DATABASE DEVELOPMENT

SFDDD501

Develop a database

Competence

REQF Level: 5

Credits: 12

Learning hours

120

Sector: ICT

Sub-sector: Software Development

Issue date:December 2018

DATABASE DEVELOPMENT HANDOUT

Unit 1: DATABASE STRUCTURE

1.1 Introduction to SQL

1.1.1 SQL description

SQL stands for “Structured Query Language”. It is a query language used for accessing and modifying information in the database.

IBM first developed SQL in 1970s. Also it is an ANSI/ISO standard. It has become a Standard Universal Language used by most of the relational database management systems (RDBMS). Some of the RDBMS systems are: Oracle, Microsoft SQL server, Sybase etc.

Commands in SQL: DDL, you always work with SQL. One common method used to categorized SQL statements is to divide them according to the functions they perform. Based on this method, SQL can be separated into three types of statements:

Data Definition Language (DDL) DDL statements are used to create, modify, or delete database object such as tables, views, schemas, domains, triggers, and stored procedures. The SQL keywords most often associated with DDL statements are: CREATE, ALTER, RENAME and DROP.

Data Control Language (DCL) statement allows you to control who has access to specific object in your database. With the DCL statements, you can grant or restrict access by using the GRANT or REVOKE statements.

Data Manipulation Language DML statements are used to view, add, modify, or delete data stored in your database objects. The primary keywords associated with DML statement are SELECT, INSERT, UPDATE, and DELETE

1.1.2 SQL SUB-LANGUAGES

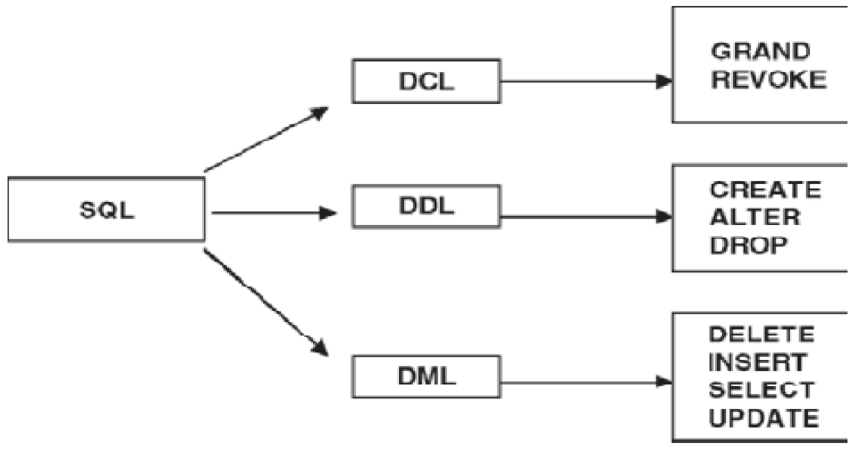
If you work with database, you always work with SQL. One common method used to categorized SQL statements is to divide them according to the functions they perform. Based on this method, SQL can be separated into three types of statements:

Data Definition Language (DDL) DDL statements are used to create, modify, or delete database object such as tables, views, schemas, domains, triggers, and stored procedures. The SQL keywords most often associated with DDL statements are: CREATE, ALTER, DROP, RENAME,and TRUNCATE.

Data Control Language (DCL) statement allows you to control who has access to specific object in your database. With the DCL statements, you can grant or restrict access by using the GRANT or REVOKE statements.

Data Manipulation Language DML statements are used to view, add, modify, or delete data stored in your database objects. The primary keywords associated with DML statement are SELECT, INSERT, UPDATE, and DELETE

The classification of commands in SQL is shown below:



1.1.3 Description of SQL commands per sublanguage

SQL SUB-LANGUAGE	**COMMAND DESCRIPTION**
DDL COMMANDS	**CREATE:** is used to create the database or its objects (like table, index, function, views, store procedure and triggers**).** **ALTER:** is used to alter the structure of the database. **DROP:** is used to delete objects from the database. **RENAME:** is used to rename an object existing in the database. **TRUNCATE:** is used to remove all records from a table, including all spaces allocated for the records are removed.
DCL COMMANDS	**GRANT:** gives user's access privileges to database. **REVOKE:** withdraw user's access privileges given by using the GRANT command.
DML COMMANDS	**INSERT:** is used to insert data into a table. **UPDATE:** is used to update existing data within a table. **DELETE:** is used to delete records from a database table. **SELECT:** is used to retrieve data from the a database.

N.B: TCL **is also SQL sublanguage but is advanced.** TCL stands for transaction Control Language. TCL commands deals with the transaction within the database.
Examples of TCL commands:

- COMMIT– commits a Transaction.
- ROLLBACK– rollbacks a transaction in case of any error occurs.
- SAVEPOINT–sets a savepoint within a transaction.
- SET TRANSACTION–specifies characteristics for the transaction.

1.2 CREATE A DATABASE

Command	**Syntaxes**	**Examples**
CREATE database	CREATE database *databasename*;	CREATE database sellingDB;
RENAME database	RENAME database old_db_name **TO** new_db_name;	RENAME database sellingDB **TO** selling;
DROP database	DROP database databasename	DROP database selling

N.B: RENAME DATABASE has since been removed from all newer versions of MySQL to avoid security risks.

1.3 CREATE TABLES WITH ATTRIBUTES

1.3.1 SQL Data Types

Each column in a database table is required to have a name and a data type.
An SQL developer must decide what type of data that will be stored inside each column when creating a table.
Note: Data types might have different names in different database. And even if the name is the same, the size and other details may be different! **Always check the documentation!** Data types provided here are for MySQL DBMS.

Text data types:

Data type	**Description**
CHAR(size)	Holds a fixed length string (can contain letters, numbers, and special characters). The fixed size is specified in parenthesis. Can store up to 255 characters
VARCHAR(size)	Holds a variable length string (can contain letters, numbers, and special characters). The maximum size is specified in parenthesis. Can store up to 255 characters. **Note:** If you put a greater value than 255 it will be converted to a TEXT type
TINYTEXT	Holds a string with a maximum length of 255 characters
TEXT	Holds a string with a maximum length of 65,535 characters
BLOB	For BLOBs (Binary Large OBjects). Holds up to 65,535 bytes of data
MEDIUMTEXT	Holds a string with a maximum length of 16,777,215 characters
MEDIUMBLOB	For BLOBs (Binary Large OBjects). Holds up to 16,777,215 bytes of data
LONGTEXT	Holds a string with a maximum length of 4,294,967,295 characters
LONGBLOB	For BLOBs (Binary Large OBjects). Holds up to 4,294,967,295 bytes of data
ENUM(x,y,z,etc.)	Let you enter a list of possible values. You can list up to 65535 values in an ENUM list. If a value is inserted that is not in the list, a blank value will be inserted. **Note:** The values are sorted in the order you enter them. You enter the possible values in this format: ENUM('X','Y','Z')
SET	Similar to ENUM except that SET may contain up to 64 list items and can store more than one choice

Number data types:

Data type	**Description**
TINYINT(size)	-128 to 127 normal. 0 to 255 UNSIGNED*. The maximum number of digits may be specified in parenthesis
SMALLINT(size)	-32768 to 32767 normal. 0 to 65535 UNSIGNED*. The maximum number of digits may be specified in parenthesis
MEDIUMINT(size)	-8388608 to 8388607 normal. 0 to 16777215 UNSIGNED*. The maximum number of digits may be specified in parenthesis
INT(size)	-2147483648 to 2147483647 normal. 0 to 4294967295 UNSIGNED*. The maximum number of digits may be specified in parenthesis
BIGINT(size)	-9223372036854775808 to 9223372036854775807 normal. 0 to 18446744073709551615 UNSIGNED*. The maximum number of digits may be specified in parenthesis
FLOAT(size,d)	A small number with a floating decimal point. The maximum number of digits may be specified in the size parameter. The maximum number of digits to the right of the decimal point is specified in the d parameter

DOUBLE(size,d)	A large number with a floating decimal point. The maximum number of digits may be specified in the size parameter. The maximum number of digits to the right of the decimal point is specified in the d parameter
DECIMAL(size,d)	A DOUBLE stored as a string , allowing for a fixed decimal point. The maximum number of digits may be specified in the size parameter. The maximum number of digits to the right of the decimal point is specified in the d parameter

*The integer types have an extra option called UNSIGNED. Normally, the integer goes from an negative to positive value. Adding the UNSIGNED attribute will move that range up so it starts at zero instead of a negative number.

