SQL Introduction

IMPORTANCE OF DATA

Modern society is driven by data.

Whether it is at a personal level, like a notebook containing scribbled notes; or at a countrywide level like Census data.

There is always a growing need to efficiently store and organize it so that meaningful information can be extracted out of raw data.

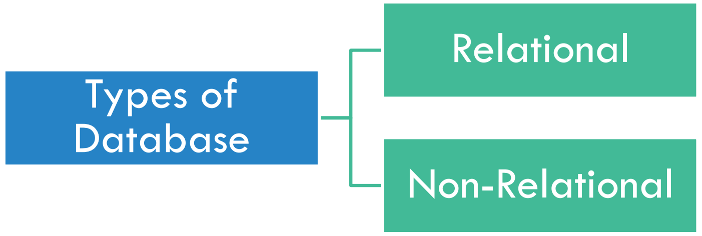
If collecting, storing & organizing data is important, do we have a capability to do it?

DATABASE

- A **database** is nothing but a collection of organized data. It doesn't have to be in a digital format to be called a database.
- A telephone directory is a good example, which stores data about people and organizations with a contact number.
- Other Examples:

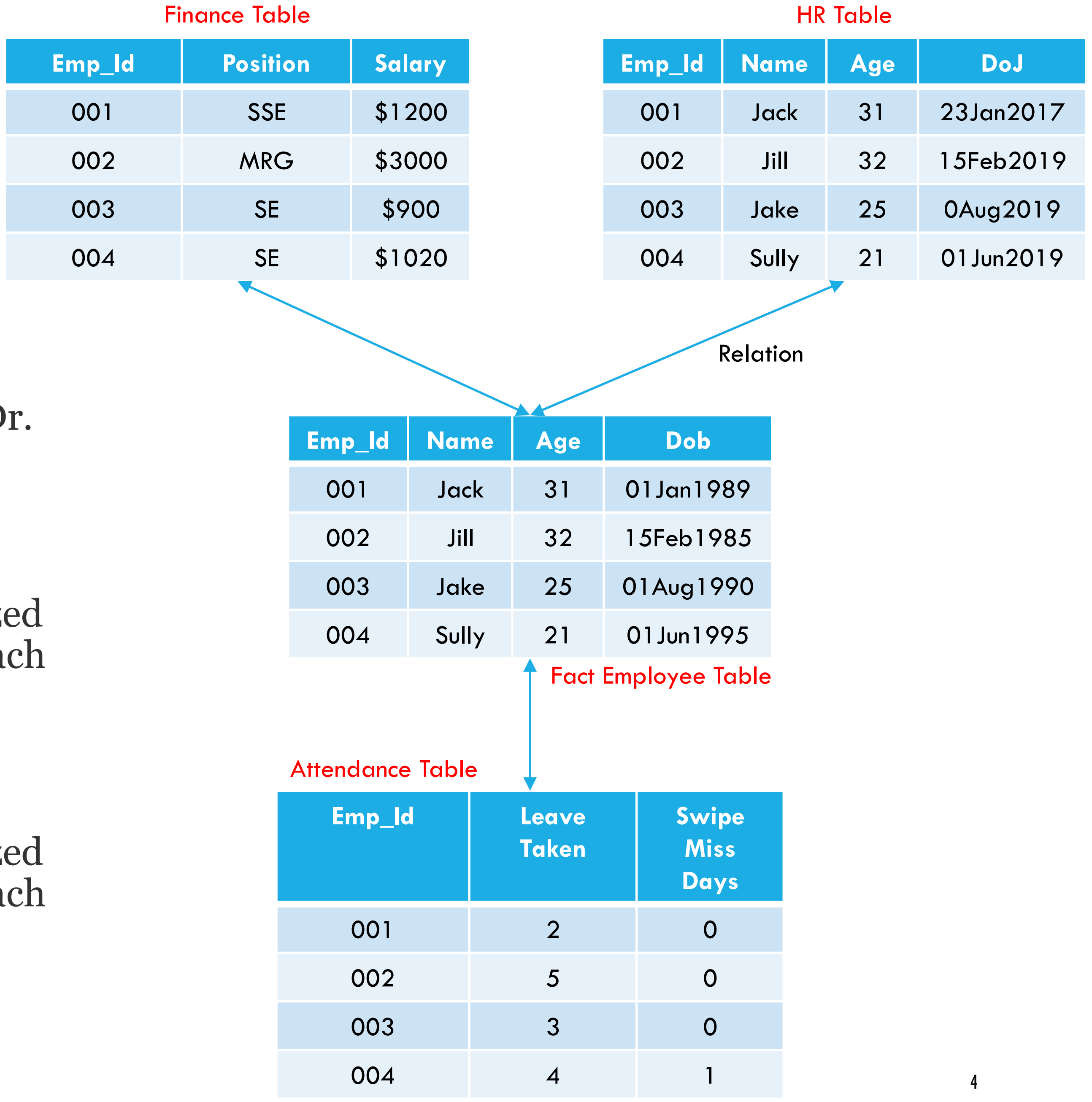
    1. Personal records of Students
    2. Students performance history
    3. Teachers data
    4. Financial department data etc.

- With ever-larger amounts of data being collected about even the most mundane of processes, digital databases have become increasingly important since their inception in the 1960s.
- Software that is used to manage a digital database is called a ***Database Management System (DBMS)*** **.**



THE RELATIONAL MODEL AND SQL

- The most prevalent database organizational model is the Relational Model, developed by Dr. E. F. Codd in his groundbreaking research paper – A Relational Model of Data for Large Shared Data Banks in 1970.
- In this model, the data to be stored is organized in a tabular format with rows and columns. Each row inside a table represents a distinct record with the column headings specifying the corresponding type of data stored.
- In this model, the data to be stored is organized in a tabular format with rows and columns. Each row inside a table represents a distinct record with the column headings specifying the corresponding type of data stored.



