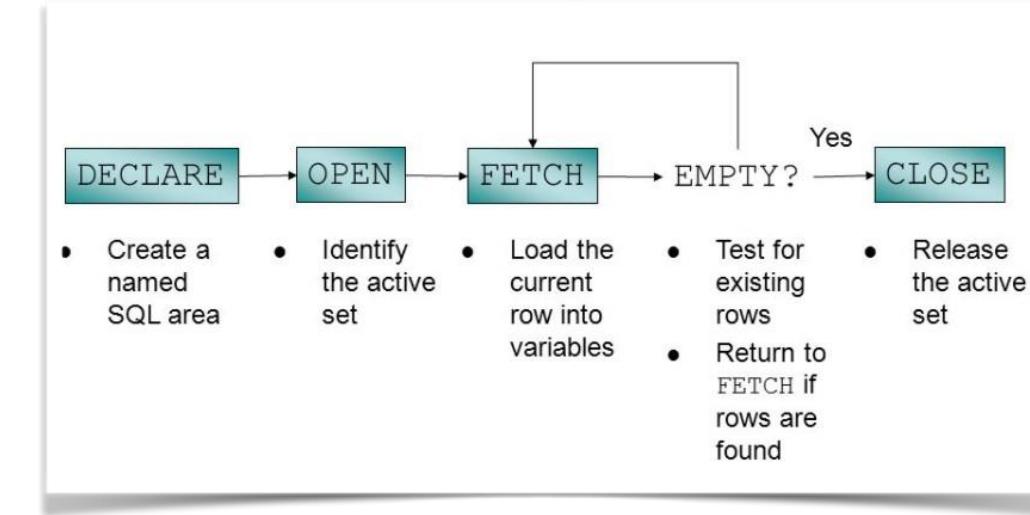
Explicit cursors:

Explicit cursors are **programmer-defined cursors** for gaining more control over the context area. An explicit cursor should be **defined in the declaration section** of the PL/SQL Block.

It is **created on a SELECT Statement** which returns more than one row.

Steps to create Explicit Cursor:

- 1) **Declare Cursor**
- 2) **Open Cursor**
- 3) **Fetch Cursor**
- 4) **Close Cursor**



Example 1:

Your manager asks you to prepare a report like 10% raise for the salaries greater than or equal to 9000 and 20% raise for the salaries less than 9000. You do not update the table, your manager just wants to see the report.

```
DECLARE  
employee_name raised_salary workers.name%TYPE;  
workers.salary%TYPE;  
  
CURSOR cursor1 IS SELECT name, salary FROM workers; BEGIN  
OPEN cursor1; LOOP  
FETCH cursor1 INTO employee_name, raised_salary;  
IF raised_salary >= 9000 THEN raised_salary := raised_salary*1.1;  
ELSE  
raised_salary := raised_salary*1.2; END IF;  
EXIT WHEN cursor1%NOTFOUND;  
DBMS_OUTPUT.PUT_LINE(employee_name||' '||raised_salary); END LOOP;  
CLOSE cursor1; END;
```

Example 2: (2nd way to solve Example 1 by using For-Loop)

Your manager asks you to prepare a report like 10% raise for the salaries greater than or equal to 9000 and 20% raise for the salaries less than 9000. You do not update the table, your manager just wants to see the report.

```
DECLARE
  CURSOR cursor2 IS SELECT name, salary FROM
workers; BEGIN
  FOR r IN cursor2 LOOP
    IF r.salary>=9000
      THEN r.salary :=
r.salary*1.1;
    ELSE
      r.salary := r.salary*1.2;
    END IF;
    DBMS_OUTPUT.PUT_LINE(r.name||'
'||r.salary); END LOOP;
END;
```

Note: If you use **FOR-LOOP** with Cursor, no need to **Open**, **Fetch** and **Close Cursor** just **Declaration** will be enough.

Note: If you use **FOR-LOOP** with Cursor, no need to declare variables, just using “**r**” is enough.
“**r**” is a variable declared by Oracle Server implicitly and it is **RowType variable** and stores all row.