Date data types:

Data type	**Description**
DATE()	A date. Format: YYYY-MM-DD **Note:** The supported range is from '1000-01-01' to '9999-12-31'
DATETIME()	A date and time combination. Format: YYYY-MM-DD HH:MI:SS **Note:** The supported range is from '1000-01-01 00:00:00' to '9999-12-31 23:59:59'

TIMESTAMP()	A timestamp. TIMESTAMP values are stored as the number of seconds since the Unix epoch ('1970-01-01 00:00:00' UTC). Format: YYYY-MM-DD HH:MI:SS **Note:** The supported range is from '1970-01-01 00:00:01' UTC to '2038-01-09 03:14:07' UTC
TIME()	A time. Format: HH:MI:SS **Note:** The supported range is from '-838:59:59' to '838:59:59'
YEAR()	A year in two-digit or four-digit format. **Note:** Values allowed in four-digit format: 1901 to 2155. Values allowed in two-digit format: 70 to 69, representing years from 1970 to 2069

Even if DATETIME and TIMESTAMP return the same format, they work very differently. In an INSERT or UPDATE query, the TIMESTAMP automatically set itself to the current date and time. TIMESTAMP also accepts various formats, like YYYYMMDDHHMISS, YYMMDDHHMISS, YYYYMMDD, or YYMMDD.

1.3.2 Table definition syntaxes

Command	**Syntax**	**Description**

Create table	CREATE TABLE *table_name* (
	    column1 datatype,
	    column2 datatype,
	    column3 datatype,
	   ....
	);
		The CREATE TABLE statement is used to create a new table in a database

	CREATE TABLE *new_table_name* AS
	    SELECT *column1*, *column2,...*
	    FROM *existing_table_name*
	    WHERE ....;
		The new table gets the same column definitions. The new table will be filled with the existing values from the old table.

Alter table: add column	ALTER TABLE *table_name* ADD *column_name datatype*;	The ALTER TABLE statement is used to add, delete, or modify columns in an existing table.

		The ALTER TABLE statement is also used to add and drop various constraints on an existing table. ALTER TABLE - ADD Column is used to add a column in a table
Alter table: drop column	ALTER TABLE *table_name* DROP COLUMN *column_name*;	It is used to delete a column in a table(notice that some database systems don't allow deleting a column)
Alter table: modify column data type	ALTER TABLE *table_name* MODIFY COLUMN *column_name datatype*;	It is used to change the data type of a column in a table
Rename table	**ALTER TABLE** table_name **RENAME** TO new_table_name	**RENAME TABLE** syntax is used to **change** the **name** of a **table**. Sometimes, we choose non-meaningful **name** for the **table**. So it is required to be changed.
Drop table	DROP TABLE *table_name*;	DROP TABLE statement is used to drop an existing table in a database
Truncate table	TRUNCATE TABLE *table_name*;	TRUNCATE TABLE statement is used to delete the data inside a table, but not the table itself.

1.3.3 Table definition Examples

1.3.3.1 Create Table Example

The following example creates a table called "Persons" that contains five columns: PersonID, LastName, FirstName, Address, and City:

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

1.3.3.2 Create Table Using another Table Example

A copy of an existing table can also be created using CREATE TABLE.

The new table gets the same column definitions. All columns or specific columns can be selected.

If you create a new table using an existing table, the new table will be filled with the existing values from the old table.

The following SQL creates a new table called "TestTables" (which is a copy of the " Persons " table):

```
CREATE TABLE TestTable AS
SELECT LastName, FirstName
FROM Persons;
```

1.3.3.3 SQL ALTER TABLE Statement examples

Look at the "Persons" table:

ID	LastName	FirstName	Address	City
1	Hansen	Ola	Timoteivn 10	Sandnes
2	Svendson	Tove	Borgvn 23	Sandnes
3	Pettersen	Kari	Storgt 20	Stavanger

- **ADD A NEW COLUMN Example**

Now we want to add a column named "DateOfBirth" in the "Persons" table.

We use the following SQL statement:

ALTER TABLE Persons ADD DateOfBirth date;

Notice that the new column, "DateOfBirth", is of type date and is going to hold a date. The data type specifies what type of data the column can hold.

The "Persons" table will now look like this:

ID	LastName	FirstName	Address	City	DateOfBirth
1	Hansen	Ola	Timoteivn 10	Sandnes	
2	Svendson	Tove	Borgvn 23	Sandnes	
3	Pettersen	Kari	Storgt 20	Stavanger	

- **CHANGE DATA TYPE Example**

Now we want to change the data type of the column named "DateOfBirth" in the "Persons" table.

We use the following SQL statement:

ALTER TABLE Persons ALTER COLUMN DateOfBirth year;

Notice that the "DateOfBirth" column is now of type year and is going to hold a year in a two- or four-digit format.

- **DROP COLUMN Example**

Next, we want to delete the column named "DateOfBirth" in the "Persons" table. We use the following SQL statement:

ALTER TABLE Persons DROP COLUMN DateOfBirth;

The "Persons" table will now look like this:

ID	LastName	FirstName	Address	City
1	Hansen	Ola	Timoteivn 10	Sandnes
2	Svendson	Tove	Borgvn 23	Sandnes
3	Pettersen	Kari	Storgt 20	Stavanger

1.3.3.4 SQL DROP TABLE Statement example

Note: Be careful before dropping a table. Deleting a table will result in loss of complete information stored in the table!

DROP TABLE Shippers;

1.3.3.5 Rename Table example

You should use any one of the following syntax to RENAME the table name:

ALTER TABLE persons RENAME TO people;

After that the table “persons” will be changed into table name "people".

1.4 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.

Syntax:

```
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.

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

1.4.1 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.

The following SQL ensures that the "ID", "LastName", and "FirstName" columns will NOT accept NULL values:

Example

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

N.B: If the table has already been created, you can add a NOT NULL constraint to a column with the **ALTER TABLE** statement

1.4.2 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.

1.4.2.1 Add a UNIQUE Constraint

SQL UNIQUE Constraint on CREATE TABLE

The following SQL creates a UNIQUE constraint on the "ID" column when the "Persons" table is created:

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

To name a UNIQUE constraint, and to define a UNIQUE constraint on multiple columns, use the following SQL syntax:

```
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

To create a UNIQUE constraint on the "ID" column when the table is already created, use the following SQL:

```
ALTER TABLE Persons ADD UNIQUE (ID);
```

To name a UNIQUE constraint, and to define a UNIQUE constraint on multiple columns, use the following SQL syntax:

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

1.4.2.2 DROP a UNIQUE Constraint

To drop a UNIQUE constraint, use the following SQL:

```
ALTER TABLE Persons DROP CONSTRAINT UC_Person;
```

1.4.3 SQL PRIMARY KEY Constraint

The PRIMARY KEY constraint uniquely identifies each record in a database table.

Primary keys must contain UNIQUE values, and cannot contain NULL values.

A table can have only one primary key, which may consist of single or multiple fields.

1.4.3.1 ADD a PRIMARY KEY Constraint

SQL PRIMARY KEY on CREATE TABLE

The following SQL creates a PRIMARY KEY on the "ID" column when the "Persons" table is created:

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

To allow naming of a PRIMARY KEY constraint, and for defining a PRIMARY KEY constraint on multiple columns, use the following SQL syntax:

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

Note: In the example above there is only ONE PRIMARY KEY (PK_Person). However, the VALUE of the primary key is made up of TWO COLUMNS (ID + LastName).

SQL PRIMARY KEY on ALTER TABLE

To create a PRIMARY KEY constraint on the "ID" column when the table is already created, use the following SQL:

```
ALTER TABLE Persons ADD PRIMARY KEY (ID);
```

To allow naming of a PRIMARY KEY constraint, and for defining a PRIMARY KEY constraint on multiple columns, use the following SQL syntax:

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

Note: If you use the ALTER TABLE statement to add a primary key, the primary key column(s) must already have been declared to not contain NULL values (when the table was first created).