Finance Table

Emp_Id	Position	Salary
001	SSE	$1200
002	MRG	$3000
003	SE	$900
004	SE	$1020

HR Table

Emp_Id	Name	Age	DoJ
001	Jack	31	23Jan2017
002	Jill	32	15Feb2019
003	Jake	25	0Aug2019
004	Sully	21	01Jun2019

Emp_Id	Name	Age	Dob
001	Jack	31	01Jan1989
002	Jill	32	15Feb1985
003	Jake	25	01Aug1990
004	Sully	21	01Jun1995

Fact Employee Table

Attendance Table

Emp_Id	Leave Taken	Swipe Miss Days
001	2	0
002	5	0
003	3	0
004	4	1

THE RELATIONAL MODEL AND SQL

What does the word *relational* in relational database mean?

It is a common misconception that the word relational implies a relationship between the tables. A relation is a mathematical term that is roughly equivalent to a table itself. When used in conjunction with the word database, we mean to say that this particular system arranges data in a tabular fashion.

A possible origin of this misconception might have been the *set relation* command in *dBase*, a DBMS from the 1980s. That command indeed was used to create linkages between tables, but it has nothing to do with relational theory.

SQL

- SQL stands for **Structured Query Language**, and it is the de facto standard for interacting with relational databases.
- Almost all database management systems you'll come across will have an SQL implementation.
- SQL was standardized by the American National Standards Institute (ANSI) in 1986 and has undergone many revisions, most notably in 1992 and 1999.
- While SQL is a computer language, it is not like the other programming languages that you may have heard of like Python or C. Such programming languages are generic in nature, suitable for a wide variety of tasks from programming basic calculating systems to advanced simulation models. SQL is a special purpose query language meant for interacting with relational databases. It has no use other than this context.
- This does not mean that it is the only database query language to exist. In the 1980s, another language called **QUEL from Ingres** was fairly popular, but the standardization effort around SQL cemented its position. In recent years, we have seen a large number of non-relational databases being developed under the umbrella term of NoSQL. Most of their query languages, however, bear some resemblance to SQL even though their data model varies significantly from the relational model.

ADVANTAGES OF USING SQL

- It is standardized – no matter which relational database you choose, it will have an SQL query interpreter built in. The sheer popularity of SQL makes it worth everyone's time who interacts with a data system.

- It has a reasonable English-like syntax. None of the painstaking detail of programming languages like C or Java have to be specified when using SQL. It is concise, easy to understand, and easy to write database queries with. It is declarative in nature, meaning you only have to declare what you want to achieve rather than going over the steps to achieve the results.

- It allows a uniform way to query and administer a relational database. Many of the database administration commands are standard SQL commands making the transfer of skills much easier.

- It is mature – SQL has been around for over 35 years. While many new features have been added to it, the core of SQL has largely been unchanged. You can derive a lot of utility knowing a few basic SQL concepts and commands, and they will serve you well into the future.

SQL COMMANDS CLASSIFICATION

SQL is a language for interacting with databases. It consists of a number of commands with further options to allow you to carry out your operations with a database. While DBMS's differ in the command subset they provide, usually you would find the classifications below.

Data Definition Language (DDL): CREATE TABLE, ALTER TABLE, DROP TABLE, etc. These commands allow you to create or modify your database structure.

Data Manipulation Language (DML): INSERT, UPDATE, DELETE. These commands are used to manipulate data stored inside your database.

Data Query Language (DQL): SELECT. Used for querying or selecting a subset of data from a database.

Data Control Language (DCL): GRANT, REVOKE, etc. Used for controlling access to data within a database, commonly used for granting user privileges.

Transaction Control Commands: COMMIT, ROLLBACK, etc. Used for managing groups of statements as a unit of work.

Besides these, your database management system may give you other sets of commands to work more efficiently or to provide extra features. But it is safe to say that the ones above would be present in almost all DBMS's you encounter.

EXPLAINING TABLES

A table in a relational database is nothing but a two-dimensional matrix of data where the columns describe the type of data, and the row contains the actual data to be stored.

Emp_Id	Name	Age	Dob
001	Jack	31	01Jan1989
002	Jill	32	15Feb1985
003	Jake	25	01Aug1990
004	Sully	21	01Jun1995

The above table stores data about programming languages. It consists of four columns (id, language, author, and year) and three rows. The formal term for a column in a database is a *field* and a row is known as a *record*.

EXPLAINING TABLES

There are two things of note in the example table.

Emp_Id	Name	Age	Dob
001	Jack	31	01Jan1989
002	Jill	32	15Feb1985
003	Jake	25	01Aug1990
004	Sully	21	01Jun1995

The first one is that the id field effectively tells you nothing about the programming language by itself, other than its sequential position in the table.

The second is that though we can understand the fields by looking at their names, we have not formally assigned a data type to them, that is, we have not restricted (not yet anyways) whether a field should contain alphabets or numbers or a combination of both.

The *id* field here serves the purpose of a *primary key* in the table

When users create a spreadsheet, they associate a file name with the spreadsheet and place it somewhere on their disk. But relational databases hide all these details from the user. The physical storage of a table on the disk might be to a single file, or to many files, or even have a relationship of storing many tables in a single file. It is the responsibility of your DBMS to provide a way to read and write to tables.

DATA TYPES IN SQL

Just like programming languages, SQL also has *data types* to define the kind of data that will be stored in its fields.