Example 3: (3rd way to solve Example 1 by using While-Loop)

Your manager asks you to prepare a report like 10% raise for the salaries greater than 9000 and 20% raise for the salaries less than 9000. You do not update the table, your manager just wants to see the report.

```
DECLARE
    CURSOR cursor3 IS SELECT name, salary FROM
    workers; workers_data cursor3%ROWTYPE;
BEGIN
    OPEN cursor3;
    FETCH cursor3 INTO
    workers_data; WHILE
        cursor3%FOUND LOOP
            IF workers_data.salary>9000 THEN
                workers_data.salary :=
                workers_data.salary*1.1; ELSE
                workers_data.salary :=
                workers_data.salary*1.2; END IF;
                DBMS_OUTPUT.PUT_LINE(workers_data.name||' -
               '||workers_data.salary); FETCH cursor3 INTO workers_data;
        END LOOP;
        CLOSE
    cursor3;
END;
```

PACKAGES

```
CREATE TABLE workers
(
    id CHAR(5),
    name VARCHAR2(50),
    salary NUMBER(5)
);

INSERT INTO workers VALUES(10001, 'Ali Can', 12000);
INSERT INTO workers VALUES(10002, 'Veli Han', 2000);
INSERT INTO workers VALUES(10003, 'Mary Star', 7000);
INSERT INTO workers VALUES(10004, 'Angie Ocean', 8500);
```

Package Creation:

```
CREATE OR REPLACE PACKAGE workers_package AS

    -- Adds a worker
    PROCEDURE addWorker(w_id workers.id%TYPE, w_name workers.name%TYPE, w_sal
workers.salary%TYPE);

    -- Removes a worker
    PROCEDURE delWorker(w_id workers.id%TYPE);

    --Lists all customers
    PROCEDURE
    listWorkers;

END workers_package;
```

Package Body Creation:

```
CREATE OR REPLACE PACKAGE BODY workers_package AS
```

```
PROCEDURE addWorker(w_id workers.id%TYPE, w_name workers.name%TYPE, w_sal  
workers.salary%TYPE) IS BEGIN  
    INSERT INTO workers(id, name, salary) VALUES(w_id, w_name,  
w_sal); END addWorker;
```

```
PROCEDURE delWorker(w_id  
workers.id%TYPE) IS BEGIN  
    DELETE FROM  
workers WHERE id =  
w_id;  
END delWorker;
```

```
PROCEDURE listWorkers IS  
CURSOR w_cursor IS SELECT name FROM  
workers; BEGIN  
    FOR r IN w_cursor LOOP  
        dbms_output.put_line(r.nam  
e);  
    EXIT WHEN  
        w_cursor%NOTFOUND  
    ; END LOOP;  
    dbms_output.put_line('All workers names are  
listed'); END listWorkers;
```

```
END workers_package;
```

USAGE OF THINGS IN THE PACKAGE:

Example 1:

- 1) Lists the names of workers before adding workers
- 2) Adds Canan Can and Ayse Yaman to the workers table
- 3) Lists the names of workers after adding workers

BEGIN

```
workers_package.listWorkers;  
workers_package.addWorker(1005, 'Canan Can', 22000);  
workers_package.addWorker(1006, 'Ayse Yaman', 33000);  
workers_package.listWorkers; END;
```

Example 2:

- 1) Lists the names of workers before deleting worker
- 2) Deletes the worker by using given id
- 3) Lists the names of workers after deleting worker

DECLARE

```
w_id workers.id%type:= '&worker_id';  
BEGIN  
    workers_package.listWorkers;  
    workers_package.delWorker(w_id);  
    workers_package.listWorkers; END;
```

SQL TECHNICAL INTERVIEW QUESTIONS

CREATE TABLE

```
students
(
    id          number(9),
    name        varchar2(50),
    state       varchar2(50),
    salary      number(20),
    company     varchar2(20)
);
```

```
INSERT INTO students VALUES(123456789, 'Johnny Walk', 'New Hampshire', 2500, 'IBM');
INSERT INTO students VALUES(234567891, 'Brian Pitt', 'Florida', 1500, 'LINUX');
INSERT INTO students VALUES(245678901, 'Eddie Murphy', 'Texas', 3000, 'WELLS FARGO');
INSERT INTO students VALUES(456789012, 'Teddy Murphy', 'Virginia', 1000, 'GOOGLE');
INSERT INTO students VALUES(567890124, 'Eddie Murphy', 'Massachuset', 7000, 'MICROSOFT');
INSERT INTO students VALUES(456789012, 'Brad Pitt', 'Texas', 1500, 'TD BANK');
INSERT INTO students VALUES(123456719, 'Adem Stone', 'New Jersey', 2500, 'IBM');
```