1.4.3.2 DROP a PRIMARY KEY Constraint

To drop a PRIMARY KEY constraint, use the following SQL:

```
ALTER TABLE Persons DROP PRIMARY KEY;
```

1.4.4 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.

Look at the following two tables:

"Persons" table:

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

"Orders" table:

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

Notice that the "PersonID" column in the "Orders" table points to the "PersonID" column in the "Persons" table.

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.

1.4.4.1 ADD a FOREIGN KEY Constraint

SQL FOREIGN KEY on CREATE TABLE

The following SQL creates a FOREIGN KEY on the "PersonID" column when the "Orders" table is created:

```
CREATE TABLE Orders (
   OrderID int NOT NULL,
   OrderNumber int NOT NULL,
   PersonID int,
   PRIMARY KEY (OrderID),
   FOREIGN KEY (PersonID) REFERENCES Persons(PersonID)
);
```

SQL FOREIGN KEY on ALTER TABLE

To create a FOREIGN KEY constraint on the "PersonID" column when the "Orders" table is already created, use the following SQL:

```
ALTER TABLE Orders ADD FOREIGN KEY (PersonID) REFERENCES Persons (PersonID);
```

1.4.4.2 DROP a FOREIGN KEY Constraint

To drop a FOREIGN KEY constraint, use the following SQL:

```
ALTER TABLE Orders DROP FOREIGN KEY FK_PersonOrder;
```

1.4.5 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.

1.4.5.1 ADD CHECK constraint

SQL CHECK on CREATE TABLE

The following SQL creates a CHECK constraint on the "Age" column when the "Persons" table is created. The CHECK constraint ensures that you can not have any person below 18 years:

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

To allow naming of a CHECK constraint, and for defining a CHECK constraint on multiple columns, use the following SQL syntax:

```
CREATE TABLE Persons (
   ID int NOT NULL,
   LastName varchar(255) NOT NULL,
   FirstName varchar(255),
   Age int,
   City varchar(255),
   CONSTRAINT CHK_Person CHECK (Age>=18 AND City='Sandnes')
);
```

SQL CHECK on ALTER TABLE

To create a CHECK constraint on the "Age" column when the table is already created, use the following SQL:

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

1.4.5.2 DROP CHECK constraint

To drop a CHECK constraint, use the following SQL:

```
ALTER TABLE Persons DROP CHECK CHK_PersonAge;
```

1.4.6 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.

1.4.6.1 ADD DEFAULT constraint

SQL DEFAULT on CREATE TABLE

The following SQL sets a DEFAULT value for the "City" column when the "Persons" table is created:

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

The DEFAULT constraint can also be used to insert system values, by using functions like GETDATE():

```
CREATE TABLE Orders (
   ID int NOT NULL,
   OrderNumber int NOT NULL,
   OrderDate date DEFAULT GETDATE()
);
```

SQL DEFAULT on ALTER TABLE

To create a DEFAULT constraint on the "City" column when the table is already created, use the following SQL:

```
ALTER TABLE Persons ALTER City SET DEFAULT 'Sandnes';
```

1.4.6.2 DROP DEFAULT constraint

To drop a DEFAULT constraint, use the following SQL:

ALTER TABLE Persons ALTER City DROP DEFAULT;

1.5: Database design refine

1.5.0 Introduction

Database design is the organization of data according to a database model. The designer determines what data must be stored and how the data elements interrelate.

What is good database design?

Certain principles guide the database design process:

- The first principle is that duplicate information (also called redundant data) is bad, because it wastes space and increases the likelihood of errors and inconsistencies.
- The second principle is that the correctness and completeness of information is important. If your database contains incorrect information, any reports that pull information from the database will also contain incorrect information. As a result, any decisions you make that are based on those reports will then be misinformed.

A good database design is, therefore, one that:

- Divides your information into subject-based tables to reduce redundant data.
- Provides Access with the information it requires to join the information in the tables together as needed.
- Helps support and ensure the accuracy and integrity of your information.
- Accommodates your data processing and reporting needs.

Refining DB design

Refine a DB design is to analyze it design for errors.

Once you have the tables, fields, and relationships you need, you should create and populate your tables with sample data and try working with the information: creating queries, adding new records, and so on. Doing this helps highlight potential problems — for example, you might need to add a column that you forgot to insert during your design phase, or you may have a table that you should split into two tables to remove duplication.

1.5.1 Evaluating the Database Design

Evaluation considerations
Before you refine the data structure diagram, you need to evaluate the design for performance. To satisfy performance requirements for each individual business transaction, you need to consider the following issues:

- **Input/output (I/O) performance**

  -- Is the number of I/O operations performed against the database sufficiently low to provide satisfactory transaction performance?

- **CPU time**

  -- Does the structure of the physical database optimize the use of CPU processing?

- **Space management**

  -- Do design choices help to conserve storage resources?

Once you have refined the database to satisfy each individual transaction, you need to determine how the system will be affected by the concurrent execution of several transactions. To avoid excessive **conflict** for database resources, you need to make appropriate changes to the physical model.

Refining the database design
Like many other database design procedures, refining the database design is an iterative process, as shown below. As you refine the design, you need to evaluate the design for performance. When you make changes, you should review the design to ensure that it will optimize processing for all critical transactions and also minimize the likelihood of contention.

1.5.2 Refinement Options

The following database options can be used to ensure optimal performance in individual business transactions:

- **Indexes**

  -- "Determining How an Entity Should Be Stored" showed you how to include indexes in the database design to provide data clustering. At this point in the design process, you have the option to include additional indexes to provide generic search capabilities as well as alternate access keys.

- **Collapsing relationships**

  -- A one-to-many relationship can be expressed within a single entity by making the many portion of the relationship a repeating data element. A one-to-many relationship expressed in this way can enhance processing performance by reducing DBMS overhead associated with processing multiple entity occurrences.

- **Introducing redundancy**

  -- By maintaining certain data redundantly, you can sometimes enhance processing efficiency in selected applications.

Each of these options is described in detail below following a discussion of how to estimate I/Os for transactions.

1.5.3 Estimating I/Os for Transactions

After you have assigned data location and access modes to the entities in a database, you need to estimate the number of input/output operations that each business transaction will perform. You estimate the I/O count for a transaction by tracing the flow of processing from one entity to another in the database. As you trace the flow of processing, you determine the number of I/Os required to access all necessary entities.
The I/O estimate for a business transaction depends on several factors, including:

- The order in which entities are accessed
- The location mode of each entity accessed
- The types of indexes (if any) used to access the data
- How the entities are clustered in the database?

General guidelines
Assuming that an entire cluster of database entities can fit on a single database page, you can use the following general guidelines for estimating I/Os:

- Zero I/Os are required to access an entity that is clustered around a previously accessed entity.
- One I/O is required to access an entity stored CALC.
- Three I/Os are required to access an entity through an index.

To calculate the time required to perform all I/O operations in a particular transaction, perform the following computations:

- **Total number of I/Os for all entity types**

  -- Compute the total number of I/O operations by adding the number of I/Os required to retrieve and update occurrences of all entity types.

- **I/O reserve factor**

  -- Multiply the total number of I/Os by 1.5 to account for possible overflow conditions and large index structures.

- **Amount of time to perform I/Os**

  -- Multiply the total number of I/Os for all entity types by the access time for the device being used. The result is a rough estimate of the time required to perform all I/O operations in the transaction.

Once you have determined how much time will be required to execute a particular transaction, you need to compare this time figure with the performance goal you established earlier in the design process. If the required time does not meet your expectations, you need to modify the physical database model until it does. Sometimes you have to change your expectations.

1.5.4 Eliminating Unnecessary Entities

Sometimes entities identified during the logical design are not required as separate entities in the physical implementation. Two ways to eliminate such entities are:

- Collapsing relationships
- Introducing redundancy

1.5.4.1 Collapsing Relationships

During the normalization process in logical database design, you separated multiply-occurring data into a separate entity type (first normal form). It may be more efficient to move this data back into the original (parent) entity.

Consider this option if data occurs a fixed number of times and the data is not related to another entity. An example of such data is monthly sales totals for the last twelve months collapsed into a sales entity.