Character types	char, varchar
Integer values	integer, smallint
Decimal numbers	numeric, decimal
Date data type	date

Identity the data Types

Emp_Id	**Name**	**Age**	**Dob**
001	Jack	31	01Jan1989
002	Jill	32	15Feb1985
003	Jake	25	01Aug1990
004	Sully	21	01Jun1995

CHARACTER DATATYPE

A string of characters is usually stored in either *char* or *varchar*. The former reserves as much space as you want when you specify the field, but if the value you store in it is shorter, the remaining space is wasted.
A *varchar*, however, stands for a varying character and will occupy the exact length of the string, nothing wasted. There is, however, a maximum limit to how long a string value you can assign to such a field, and that is specified during the field definition itself.

```
char(12)
varchar(12)
```

If you store the value 'McCarthy' that is eight characters long, the *char* will store it but waste four characters. The *varchar* will store it as exactly eight characters but the whole dynamism comes at a cost of speed. Nonetheless, the speed difference is small enough that for most scenario's you would see the varying character data type being used.

INTEGER DATATYPE

In case of number values, we get a split across two major classes – integer for storing whole numbers and numeric for storing number values with a decimal point in them. The ranges and limits of the values being stored in them vary with your choice of DBMS . However, a good rule of thumb to follow is to use the smallest data type that will suffice for the present and foreseeable future of your application.

For example, if I were storing student roll numbers, using a smallint would suit just fine. In most implementations, this data type allows a maximum value of 32767, a number I mostly expect to be much greater than the number of students in any class.

Decimal point numbers are trickier to specify. We use the numeric data type to fix how large the number could be and how many numbers can occur after the decimal point.

The total number of digits is specified by the precision and the number of digits after the decimal point is represented by scale. So in the example given, we would be able to store a number like 999.99 but not any further.

```
numeric(precision, scale)
numeric(5, 2)
```

QUERY

A Query is a set of instruction given to the database management system, which tells RDBMS what information you would like to get from the database.

For e.g. to fetch the employee name from the database table EMPLOYEE, we write the SQL Query like this:

```
SELECT name from EMPLOYEE;
```

CREATE DATABASE

The CREATE DATABASE statement is used to create a new User SQL database.

Syntax

```
CREATE DATABASE databasename;
```

Example

```
CREATE DATABASE testDB;
```

Task

```
Create a Database with your Name
```

DROP DATABASE

The DROP DATABASE statement is used to drop an existing SQL database.

Syntax

```
DROP DATABASE databasename;
```

Example

```
DROP DATABASE testDB;
```

Task

Drop the Database created with your Name

DATA DEFINITION LANGUAGE (DDL):

CREATE TABLE, ALTER TABLE, DROP TABLE, etc. These commands allow you to create or modify your database structure.

CREATE TABLE

Syntax

```
CREATE TABLE table_name (
    column1 datatype,
    column2 datatype,
    column3 datatype,
   ....
);
```

Example

```
CREATE TABLE Persons (
    PersonID int,
    LastName varchar(255),
    FirstName varchar(255),
    Address varchar(255),
    City varchar(255)
);
```

Task

Create a table as per your imagination

DATA DEFINITION LANGUAGE (DDL):

CREATE TABLE USING ANOTHER TABLE

Syntax

```
CREATE TABLE new_table_name AS
    SELECT column1, column2,...
    FROM existing_table_name;
```

Example

```
CREATE TABLE TestTable AS
SELECT customername, contactname
FROM customers;
```

Task

Create a table as per your imagination

DATA DEFINITION LANGUAGE (DDL):

DROP TABLE

Syntax

```
DROP TABLE table_name;
```

Example

```
DROP TABLE JACK;
```

Task

Drop the table you have created

1.This command will completely destroy Structure and Data

DATA DEFINITION LANGUAGE (DDL):

TRUNCATE TABLE

Syntax

```
TRUNCATE TABLE table_name;
```

Example

```
TRUNCATE TABLE Jack;
```

Task

Truncate the table you have created

1.This command will **permanently** remove all the rows permanently.
2.If we roll back also we won’t get the data.

DATA DEFINITION LANGUAGE (DDL):

ALTER TABLE - ADD Column

```
ALTER TABLE table_name                  ALTER TABLE Customers
ADD column_name datatype;               ADD Email varchar(255);
```

ALTER TABLE - Drop Column

```
ALTER TABLE table_name                  ALTER TABLE Customers
DROP COLUMN column_name;                DROP COLUMN Email;
```

ALTER TABLE - ALTER/MODIFY COLUMN

```
ALTER TABLE table_name                   ALTER TABLE Persons
MODIFY COLUMN column_name datatype;  MODIFY COLUMN DateOfBirth year;
```

SQL CONSTRAINTS

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

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

```
CREATE TABLE table_name (
    column1 datatype constraint,
    column2 datatype constraint,
    column3 datatype constraint,
    ....
);
```

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

SQL CONSTRAINTS

The following constraints are commonly used in SQL:

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

SQL NOT NULL CONSTRAINT

- By default, a column can hold NULL values.
- The NOT NULL constraint enforces a column to NOT accept NULL values.
- This enforces a field to always contain a value, which means that you cannot insert a new record, or update a record without adding a value to this field.