ID	NAME	STATE	SALARY	COMPANY
123456789	Johnny Walk	New Hampshire	2500	IBM
234567891	Brian Pitt	Florida	1500	LINUX
245678901	Eddie Murphy	Texas	3000	WELLS FARGO
456789012	Teddy Murphy	Virginia	1000	GOOGLE
567890124	Eddie Murphy	Massachuset	7000	MICROSOFT
456789012	Brad Pitt	Texas	1500	TD BANK
123456719	Adem Stone	New Jersey	2500	IBM

CREATE TABLE employees

```
(

    id          number(9),
    name        varchar2(50),
    state       varchar2(50),
    salary      number(20),
    company     varchar2(20)
);
```

```
INSERT INTO employees VALUES(123456789, 'John Walker', 'Florida', 2500, 'IBM');
INSERT INTO employees VALUES(234567890, 'Brad Pitt', 'Florida', 1500, 'APPLE');
INSERT INTO employees VALUES(345678901, 'Eddie Murphy', 'Texas', 3000, 'IBM');
INSERT INTO employees VALUES(456789012, 'Eddie Murphy', 'Virginia', 1000, 'GOOGLE');
INSERT INTO employees VALUES(567890123, 'Eddie Murphy', 'Texas', 7000, 'MICROSOFT');
INSERT INTO employees VALUES(456789012, 'Brad Pitt', 'Texas', 1500, 'GOOGLE');
INSERT INTO employees VALUES(123456710, 'Mark Stone', 'Pennsylvania', 2500, 'IBM');
```

ID	NAME	STATE	SALARY	COMPANY
123456789	John Walker	Florida	2500	IBM
234567890	Brad Pitt	Florida	1500	APPLE
345678901	Eddie Murphy	Texas	3000	IBM
456789012	Eddie Murphy	Virginia	1000	GOOGLE
567890123	Eddie Murphy	Texas	7000	MICROSOFT
456789012	Brad Pitt	Texas	1500	GOOGLE
123456710	Mark Stone	Pennsylvania	2500	IBM

1) How to fetch common records from two tables?

```
SELECT id, name
FROM students
INTERSECT
SELECT id, name
FROM employees;
```

ID	NAME
456789012	Brad Pitt

```
SELECT name
FROM students
INTERSECT
SELECT name FROM
employees;
```

NAME
Brad Pitt
Eddie Murphy

2) How to fetch unique records from a table?

`SELECT DISTINCT state
FROM employees;`



STATE
Florida
Pennsylvania
Virginia
Texas

Note: DISTINCT Clause is used to remove duplicates from the result set.

3) What is the command used to fetch even id's?

`SELECT *
FROM students
WHERE
MOD(id,2)=0;`



ID	NAME	STATE	SALARY	COMPANY
456789012	Teddy Murphy	Virginia	1000	GOOGLE
567890124	Eddie Murphy	Massachusetts	7000	MICROSOFT
456789012	Brad Pitt	Texas	1500	TD BANK

Note: To fetch odd id's use the following script

`SELECT *
FROM students
WHERE
MOD(id,2)=1;`

4) What is the command to count records in a table?

```
SELECT COUNT(*) AS  
number_of_records FROM students;
```

A diagram illustrating the execution flow of an SQL query. On the left, a red-bordered box contains the SQL code: "SELECT COUNT(*) AS number_of_records FROM students;". An arrow points from this box to a small rectangular window titled "NUMBER_OF_RECORDS" which displays the value "7".

NUMBER_OF_RECORDS	
7	

5) What is the SQL Query to get the highest salary of a worker from a table?

```
SELECT MAX(salary) AS  
maximum_salary FROM workers;
```

```
SELECT salary  
FROM employees  
ORDER BY salary DESC  
FETCH NEXT 1 ROW ONLY;
```

A diagram illustrating the execution flow of two SQL queries. Two red-bordered boxes contain the queries: "SELECT MAX(salary) AS maximum_salary FROM workers;" and "SELECT salary FROM employees ORDER BY salary DESC FETCH NEXT 1 ROW ONLY;". Arrows point from both boxes to a small rectangular window titled "MAXIMUM_SALARY" which displays the value "7000".