Advantages
By maintaining the data in a single entity instead of maintaining two separate entity types, you can:

- Save storage space that might otherwise be used for pointers or foreign-key data.
- Reduce database overhead by eliminating the need to retrieve two entities. When you express a one-to-many relationship within a single entity, application programs can access all desired data with a single DBMS access.

Note:

Expressing a one-to-many relationship within a single entity offers little I/O performance advantage over clustering two separate entities.

Comparison of collapsing relationships and maintaining separate entities
The following table presents a comparison of collapsing relationships into a single entity type and maintaining separate entities.

Efficiency Considerations	**Potential Impact**
I/O	Expressing a one-to-many relationship within a single entity offers little I/O performance advantage over clustering two entities.
CPU time	By storing a repeating element in an entity, you can reduce the amount of CPU time required to access the necessary data.
Space management	By storing a repeating element in an entity instead of maintaining two separate entity types, you can save storage space that might otherwise be required for pointers or foreign key data.
Contention	No difference

SQL considerations

Because repeating elements violate first normal form, they are incompatible with the relational model and cannot be defined in SQL. However, if there are a fixed number of repetitions (such as months in a year), the repeating elements can be separately named (such as JANUARY, FEBRUARY, and so on). If there is a variable but quite small number of occurrences (such as phone numbers), a fixed maximum number of elements can be named (PHONE1, PHONE2, for example), using the nullable attribute to allow identification of occurrences that might not have a value.

1.5.4.2 Introducing Redundancy

Although data redundancy should normally be avoided, you can sometimes enhance processing efficiency in selected applications by storing redundant information. A certain amount of planned data redundancy can be used to simplify processing logic.
In some instances, you can eliminate an entity type from the database design by maintaining some redundant information. For example, you might be able to eliminate an entity type by maintaining the information associated with this entity in another entity type in the database. When you *merge* two or more entity types in this way, you simplify the physical data structures and reduce relationship overhead.

Considerations

Consider maintaining redundant data under the following circumstances:

- **An entity type is never processed independently of other entity types**

  If an entity is always processed with one or more additional entity types, you may be able to eliminate the entity and store the information elsewhere in the database. Since the information associated with the entity is not meaningful by itself, inconsistent copies of the data should not present a problem for the business.

- **An entity type is not used as an entry point to the database**

  If application programs do not use a particular entity type as an entry point to the database, you may be able to eliminate the entity type from the design. However, do not eliminate the entity if it is a junction entity type in a many-to-many relationship.

- **The volume of data to be stored redundantly is minimal**

  Do not maintain large amounts of data redundantly. A high volume of redundant information will require excessive storage space.

1.5.5 Adding Indexes

In Determining How an Entity Should Be Stored, you included indexes in the physical database model for entities that will be accessed through multi-occurrence retrievals. These entity occurrences will be clustered around the index. You now have the option to define additional indexes for database entities to satisfy processing requirements.

What is an index?
An index is a data structure consisting of addresses (db-keys) and values from one or more data elements of a given entity. Indexes enhance processing performance by providing alternate access keys to entities.

Advantages and disadvantages
While indexes minimize the number of I/Os required to retrieve data from the database, they require extra storage space and add overhead for maintenance. The addition of an index actually *increases*
the I/Os and processing time required to add or remove an entity occurrence. You will need to weigh the options when considering the use of indexes.
Why add additional indexes?
Indexes provide a quick and efficient method for performing several types of processing.

- **Direct retrieval by key**

  -- With an index, the DBMS can retrieve individual entity occurrences directly by means of a key. For example, an application programmer could use an index to quickly access an employee by social security number.

  Because more than one index can be defined on an entity (each on a different data element), they can be used to implement multiple access keys to an entity.

- **Generic access by key**

  -- Indexes allow the DBMS to retrieve a group of entity occurrences by specifying a complete or partial (generic) key value. For example, an index could be used to quickly access all employees whose last names begin with the letter M. A string of characters, up to the length of the symbolic key, can be used as a generic key.

- **Ordered retrieval of occurrences**

  -- The DBMS can use a sorted index to retrieve entity occurrences in sorted order. In this case, the keys in the index are automatically maintained in sorted order; the entity occurrences can then be retrieved in ascending or descending sequence by key value. The application program does not have to sort the entity occurrences after retrieval. For example, all employees could be listed by name. Because entity occurrences can be accessed through more than one index, they can be retrieved in more than one sort sequence.

- **Retrieval of a small number of entity occurrences**

  -- An index improves retrieval of all occurrences of a sparsely-populated entity and provides a way of locating all occurrences of such entities without reading every page in the area (an area sweep). Area sweeps are the most efficient means of retrieving entities with occurrences on all (or almost all) pages in an area.

- **Physical sequential processing by key**

  -- Entity occurrences can be stored clustered around an index. With this storage mode, the physical location of the clustered entity occurrences reflects the ascending or descending order of their db-keys or symbolic keys. If occurrences of an entity are to be retrieved in sequential order, storing entity occurrences clustered via the index reduces I/O. This option is most effective when used with a stable database.

- **Enforcement of unique constraints**

  -- An index can be used to ensure that entity occurrences have unique values for data elements; for example, to ensure that employees are not assigned duplicate social security numbers.

  Other means of enforcing unique constraints include:

  - Using a unique CALC key
  - Using a sorted relationship

Unit 2: APPLY DML QUERIES

2.1 SQL INSERT Statement

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

2.1.1 INSERT INTO Syntax

It is possible to write the INSERT INTO statement in two ways.
The first way specifies both the column names and the values to be inserted:

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

If you are adding values for all the columns of the table, you do not need to specify the column names in the SQL query. However, make sure the order of the values is in the same order as the columns in the table. The INSERT INTO syntax would be as follows:
INSERT INTO *table_name* VALUES (*value1*, *value2*, *value3*, ...);

Example:
Below is a selection from the "Customers" table in the sample database:

Customer ID	**CustomerName**	**ContactName**	**City**	**PostalCode**	**Country**
90	Wilman Kala	Matti Karttunen	Helsinki	21240	Finland
91	Wolskii	Zbyszek	Walla	01-012	Poland

The following SQL statement inserts a new record in the "Customers" table:
INSERT INTO Customers (CustomerName, ContactName, City, PostalCode, Country)
VALUES ('Cardinal', 'Tom B. Erichsen', 'Stavanger', '4006', 'Norway');

The selection from the "Customers" table will now look like this:

Customer ID	**CustomerName**	**ContactName**	**City**	**PostalCode**	**Country**
89	White Clover Markets	Karl Jablonski	Seattle	98128	USA
90	Wilman Kala	Matti Karttunen	Helsinki	21240	Finland
91	Wolski	Zbyszek	Walla	01-012	Poland
92	Cardinal	Tom B. Erichsen	Stavanger	4006	Norwa

2.1.2 INSERT DATA ONLY IN SPECIFIED COLUMNS

It is also possible to only insert data in specific columns.
The following SQL statement will insert a new record, but only insert data in the "CustomerName", "City", and "Country" columns (CustomerID will be updated automatically):

INSERT INTO Customers (CustomerName, City, Country)
VALUES ('Cardinal', 'Stavanger', 'Norway');

The selection from the "Customers" table will now look like this:

CustomerID	**CustomerName**	**ContactName**	**Address**	**City**	**PostalCode**	**Country**
89	White Clover Markets	Karl Jablonski	305 - 14th Ave. S. Suite 3B	Seattle	98128	USA
90	Wilman Kala	Matti Karttunen	Keskuskatu 45	Helsinki	21240	Finland
91	Wolski	Zbyszek	ul. Filtrowa 68	Walla	01-012	Poland
92	Cardinal	Null	null	Stavanger	null	Norway

2.1.3 INSERT INTO SELECT Statement

The INSERT INTO SELECT statement copies data from one table and inserts it into another table.

- INSERT INTO SELECT requires that data types in source and target tables match
- The existing records in the target table are unaffected

Copy all columns from one table to another table.