SQL NOT NULL on CREATE TABLE

```
CREATE TABLE Persons (
    ID int NOT NULL,
    LastName varchar(255) NOT NULL,
    FirstName varchar(255) NOT NULL,
    Age int
);
```

SQL NOT NULL on ALTER TABLE

```
ALTER TABLE Persons
MODIFY Age int NOT NULL;
```

SQL UNIQUE CONSTRAINT

The UNIQUE constraint ensures that all values in a column are different.
Both the UNIQUE and PRIMARY KEY constraints provide a guarantee for uniqueness for a column or set of columns.
A PRIMARY KEY constraint automatically has a UNIQUE constraint.
However, you can have many UNIQUE constraints per table, but only one PRIMARY KEY constraint per table.

SQL UNIQUE Constraint on CREATE TABLE

```
CREATE TABLE Persons (
    ID int NOT NULL,
    LastName varchar(255) NOT NULL,
    FirstName varchar(255),
    Age int,
    CONSTRAINT UC_Person UNIQUE (ID,LastName)
);
```

SQL UNIQUE Constraint on ALTER TABLE

```
ALTER TABLE Persons
ADD CONSTRAINT UC_Person UNIQUE (ID,LastName);

ALTER TABLE Persons
DROP CONSTRAINT UC_Person;
```

PersonID	LastName	FirstName	Age
1	Hansen	Ola	30
2	Svendson	Tove	23
3	Pettersen	Kari	20

SQL PRIMARY KEY CONSTRAINT

The PRIMARY KEY constraint uniquely identifies each record in a table.
Primary keys must contain UNIQUE values, and cannot contain NULL values.
A table can have only ONE primary key; and in the table, this primary key can consist of single or multiple columns (fields).

```
CREATE TABLE Persons (
    ID int NOT NULL PRIMARY KEY,
    LastName varchar(255) NOT NULL,
    FirstName varchar(255),
    Age int
);
```

```
CREATE TABLE Persons (
    ID int NOT NULL,
    LastName varchar(255) NOT NULL,
    FirstName varchar(255),
    Age int,
    CONSTRAINT PK_Person PRIMARY KEY (ID,LastName)
);
```

```
ALTER TABLE Persons
ADD CONSTRAINT PK_Person PRIMARY KEY
 (ID,LastName);
```

```
ALTER TABLE Persons
DROP CONSTRAINT PK_Person;
```

SQL FOREIGN KEY CONSTRAINT

A FOREIGN KEY is a key used to link two tables together.
A FOREIGN KEY is a field (or collection of fields) in one table that refers to the PRIMARY KEY in another table.
The table containing the foreign key is called the child table, and the table containing the candidate key is called the referenced or parent table.

PersonID	LastName	FirstName	Age
1	Hansen	Ola	30
2	Svendson	Tove	23
3	Pettersen	Kari	20

OrderID	OrderNumber	PersonID
1	77895	3
2	44678	3
3	22456	2
4	24562	1

The "PersonID" column in the "Persons" table is the PRIMARY KEY in the "Persons" table.
The "PersonID" column in the "Orders" table is a FOREIGN KEY in the "Orders" table.
The FOREIGN KEY constraint is used to prevent actions that would destroy links between tables.
The FOREIGN KEY constraint also prevents invalid data from being inserted into the foreign key column, because it has to be one of the values contained in the table it points to.

SQL CHECK CONSTRAINT

The CHECK constraint is used to limit the value range that can be placed in a column.
If you define a CHECK constraint on a single column it allows only certain values for this column.
If you define a CHECK constraint on a table it can limit the values in certain columns based on values in other columns in the row.

SQL CHECK on CREATE TABLE

```
CREATE TABLE Persons (
    ID int NOT NULL,
    LastName varchar(255) NOT NULL,
    FirstName varchar(255),
    Age int CHECK (Age>=18)
);
```

SQL CHECK on ALTER TABLE

```
ALTER TABLE Persons
ADD CONSTRAINT CHK_PersonAge
 CHECK (Age>=18 AND City='Sa
ndnes');
```

```
ALTER TABLE Persons
ADD CHECK (Age>=18);
```

SQL DEFAULT CONSTRAINT

The DEFAULT constraint is used to provide a default value for a column.
The default value will be added to all new records IF no other value is specified.

SQL DEFAULT on CREATE TABLE

```
CREATE TABLE Persons (
    ID int NOT NULL,
    LastName varchar(255) NOT NULL,
    FirstName varchar(255),
    Age int,
    City varchar(255) DEFAULT 'Sandnes'
);
```

SQL DEFAULT on ALTER TABLE

```
ALTER TABLE Persons
ADD CONSTRAINT df_City
DEFAULT 'Sandnes' FOR City;
```

SQL CREATE INDEX STATEMENT

The CREATE INDEX statement is used to create indexes in tables.
Indexes are used to retrieve data from the database very fast. The users cannot see the indexes, they are just used to speed up searches/queries.

```
CREATE INDEX index_name
ON table_name (column1, column2, ...);

CREATE UNIQUE INDEX index_name
ON table_name (column1, column2, ...);
```

```
CREATE INDEX idx_lastname
ON Persons (LastName);

CREATE INDEX idx_pname
ON Persons (LastName, FirstName);
```

```
DROP INDEX table_name.index_name;
```

DATA MANIPULATION LANGUAGE (DML):

INSERT, UPDATE, DELETE. These commands are used to manipulate data stored inside your database.

The SQL INSERT INTO Statement

The INSERT INTO statement is used to insert new records in a table.

```
INSERT INTO table_name (column1, column2, column3, ...)
VALUES (value1, value2, value3, ...);
```

```
INSERT INTO Customers (CustomerName, ContactName, Address, City, PostalCode, Country)
VALUES ('Cardinal', 'Tom B. Erichsen', 'Skagen 21', 'Stavanger', '4006', 'Norway');
```

DATA MANIPULATION LANGUAGE (DML):

The SQL UPATE Statement

The UPDATE statement is used to modify the existing records in a table.

```
UPDATE table_name
SET column1 = value1, column2 = value2, ...
WHERE condition;
```

```
UPDATE Customers
SET ContactName = 'Alfred Schmidt',
City= 'Frankfurt'
WHERE CustomerID = 1;
```

```
UPDATE Customers
SET ContactName='Juan’;
```

Be careful when updating records. If you omit the WHERE clause, ALL records will be updated!