MAXIMUM_SALARY	
7000	

6) What is the SQL Query to get all records about the worker who has the highest salary from a table?

```
SELECT *  
FROM workers  
WHERE salary = (SELECT  
MAX(salary)  
FROM workers);
```

```
SELECT *  
FROM employees  
ORDER BY salary  
DESC  
FETCH NEXT 1 ROW ONLY;
```

A diagram illustrating the execution flow of two SQL queries. Two red-bordered boxes contain the queries: "SELECT * FROM workers WHERE salary = (SELECT MAX(salary) FROM workers);" and "SELECT * FROM employees ORDER BY salary DESC FETCH NEXT 1 ROW ONLY;". Arrows point from both boxes to a larger table showing a single row of data for a worker with ID 567890123, name Eddie Murphy, state Texas, salary 7000, company Microsoft, country USA, and gender -.

ID	NAME	STATE	SALARY	COMPANY_NAME	COUNTRY_NAME	GENDER
567890123	Eddie Murphy	Texas	7000	MICROSOFT	USA	-

7) What is the SQL Query to get the **second highest salary** of a worker from a table?

```
SELECT MAX(salary) AS  
second_maximum_salary FROM employees  
WHERE salary < (SELECT MAX(salary)  
FROM employees);
```

```
SELECT salary  
FROM  
employees  
ORDER BY salary  
DESC OFFSET 1 ROW  
FETCH NEXT 1 ROW ONLY;
```

SECOND_MAXIMUM_SALARY
3000

8) What is the SQL Query to get **all records about the worker who has the second highest salary** from a table?

ID	NAME	STATE	SALARY	COMPANY
123456789	John Walker	Florida	2500	IBM
234567890	Brad Pitt	Florida	1500	APPLE
345678901	Eddie Murphy	Texas	3000	IBM
456789012	Eddie Murphy	Virginia	1000	GOOGLE
567890123	Eddie Murphy	Texas	7000	MICROSOFT
456789012	Brad Pitt	Texas	1500	GOOGLE
123456710	Mark Stone	Pennsylvania	2500	IBM

```
SELECT *  
FROM (SELECT *  
      FROM employees  
      WHERE salary < (SELECT MAX(salary)  
                       FROM  
                       employees) ORDER BY salary  
              DESC)  
      FETCH NEXT 1 ROW ONLY;
```

ID	NAME	STATE	SALARY	COMPANY_NAME	COUNTRY_NAME	GENDER
345678901	Eddie Murphy	Texas	3000	IBM	USA	-

```
SELECT *  
FROM employees  
ORDER BY salary  
DESC OFFSET 1 ROW  
FETCH NEXT 1 ROW ONLY;
```

9) What is the SQL Query to get all records from a column in uppercase from a table?

NAME	STATE
John Walker	Florida
Brad Pitt	Florida
Eddie Murphy	Texas
Eddie Murphy	Virginia
Eddie Murphy	Texas
Brad Pitt	Texas
Mark Stone	Pennsylvania

`SELECT name,
 UPPER(state) FROM
 workers;`

NAME	UPPER(STATE)
John Walker	FLORIDA
Brad Pitt	FLORIDA
Eddie Murphy	TEXAS
Eddie Murphy	VIRGINIA
Eddie Murphy	TEXAS
Brad Pitt	TEXAS
Mark Stone	PENNSYLVANIA

10) What is the SQL Query to get all records from a column in lowercase from a table?

NAME	STATE
John Walker	Florida
Brad Pitt	Florida
Eddie Murphy	Texas
Eddie Murphy	Virginia
Eddie Murphy	Texas
Brad Pitt	Texas
Mark Stone	Pennsylvania

`SELECT name,
 LOWER(state) FROM
 workers;`

NAME	LOWER(STATE)
John Walker	florida
Brad Pitt	florida
Eddie Murphy	texas
Eddie Murphy	virginia
Eddie Murphy	texas
Brad Pitt	texas
Mark Stone	pennsylvania

11) What is the SQL Query to get all records from a column in initials uppercase rests lowercase from a table?

NAME	COMPANY_NAME
John Walker	IBM
Brad Pitt	APPLE
Eddie Murphy	IBM
Eddie Murphy	GOOGLE
Eddie Murphy	MICROSOFT
Brad Pitt	GOOGLE
Mark Stone	IBM

`SELECT name,
 INITCAP(state) FROM
 workers;`

NAME	INITCAP(COMPANY_NAME)
John Walker	Ibm
Brad Pitt	Apple
Eddie Murphy	Ibm
Eddie Murphy	Google
Eddie Murphy	Microsoft
Brad Pitt	Google
Mark Stone	Ibm