Syntax:
INSERT INTO *table2* SELECT * FROM *table1* WHERE *condition*;
Copy only some columns from one table into another table:

Syntax:
INSERT INTO *table2* (*column1*, *column2*, *column3*, ...) SELECT *column1*, *column2*, *column3*, ... FROM *table1* WHERE *condition*;

Example1
The following SQL statement copies "Suppliers" into "Customers" (the columns that are not filled with data, will contain NULL):
INSERT INTO Customers (CustomerName, City, Country) SELECT SupplierName, City, Country FROM Suppliers;
The following SQL statement copies "Suppliers" into "Customers" (fill all columns):

Example2

The following SQL statement copies "Suppliers" into "Customers" (fill all columns):
INSERT INTO Customers (CustomerName, ContactName, Address, City, PostalCode, Country)
SELECT SupplierName, ContactName, Address, City, PostalCode, Country FROM Suppliers;

Example3
The following SQL statement copies only the German suppliers into "Customers":
INSERT INTO Customers (CustomerName, City, Country)
SELECT SupplierName, City, Country FROM Suppliers WHERE Country='Germany';

2.2 SELECT STATEMENT

2.2.1 SQL SELECT STATEMENT INTRODUCTION

The **SELECT** statement in SQL is used to retrieve data from a relational database.

SYNTAX

Number of Columns	**SQL Syntax**
1	SELECT "column_name" FROM "table_name";
More Than 1	SELECT "column_name1"[, "column_name2"] FROM "table_name";
All	SELECT * FROM "table_name";

"table_name" is the name of the table where data is stored, and "column_name" is the name of the column containing the data to be retrieved.

To select more than one column, add a comma to the name of the previous column, and then add the column name. If you are selecting three columns, the syntax will be,

SELECT "column_name1", "column_name2", "column_name3" FROM "table_name";

Note there is no comma after the last column selected.

EXAMPLES

We will provide examples for each of the following three use cases:

- Retrieve one column
- Retrieve multiple columns
- Retrieve all columns

Let's use the following table to illustrate all three cases:

Table ***Store_Information***

Store_Name	Sales	Txn_Date
Los Angeles	1500	Jan-05-1999
San Diego	250	Jan-07-1999
Los Angeles	300	Jan-08-1999
Boston	700	Jan-08-1999

Example 1: Select one column

To select a single column, we specify the column name between **SELECT** and **FROM** as follows:

```
SELECT Store_Name FROM Store_Information;
```

Result:

Store_Name
Los Angeles
San Diego
Los Angeles
Boston

Example 2: Select multiple columns

We can use the **SELECT** statement to retrieve more than one column. To select Store_Name and Sales columns from ***Store_Information***, we use the following SQL:

```
SELECT Store_Name, Sales FROM Store_Information;
```

Result:

Store_Name	Sales
Los Angeles	1500
San Diego	250

Los Angeles	300
Boston	700

Example 3: Select all columns

There are two ways to select all columns from a table. The first is to list the column name of each column. The second, and the easier, way is to use the symbol *. For example, to select all columns from ***Store_Information***, we issue the following SQL:

```
SELECT * FROM Store_Information;
```

Result:

Store_Name	**Sales**	**Txn_Date**
Los Angeles	1500	Jan-05-1999
San Diego	250	Jan-07-1999
Los Angeles	300	Jan-08-1999
Boston	700	Jan-08-1999

2.2.2 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

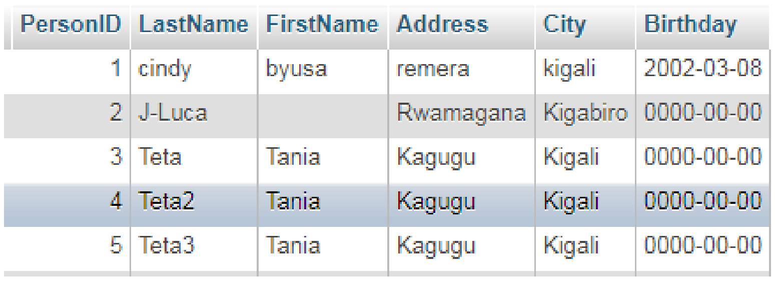
SELECT *column_name* AS *alias_name*
FROM *table_name;*

Alias Table Syntax

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

EXAMPLE:

Below is a selection from the "Persons" table:



PersonID	LastName	FirstName	Address	City	Birthday
1	cindy	byusa	remera	kigali	2002-03-08
2	J-Luca		Rwamagana	Kigabiro	0000-00-00
3	Teta	Tania	Kagugu	Kigali	0000-00-00
4	Teta2	Tania	Kagugu	Kigali	0000-00-00
5	Teta3	Tania	Kagugu	Kigali	0000-00-00

Alias for Columns Examples

The following SQL statement creates two aliases, one for the PersonID column and one for the LastName column:

SELECT PersonID AS ID, LastName AS Person FROM Persons;

Note: It requires double quotation marks or square brackets if the alias name contains spaces:

Example:

SELECT PersonID AS ID, LastName AS 'Person Name' FROM Persons

The following SQL statement creates an alias named "Address" that combine two columns (Address, City):

Example:

SELECT PersonID, LastName, Address + ', ' + City AS Address FROM Persons;

Note: To get the SQL statement above to work in MySQL use the following:
SELECT PersonID, LastName, CONCAT(Address, ',', City) AS Address FROM persons

2.2.3 SQL SELECT DISTINCT Statement

In SQL, the **DISTINCT** keyword is used in the SELECT statement to retrieve unique values from a database table. Any value that has a duplicate will only show up once.

SYNTAX:

```
SELECT DISTINCT "column_name"
FROM "table_name";
```

"table_name" is the name of the table where data is stored, and "column_name" is the name of the column containing the data to be retrieved.

EXAMPLES

The examples will use the following table:

Table ***Store_Information***

Store_Name	**Sales**	**Txn_Date**
Los Angeles	1500	Jan-05-1999
San Diego	250	Jan-07-1999
Los Angeles	300	Jan-08-1999
Boston	700	Jan-08-1999

Example:

To select all distinct stores in Table ***Store_Information***, refer to the query below:

```
SELECT DISTINCT Store_Name FROM Store_Information;
```

Result:

Store_Name
Los Angeles
San Diego
Boston

2.2.4 SQL TOP, LIMIT or ROWNUM Clause

The SELECT TOP clause is used to specify the number of records to return. The SELECT TOP clause is useful on large tables with thousands of records. Returning a large number of records can impact on performance.

Note: Not all database systems support the SELECT TOP clause. MySQL supports the LIMIT clause to select a limited number of records, while Oracle uses ROWNUM.

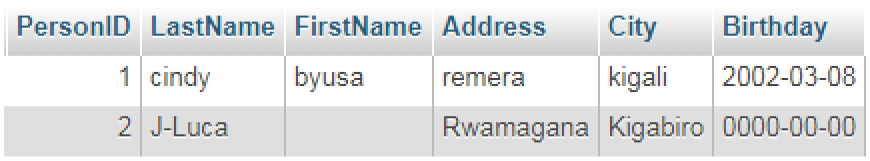
MySQL Syntax:

```
SELECT column_name(s)
FROM table_name
WHERE condition
LIMIT number;
```

Example: Consider SAMPLE persons table above

```
SELECT * FROM  persons LIMIT 2
```

RESULT-SET:



PersonID	LastName	FirstName	Address	City	Birthday
1	cindy	byusa	remera	kigali	2002-03-08
2	J-Luca		Rwamagana	Kigabiro	0000-00-00

2.3 CREATE REPORTS OF SORTED AND RESTRICTED DATA

2.3.1 Limit the Rows Retrieved by a Query

2.3.1.1 Where clause

The WHERE clause is used to filter the result set based on the condition specified following the word WHERE.