DATA MANIPULATION LANGUAGE (DML):

The SQL DELETE Statement

The DELETE statement is used to delete existing records in a table.

```
DELETE FROM table_name WHERE condition;

DELETE FROM Customers WHERE CustomerName='Alfreds Futterkiste';

DELETE FROM Customers;
```

DATA QUERY LANGUAGE (DQL)

SQL SELECT Statement

The SELECT statement is used to select data from a database.
The data returned is stored in a result table, called the result-set.

```
SELECT * FROM table_name;

SELECT column1, column2… FROM table_name;

SELECT * FROM Customers;

SELECT CustomerName, City FROM Customers;
```

```
SELECT DISTINCT column1, column2, ...
FROM table_name;

SELECT DISTINCT Country FROM Customers;

SELECT TOP 3 * FROM Customers;
```

DATA QUERY LANGUAGE (DQL)

SQL WHERE Clause

The WHERE clause is used to filter records.
The WHERE clause is used to extract only those records that fulfill a specified condition.

```
SELECT column1, column2, ...
FROM table_name
WHERE condition;
```

```
SELECT * FROM Customers
WHERE Country='Mexico';
```

Text Fields vs. Numeric Fields

```
SELECT * FROM Customers
WHERE CustomerID=1;
```

DATA QUERY LANGUAGE (DQL)

Operator	Description
=	Equal
>	Greater than
<	Less than
>=	Greater than or equal
<=	Less than or equal
<>	Not equal. Note: In some versions of SQL this operator may be written as !=
BETWEEN	Between a certain range
LIKE	Search for a pattern
IN	To specify multiple possible values for a column

DATA QUERY LANGUAGE (DQL)

```
SELECT * FROM Products
WHERE Price = 18;

SELECT * FROM Products
WHERE Price > 30;

SELECT * FROM Products
WHERE Price < 30;

SELECT * FROM Products
WHERE Price >= 30;

SELECT * FROM Products
WHERE Price <= 30;
```

```
SELECT * FROM Products
WHERE Price <> 18;

SELECT * FROM Products
WHERE Price BETWEEN 50 AND 60;
```

DATA QUERY LANGUAGE (DQL)

```
SELECT column1, column2, ...
FROM table_name
WHERE columnN LIKE pattern;
```

SQL LIKE Operator

The LIKE operator is used in a WHERE clause to search for a specified pattern in a column.
There are two wildcards often used in conjunction with the LIKE operator:

% - The percent sign represents zero, one, or multiple characters

_ - The underscore represents a single character

LIKE Operator	Description
WHERE CustomerName LIKE 'a%'	Finds any values that start with "a"
WHERE CustomerName LIKE '%a'	Finds any values that end with "a"
WHERE CustomerName LIKE '%or%'	Finds any values that have "or" in any position
WHERE CustomerName LIKE '_r%'	Finds any values that have "r" in the second position
WHERE CustomerName LIKE 'a__%'	Finds any values that start with "a" and are at least 3 characters in length
WHERE ContactName LIKE 'a%o'	Finds any values that start with "a" and ends with "o"

SQL WILDCARDS

A wildcard character is used to substitute one or more characters in a string.

Wildcard characters are used with the SQL LIKE operator. The LIKE operator is used in a WHERE clause to search for a specified pattern in a column.

Symbol	Description	Example
%	Represents zero or more characters	bl% finds bl, black, blue, and blob
_	Represents a single character	h_t finds hot, hat, and hit
[]	Represents any single character within the brackets	h[oa]t finds hot and hat, but not hit
^	Represents any character not in the brackets	h[^oa]t finds hit, but not hot and hat
-	Represents a range of characters	c[a-b]t finds cat and cbt

SQL WILDCARDS

```
SELECT * FROM Customers
WHERE City LIKE 'ber%';
```

```
SELECT * FROM Customers
WHERE City LIKE '%es%';
```

```
SELECT * FROM Customers
WHERE City LIKE '_ondon';
```

```
SELECT * FROM Customers
WHERE City LIKE 'L_n_on';
```

```
SELECT * FROM Customers
WHERE City LIKE '[bsp]%';
```

```
SELECT * FROM Customers
WHERE City LIKE '[a-c]%';
```

```
SELECT * FROM Customers
WHERE City LIKE '[!bsp]%';
```

```
SELECT * FROM Customers
WHERE City NOT LIKE '[bsp]%';
```

SQL IN OPERATOR

The IN operator allows you to specify multiple values in a WHERE clause.
The IN operator is a shorthand for multiple OR conditions.

```
SELECT column_name(s)
FROM table_name
WHERE column_name IN (value1, value2, ...);
```

```
SELECT column_name(s)
FROM table_name
WHERE column_name IN (SELECT STATEMENT);
```

```
SELECT * FROM Customers
WHERE Country IN ('Germany', 'France', 'UK');
```

```
SELECT * FROM Customers
WHERE Country NOT IN ('Germany', 'France', 'UK');
```

```
SELECT * FROM Customers
WHERE Country IN (SELECT Country FROM Suppliers
);
```

SQL BETWEEN OPERATOR

The BETWEEN operator selects values within a given range. The values can be numbers, text, or dates.