The WHERE clause can be used with the following types of SQL statements:

- SELECT
- UPDATE
- DELETE

Syntax

The syntax for using WHERE in the SELECT statement is as follows:

```
SELECT "column_name"
FROM "table_name"
WHERE "condition";
```

Example: WHERE Clause with Simple Condition

To select all stores with sales above $1,000 in Table ***Store_Information***,

Table ***Store_Information***

Store_Name	Sales	Txn_Date
Los Angeles	1500	Jan-05-1999
San Diego	250	Jan-07-1999
Los Angeles	300	Jan-08-1999
Boston	700	Jan-08-1999

```
SELECT Store_Name
FROM Store_Information
WHERE Sales > 1000;
```

Result:

Store_Name
Los Angeles

2.3.1.2 SQL Operators in The WHERE Clause

a) SQL Comparison Operators

The following operators can be used in the WHERE clause:

Operator	Description	Example

=	Equal	SELECT * FROM persons
		WHERE city = ‘Kigali’;

<>	Not equal. **Note:** In some versions of SQL this operator may be written as !=	SELECT * FROM persons
		WHERE Birthday != '0000-00-00'

>	Greater than	SELECT * FROM persons
		WHERE Birthday > '0000-00-00'

<	Less than	SELECT * FROM persons
		WHERE Birthday < '0000-00-00'

>=	Greater than or equal	SELECT * FROM persons
		WHERE Birthday >= '2002-00-00'

<=	Less than or equal	SELECT * FROM persons
		WHERE Birthday <= '2002-00-00'

b) SQL Logical Operators

Logical operators, like comparison *operators*, return a *Boolean* data type with a value of TRUE, FALSE, or UNKNOWN.

Operator	**Description**
ALL	TRUE if all of the subquery values meet the condition
AND	TRUE if all the conditions separated by AND is TRUE
ANY	TRUE if any of the subquery values meet the condition
BETWEEN	TRUE if the operand is within the range of comparisons
EXISTS	TRUE if the subquery returns one or more records
IN	TRUE if the operand is equal to one of a list of expressions
LIKE	TRUE if the operand matches a pattern
NOT	Displays a record if the condition(s) is NOT TRUE
OR	TRUE if any of the conditions separated by OR is TRUE
SOME	TRUE if any of the subquery values meet the condition

a) AND/OR operators

The keywords AND and OR are Boolean operators used to specify compound conditions in the **WHERE** clause.

SYNTAX

The syntax for using **AND** and **OR** in a compound condition is as follows:

```
SELECT "column_name"
FROM "table_name"
WHERE "simple condition"
{ [AND|OR] "simple condition"}+;
```

The { }+ means that the expression inside the bracket will occur one or more times. [AND|OR] means that either **AND** or **OR** can be used. In addition, we can use the parenthesis sign ( ) to indicate the order of the condition.

EXAMPLE

We use the following table as our example:

Table ***Store_Information***

Store_Name	**Sales**	**Txn_Date**
Los Angeles	1500	Jan-05-1999
San Diego	250	Jan-07-1999
San Francisco	300	Jan-08-1999
Boston	700	Jan-08-1999

If we want to select all stores with sales greater than $1,000 or all stores with sales less than $500 but greater than $275 in Table ***Store_Information***, we key in,

```
SELECT Store_Name
FROM Store_Information
WHERE Sales > 1000
OR (Sales < 500 AND Sales > 275);
```

Result:

Store_Name
Los Angeles
San Francisco

b) IN operator

The **IN** operator in SQL filters the result set based on a list of discrete values. The list of discrete values can be simply be listed out or is provided by a separate **SELECT** statement (this is called a **subquery**).

The **IN** operator is always used with the **WHERE** clause.

SYNTAX

Below is the syntax for the **IN** operator when the possible values are listed out directly.

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

The number of values in the parenthesis can be one or more, with each values separated by comma. Values can be numerical or string characters. If there is only one value inside the parenthesis, this commend is equivalent to,

```
WHERE "column_name" = 'value1'
```

The syntax for the **IN** operator in a subquery construct is as follows:

```
SELECT "column_name"
FROM "table_name"
WHERE "column_name" IN ( [SELECT STATEMENT] );
```

Please note that the **IN** operator cannot be used if the filtering criteria is a continuous range. For example, if we are looking for any value that is between 0 and 1, we cannot use the **IN** operator because it is not possible to list every possible value between 0 and 1.

EXAMPLE

We use the following table for our example.

Table ***Store_Information***

Store_Name	**Sales**	**Txn_Date**
Los Angeles	1500	Jan-05-1999
San Diego	250	Jan-07-1999
San Francisco	300	Jan-08-1999
Boston	700	Jan-08-1999

To select all records for the Los Angeles and the San Diego stores in Table ***Store_Information***, we key in,

```
SELECT *
FROM Store_Information
WHERE Store_Name IN ('Los Angeles', 'San Diego');
```

Result:

Store_Name	Sales	Txn_Date

Los Angeles	1500	Jan-05-1999
San Diego	250	Jan-07-1999

c) BETWEEN OPERATOR

The **BETWEEN** operator is used when the filtering criteria is a continuous range with a maximum value and a minimum value. It is always used in the **WHERE** clause.

SYNTAX

```
SELECT "column_name"
FROM "table_name"
WHERE "column_name" BETWEEN 'value1' AND 'value2';
```

This will select all rows whose column has a value between 'value1' and 'value2.'

EXAMPLES

We use the following table for our examples.

Table ***Store_Information***

Store_Name	**Sales**	**Txn_Date**
Los Angeles	1500	Jan-05-1999
San Diego	250	Jan-07-1999
San Francisco	300	Jan-08-1999
Boston	700	Jan-08-1999

Example 1

To select view all sales information between January 6, 1999, and January 10, 1999, we key in,

```
SELECT *
FROM Store_Information
WHERE Txn_Date BETWEEN 'Jan-06-1999' AND 'Jan-10-1999';
```

Note that date may be stored in different formats in different databases. This tutorial simply choose one of the formats.

Result:

Store_Name	**Sales**	**Txn_Date**
San Diego	**250**	**Jan-07-1999**
San Francisco	**300**	**Jan-08-1999**
Boston	**700**	**Jan-08-1999**

BETWEEN is an inclusive operator, meaning that 'value1' and 'value2' are included in the result. If we wish to exclude 'value1' and 'value2' but include everything in between, we need to change the query to the following:

```
SELECT "column_name"
FROM "table_name"
WHERE ("column_name" > 'value1')
AND ("column_name" < 'value2');
```

Example 2

We can also use the **BETWEEN** operator to exclude a range of values by adding **NOT** in front of **BETWEEN**. In the above example, if we want to show all rows where the Sales column is not between 280 and 1000, we will use the following SQL:

```
SELECT *
FROM Store_Information
WHERE Sales NOT BETWEEN 280 and 1000;
```

Result:

Store_Name	**Sales**	**Txn_Date**
Los Angeles	1500	Jan-05-1999
San Diego	250	Jan-07-1999

d) LIKE operator

The **LIKE** operator is used to filter the result set based on a string pattern.

SYNTAX

```
SELECT "column_name"
FROM "table_name"
WHERE "column_name" LIKE {PATTERN};
```

{PATTERN} often consists of wildcards. We saw several examples of **wildcard** matching in the previous section.

EXAMPLE

We use the following table for our example.

Table ***Store_Information***

Store_Name	**Sales**	**Txn_Date**
LOS ANGELES	1500	Jan-05-1999
SAN DIEGO	250	Jan-07-1999
SAN FRANCISCO	300	Jan-08-1999
BOSTON	700	Jan-08-1999

We want to find all stores whose name contains 'AN'. To do so, we key in,

```
SELECT *
FROM Store_Information
WHERE Store_Name LIKE '%AN%';
```

Result:

Store_Name	Sales	Txn_Date
LOS ANGELES	1500	Jan-05-1999
SAN DIEGO	250	Jan-07-1999
SAN FRANCISCO	300	Jan-08-1999

Wildcard

Wildcards are used in SQL to match a string pattern. There are two types of wildcards:

- **%** (percent sign) represents zero, one, or more characters.
- **_** (underscore) represents exactly one character.

Wildcards are used with the **LIKE** operator in SQL.

EXAMPLES

Below are some wildcard examples:

- 'A_Z': All string that starts with 'A', another character, and end with 'Z'. For example, 'ABZ' and 'A2Z' would both satisfy the condition, while 'AKKZ' would not (because there are two characters between A and Z instead of one).
- 'ABC%': All strings that start with 'ABC'. For example, 'ABCD' and 'ABCABC' would both satisfy the condition.
- '%XYZ': All strings that end with 'XYZ'. For example, 'WXYZ' and 'ZZXYZ' would both satisfy the condition.
- '%AN%': All strings that contain the pattern 'AN' anywhere. For example, 'LOS ANGELES' and 'SAN FRANCISCO' would both satisfy the condition.
- '_AN%': All strings that contain a character, then 'AN', followed by anything else. For example, 'SAN FRANCISCO' would satisfy the condition, while 'LOS ANGELES' would not satisfy the condition.

e) Exists operator

EXISTS is a Boolean operator used in a **subquery** to test whether the inner query returns any row. If it does, then the outer query proceeds. If not, the outer query does not execute, and the entire SQL statement returns nothing.