The BETWEEN operator is inclusive: begin and end values are included.

```
SELECT column_name(s)
FROM table_name
WHERE column_name BETWEEN value1 AND value2;
```

```
SELECT * FROM Products
WHERE Price BETWEEN 10 AND 20
AND NOT CategoryID IN (1,2,3);
```

```
SELECT * FROM Products
WHERE Price BETWEEN 10 AND 20;
```

```
SELECT * FROM Products
WHERE Price NOT BETWEEN 10 AND 20;
```

```
SELECT * FROM Orders
WHERE OrderDate BETWEEN '1996-07-01' AND '1996-07-31';
```

SQL ALIASES

SQL aliases are used to give a table, or a column in a table, a temporary name.

Aliases are often used to make column names more readable.

An alias only exists for the duration of the query.

Alias Column Syntax

```
SELECT column_name AS alias_name
FROM table_name;
```

Alias Table Syntax

```
SELECT column_name(s)
FROM table_name AS alias_name;
```

```
SELECT CustomerID AS ID,
CustomerName AS Customer
FROM Customers;
```

```
SELECT CustomerName AS Customer,
ContactName AS [Contact Person]
FROM Customers;
```

SQL ORDER BY KEYWORD

```
SELECT column1, column2, ...
FROM table_name
ORDER BY column1, column2, ... ASC|DESC;
```

```
SELECT * FROM Customers
ORDER BY Country;
```

```
SELECT * FROM Customers
ORDER BY Country, CustomerName;
```

```
SELECT * FROM Customers
ORDER BY Country ASC, CustomerName DESC;
```

SQL AGGREGATION FUNCTIONS

The COUNT() function returns the number of rows that matches a specified criteria.
The AVG() function returns the average value of a numeric column.
The SUM() function returns the total sum of a numeric column.
The MIN() function returns the smallest value of the selected column.
The MAX() function returns the largest value of the selected column.

```
SELECT MIN(column_name)
FROM table_name
WHERE condition;
```

```
SELECT COUNT(column_name)
FROM table_name
WHERE condition;
```

```
SELECT SUM(column_name)
FROM table_name
WHERE condition;
```

```
SELECT MAX(column_name)
FROM table_name
WHERE condition;
```

```
SELECT AVG(column_name)
FROM table_name
WHERE condition;
```

SQL GROUP BY STATEMENT

The GROUP BY statement groups rows that have the same values into summary rows, like "find the number of customers in each country".

The GROUP BY statement is often used with aggregate functions (COUNT, MAX, MIN, SUM, AVG) to group the result-set by one or more columns.

```
SELECT column_name(s)
FROM table_name
WHERE condition
GROUP BY column_name(s)
ORDER BY column_name(s);
```

```
SELECT COUNT(CustomerID), Country
FROM Customers
GROUP BY Country;
```

SQL HAVING CLAUSE

The HAVING clause was added to SQL because the WHERE keyword could not be used with aggregate functions.

```
SELECT column_name(s)
FROM table_name
WHERE condition
GROUP BY column_name(s)
HAVING condition
ORDER BY column_name(s);
```

```
SELECT COUNT(CustomerID), Country
FROM Customers
GROUP BY Country
HAVING COUNT(CustomerID) > 5;
```

SQL SELECT INTO STATEMENT

```
SELECT * INTO CustomersBackup2017
FROM Customers;

SELECT * INTO CustomersGermany
FROM Customers
WHERE Country = 'Germany';

INSERT INTO Customers (CustomerName, City, Country)
SELECT SupplierName, City, Country FROM Suppliers;

INSERT INTO Customers (CustomerName, ContactName, Address, City, PostalCode, Country)
SELECT SupplierName, ContactName, Address, City, PostalCode, Country FROM Suppliers;

INSERT INTO Customers (CustomerName, City, Country)
SELECT SupplierName, City, Country FROM Suppliers
WHERE Country='Germany';
```

SQL CASE STATEMENT

The CASE statement goes through conditions and returns a value when the first condition is met (like an IF-THEN-ELSE statement). So, once a condition is true, it will stop reading and return the result. If no conditions are true, it returns the value in the ELSE clause.

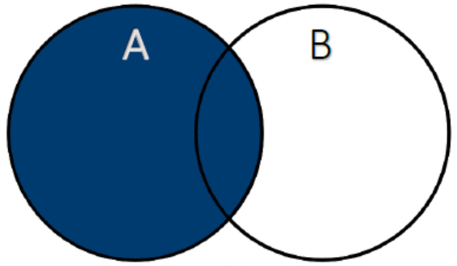
If there is no ELSE part and no conditions are true, it returns NULL.

```
CASE
    WHEN condition1 THEN result1
    WHEN condition2 THEN result2
    WHEN conditionN THEN resultN
    ELSE result
END;
```

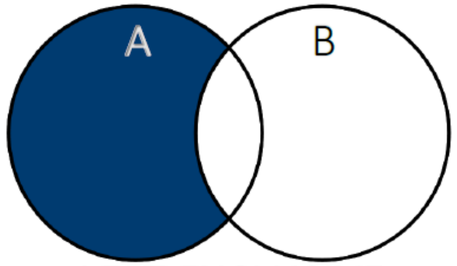
```
SELECT OrderID, Quantity,
CASE
    WHEN Quantity > 30 THEN "The quantity is
greater than 30"
    WHEN Quantity = 30 THEN "The quantity is
30"
    ELSE "The quantity is under 30"
END AS QuantityText
FROM OrderDetails;
```

SQL JOINS

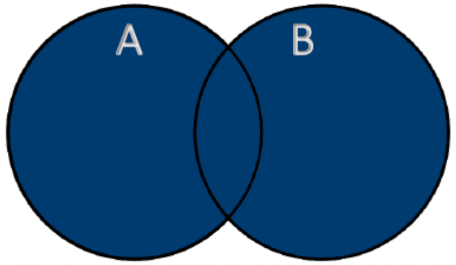
A JOIN clause is used to combine rows from two or more tables, based on a related column between them.