SYNTAX

The syntax for **EXISTS** is:

```
SELECT "column_name1"
FROM "table_name1"
WHERE EXISTS
(SELECT *
FROM "table_name2"
WHERE "condition");
```

Please note that instead of *, you can select one or more columns in the inner query. The effect will be identical.

Example

We use the following tables for our example.

Table ***Store_Information***

Store_Name	Sales	Txn_Date

Los Angeles	1500	Jan-05-1999
San Diego	250	Jan-07-1999
Los Angeles	300	Jan-08-1999
Boston	700	Jan-08-1999

Table ***Geography***

Region_Name	**Store_Name**
East	Boston
East	New York
West	Los Angeles
West	San Diego

The following SQL query,

```
SELECT SUM(Sales) FROM Store_Information
WHERE EXISTS
(SELECT * FROM Geography
WHERE Region_Name = 'West');
```

produces the result below:

SUM(Sales)
2750

At first, this may appear confusing, because the subquery includes the [region_name = 'West'] condition, yet the query summed up sales for stores in all regions. Upon closer inspection, we find that since the subquery returns more than zero row, the **EXISTS** condition is true, and the rows returned from the query "SELECT SUM(Sales) FROM Store_Information" become the final result.

f) ALL/ANY/SOME Operators

Syntax

```
SELECT ... WHERE expression comparison {ALL | ANY | SOME} ( subquery )
```

Keywords:

WHERE **expression**

Tests a scalar expression (such as a column) against every value in the subquery for *ALL*, and against every value until a match is found for *ANY* and *SOME*. All rows must match the expression to return a Boolean *TRUE* value for the *ALL* operator, while one or more rows must match the expression to return a Boolean *TRUE* value for the *ANY* and *SOME* operators.

Comparison

Compares the expression to the subquery. The comparison must be a standard comparison operator like =, <>, !=, >, >=, <, or <=.

Note: These SQL operators will be coved in details in SUBQUERY topic.

2.3.2 Sort the Rows Retrieved by a Query

The **ORDER BY** command in SQL sorts the result set in either ascending or descending order.

ORDER BY usually appears ***last*** in a SQL statement because it is performed after the result set has been retrieved.

SYNTAX

The syntax for an **ORDER BY** statement is as follows:

```
SELECT "column_name"
FROM "table_name"
[WHERE "condition"]
ORDER BY "column_name" [ASC, DESC];
```

The [ ] means that the **WHERE** statement is optional. However, if a **WHERE** clause exists, it comes before the **ORDER BY** clause. **ASC** means that the results will be shown in ascending order, and **DESC** means that the results will be shown in descending order. If neither is specified, the default is **ASC**.

It is possible to order by more than one column. In this case, the **ORDER BY** clause above becomes

```
ORDER BY "column_name1" [ASC, DESC], "column_name2" [ASC, DESC]
```

Assuming that we choose ascending order for both columns, the output will be ordered in ascending order according to column 1. If there is a tie for the value of column 1, we then sort in ascending order by column 2.

EXAMPLES

Table ***Store_Information***

Store_Name	**Sales**	**Txn_Date**
Los Angeles	1500	Jan-05-1999
San Diego	250	Jan-07-1999
San Francisco	300	Jan-08-1999
Boston	700	Jan-08-1999

Example 1: ORDER BY a single column using column name

To list the contents of Table ***Store_Information*** by Sales in descending order, we key in,

```
SELECT Store_Name, Sales, Txn_Date
FROM Store_Information
ORDER BY Sales DESC;
```

Result:

Store_Name	**Sales**	**Txn_Date**
Los Angeles	**1500**	**Jan-05-1999**
Boston	**700**	**Jan-08-1999**
San Francisco	**300**	**Jan-08-1999**
San Diego	**250**	**Jan-07-1999**

Example 2: ORDER BY a single column using column position

In addition to column name, we may also use column position (based on the SQL query) to indicate which column we want to apply the **ORDER BY** clause. The first column is 1, second column is 2, and so on. In the above example, we will achieve the same results by the following command:

```
SELECT Store_Name, Sales, Txn_Date
FROM Store_Information
ORDER BY 2 DESC;
```

Example 3: ORDER BY a single column using a column not in the SELECT statement

The column(s) we use to sort the result do not need to be in the **SELECT** clause. For example, the following SQL,

```
SELECT Store_Name
FROM Store_Information
ORDER BY Sales DESC;
```

works fine and will give the following result:

Store_Name
Los Angeles
Boston
San Francisco
San Diego

Example 4: ORDER BY an expression

It is also possible to sort the result by an expression. For example, in the following table,

Table ***Product_Sales***

Product_ID	**Price**	**Units**
1	10	9
2	15	4
3	25	3

We can use the SQL statement below to order the results by Revenue (defined as Price * Units):

```
SELECT Product_ID, Price*Units Revenue
FROM Product_Sales
ORDER BY Price*Units DESC;
```

Result:

Product_ID	**Revenue**
1	**90**
3	**75**
2	**60**

2.4 Single-row functions to generate and retrieve customized data

MySQL has many built-in functions. They are categorized into string, numeric, date, and some advanced functions in MySQL.

2.4.1 String functions

2.4.1.1 Character Case Conversion Functions

a) UPPER

We use SQL UPPER function to convert the characters in the expression into uppercase. It converts all characters into capital letters.

Syntax:

```
SELECT UPPER(expression) FROM [Source Data]
```

Let's use some examples for this function and view the output.

Example 1:

```
SELECT UPPER('Learn SQL to become a good database administrator');
```

OUTPUT:

```
LEARN SQL TO BECOME A GOOD DATABASE ADMINISTRATOR
```

Example 2:

In the following query, it creates an employee table and inserts record in it.

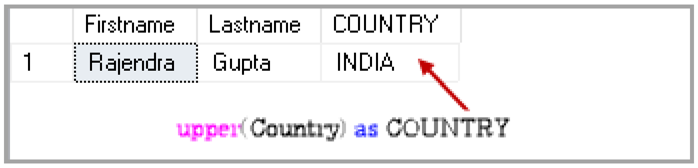
```
Create table Employee
(
  Firstname varchar(20),
  Lastname varchar(20),
  Country varchar(20)
)

Insert into Employee values ('Rajendra','Gupta','India')
```

We want values in the country column to be in uppercase. Let's use the Upper function.

```
SELECT Firstname,
       Lastname,
       upper(Country) as COUNTRY
FROM Employee;
```

In the output, we can see the uppercase value for a Country column.



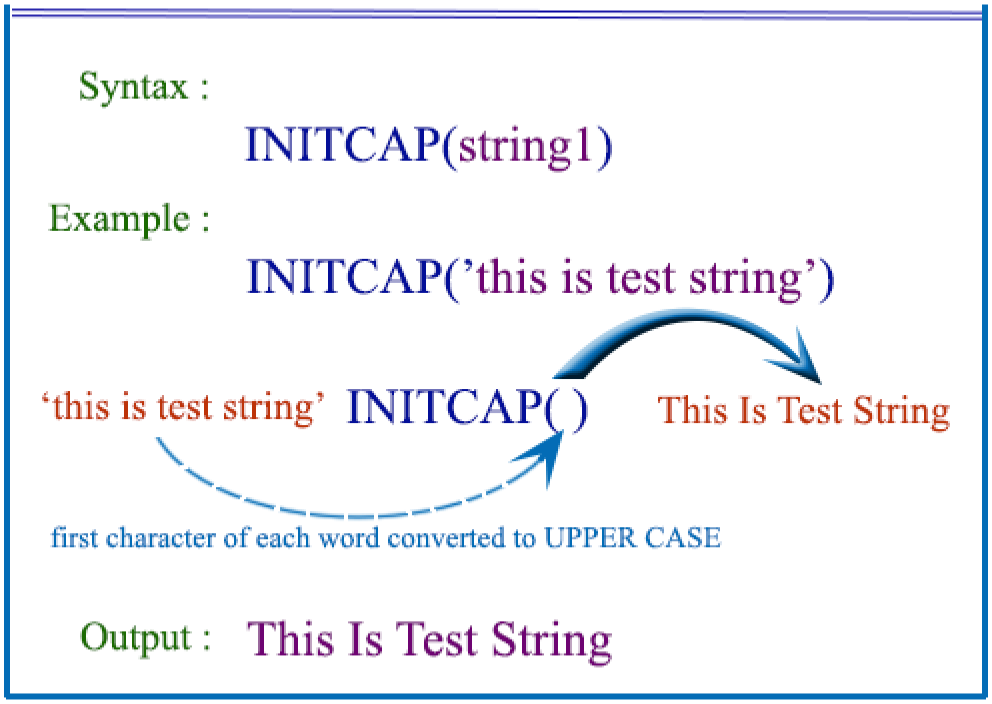
	Firstname	Lastname	COUNTRY
1	Rajendra	Gupta	INDIA

b) LOWER

It converts uppercase letters to lower case for the specified text or expression. It works opposite to the SQL UPPER function.