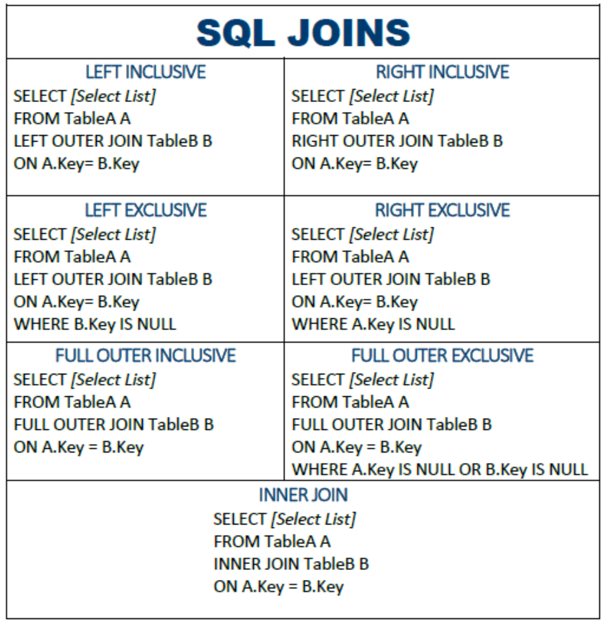
LEFT INCLUSIVE



LEFT EXCLUSIVE



FULL OUTER INCLUSIVE



SQL JOINS

LEFT INCLUSIVE

```
SELECT [Select List]
FROM TableA A
LEFT OUTER JOIN TableB B
ON A.Key= B.Key
```

RIGHT INCLUSIVE

```
SELECT [Select List]
FROM TableA A
RIGHT OUTER JOIN TableB B
ON A.Key= B.Key
```

LEFT EXCLUSIVE

```
SELECT [Select List]
FROM TableA A
LEFT OUTER JOIN TableB B
ON A.Key= B.Key
WHERE B.Key IS NULL
```

RIGHT EXCLUSIVE

```
SELECT [Select List]
FROM TableA A
LEFT OUTER JOIN TableB B
ON A.Key= B.Key
WHERE A.Key IS NULL
```

FULL OUTER INCLUSIVE

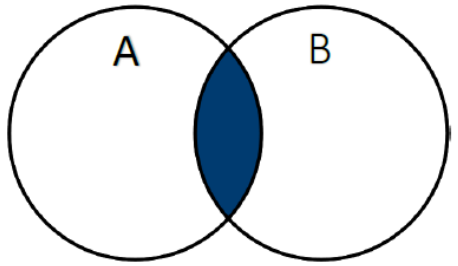
```
SELECT [Select List]
FROM TableA A
FULL OUTER JOIN TableB B
ON A.Key = B.Key
```

FULL OUTER EXCLUSIVE

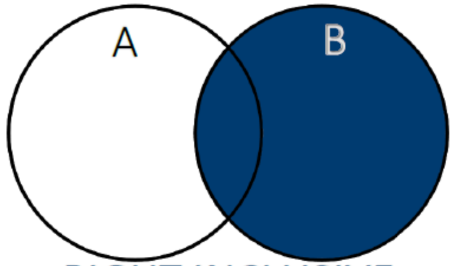
```
SELECT [Select List]
FROM TableA A
FULL OUTER JOIN TableB B
ON A.Key = B.Key
WHERE A.Key IS NULL OR B.Key IS NULL
```

INNER JOIN

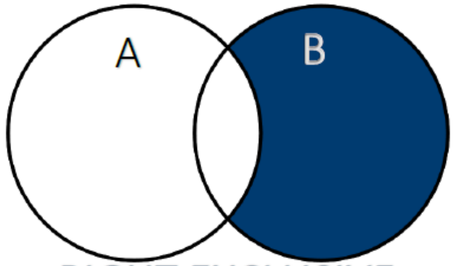
```
SELECT [Select List]
FROM TableA A
INNER JOIN TableB B
ON A.Key = B.Key
```



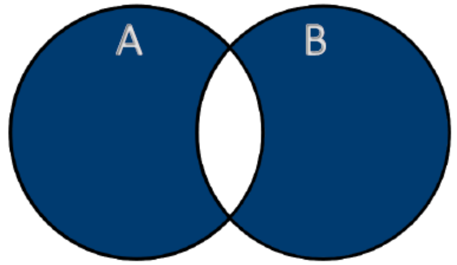
INNER JOIN



RIGHT INCLUSIVE



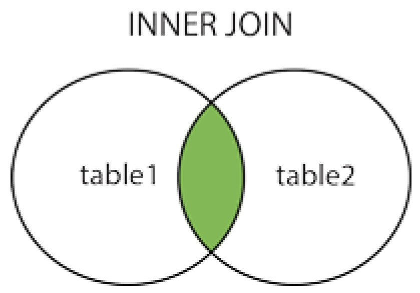
RIGHT EXCLUSIVE



FULL OUTER EXCLUSIVE

SQL INNER JOIN

The INNER JOIN keyword selects records that have matching values in both tables.

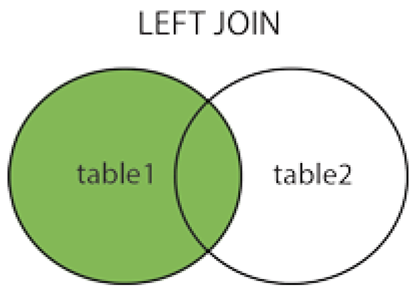


```
SELECT Orders.OrderID, Customers.CustomerName
FROM Orders
INNER JOIN Customers ON Orders.CustomerID = Customers.CustomerID;

SELECT Orders.OrderID, Customers.CustomerName, Shippers.ShipperName
FROM ((Orders
INNER JOIN Customers ON Orders.CustomerID = Customers.CustomerID)
INNER JOIN Shippers ON Orders.ShipperID = Shippers.ShipperID);
```

SQL LEFT JOINS

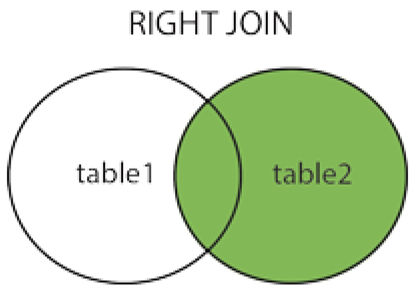
The LEFT JOIN keyword returns all records from the left table (table1), and the matched records from the right table (table2). The result is NULL from the right side, if there is no match.



```
SELECT Customers.CustomerName, Orders.OrderID
FROM Customers
LEFT JOIN Orders ON Customers.CustomerID = Orders.CustomerID
ORDER BY Customers.CustomerName;
```

SQL RIGHT JOINS

The RIGHT JOIN keyword returns all records from the right table (table2), and the matched records from the left table (table1). The result is NULL from the left side, when there is no match.



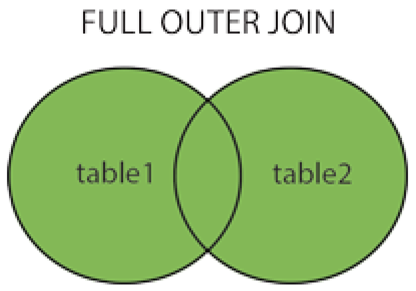
```
SELECT Orders.OrderID, Employees.LastName, Employees.FirstName
FROM Orders
RIGHT JOIN Employees ON Orders.EmployeeID = Employees.EmployeeID
ORDER BY Orders.OrderID;
```

SQL FULL JOINS

The FULL OUTER JOIN keyword return all records when there is a match in left (table1) or right (table2) table records.

Note: FULL OUTER JOIN can potentially return very large result-sets!

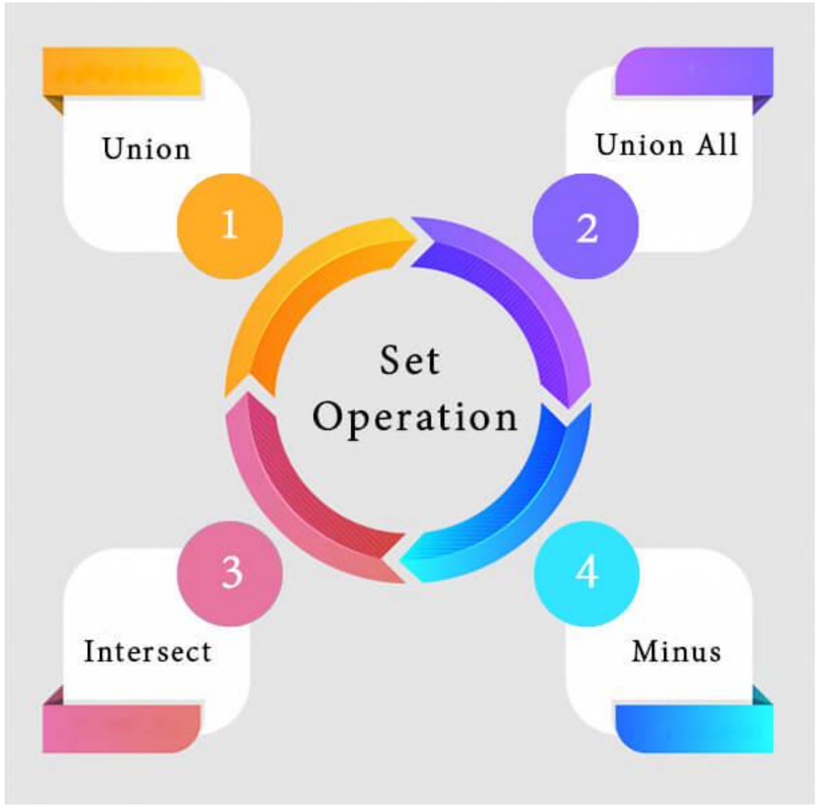
Tip: FULL OUTER JOIN and FULL JOIN are the same.



```sql
SELECT Customers.CustomerName, Orders.OrderID
FROM Customers
FULL OUTER JOIN Orders ON Customers.CustomerID=Orders.CustomerID
ORDER BY Customers.CustomerName;
```

SQL SET OPERATORS

A JOIN clause is used to combine rows from two or more tables, based on a related column between them.



Minus and Except are synonyms

SQL SET OPERATORS

A JOIN clause is used to combine rows from two or more tables, based on a related column between them.

```
SELECT City FROM Customers
UNION
SELECT City FROM Suppliers
;
```

```
SELECT City FROM Customers
UNION ALL
SELECT City FROM Suppliers
;
```

```
SELECT City FROM Customers
Intersect
SELECT City FROM Suppliers
;
```

```
SELECT City FROM Customers
Except
SELECT City FROM Suppliers
;
```

Where and Order by can be used in SET operators

Thank You!