c) INITCAP

The **INITCAP function** converts the first letter of each word in a string to uppercase, and converts any remaining characters in each word to lowercase. Words are delimited by white space characters, or by characters that are not alphanumeric.



Example 1:

```
SELECT INITCAP('steven king') "Emp. Name"
FROM DUAL;
```

Output :

Emp.name
Steven King

Example2: The following SQL INITCAP Function will return employee names with capital first letter and rest of the name as lower letters:

SELECT INITCAP(first_name) FROM employee;

Output:

```
John
Franklin
Alicia
Jennifer
Ramesh
Joyce
Ahmad
James
```

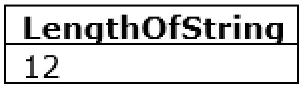
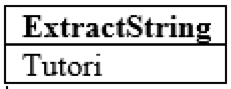
2.4.1.2 Character Manipulation Functions

Also called string functions, they take some value and return string value.

String function	**Usage**	**Syntax**	**Example**
CONCAT ()	adds two or more expressions together	CONCAT(*expression1*, *expression2*, *expression3*,...)	SELECT CONCAT("SQL ", "Tutorial ", "is ", "fun!") AS ConcatenatedString; Output: SQL Tutorial is fun!
REPLACE ()	Replaces all occurrences of a substring within a string, with a new substring. **Note**: This function performs a case-sensitive replacement.	REPLACE(*string*, *from_string*, *new_string*)	SELECT REPLACE("XYZ FGH XYZ", "X", "M"); Output: MYZ FGH MYZ
RTRIM ()	The RTRIM() function removes trailing spaces from a string.	RTRIM(*string*)	SELECT RTRIM("SQL Tutorial ") AS RightTrimmedString; Output: SQL Tutorial

LTRIM ()	removes leading spaces from a string.	LTRIM(*string*)	SELECT LTRIM(" SQL Tutorial") AS LeftTrimmedString; Output: SQL Tutorial

LPAD | RPAD	LPAD() function left-pads a string with another string, to a certain length. For RPAD, the string is added on right	.LPAD(*string*, *length*, *lpad_string*)
		.RPAD(*string*, *length*, *lpad_string*)
			SELECT LPAD("SQL Tutorial", 16, "ABC");
			Output: ABCASQL Tutorial
			
			SELECT RPAD("SQL Tutorial", 19, "ABC");
			Output: SQL TutorialABCABCA

INSTR	It returns the position of the first happening of a string in another string.	INSTR (original_string, sub_string)	SELECT INSTR("soshgths.org", "org") AS MatchPosition; **Output:** **MatchPosition** 10 SELECT INSTR("soshgths.org", "G") AS MatchPosition; **MatchPosition** 5
LENGTH	It returns the length of a string.	LENGTH (string)	SELECT LENGTH("SQL Tutorial") AS LengthOfString; **Output:**  **LengthOfString** 12
SUBSTR	It extracts a substring from a string (starting at any position).	SUBSTR (string, start_position, substr_length)	SELECT SUBSTR("SQL Tutorial", 5, 6) AS ExtractString; **Output:**  **ExtractString** Tutori

2.4.2 Numeric Functions

ROUND FUNCTION

The **ROUND** function in SQL is used to round a number to a specified precision.

Syntax: ROUND(*number*, *decimals*)

Example: SELECT ROUND(345.156, 2);

Output: 345.16

TRUNC FUNCTION
The TRUNCATE() function truncates a number to the specified number of decimal places.
Syntax: TRUNCATE(*number*, *decimals*)
Example: SELECT TRUNCATE(345.156, 2);
Output: 345.15

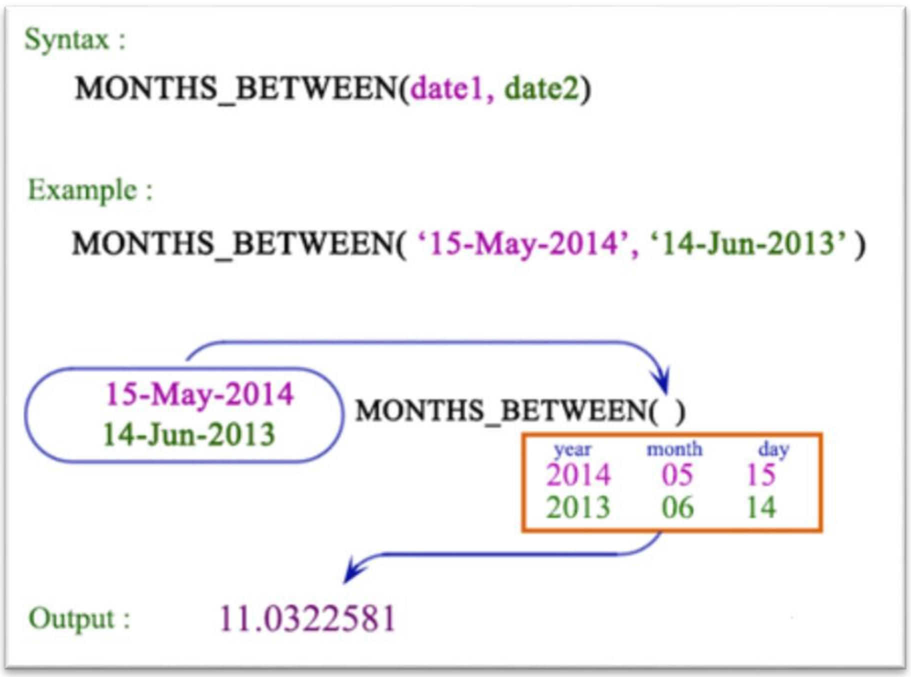
CEIL FUNCTION
The CEIL() function returns the smallest integer value that is bigger than or equal to a number.
Syntax: CEIL(*number*)
Example: SELECT CEIL(25.1);
Output: 26

FLOOR FUNCTION
The FLOOR() function returns the largest integer value that is smaller than or equal to a number.
Syntax: FLOOR(*number*)
Example: SELECT FLOOR(25.7);
Output: 25

MOD FUNCTION
The MOD() function returns the remainder of a number divided by another number.
Syntax: MOD(*x*, *y*)
Example: SELECT MOD (18, 7);
Output: 2

2.4.3 Date functions

a) MONTHS_BETWEEN FUNCTION
The MONTHS_BETWEEN() function is used to get the number of months between dates (date1, date2).



b) ADD_MONTHS FUNCTION

It adds the specified number of months to a date and returns the sum as a DATE.
Syntax: ADD_MONTHS (*start-date*, *num-months*);
Example: SELECT ADD_MONTHS('2008-02-29 23:30 PST', 24);
Output: 2010-03-01

c) LAST_DAY FUNCTION

The LAST_DAY() function extracts the last day of the month for a given date.
Syntax: LAST_DAY(*date*)
Example: SELECT LAST_DAY("2020-02-10 09:34:00");
Output: 2020-02-29

d) NEXT_DAY FUNCTION

It returns the date of the first weekday that is later than the date.
The NEXT_DAY function returns the date of the next specified day of the week after a specified date.

Syntax: NEXT_DAY(source_date, day_of_week);

***source_date**:* the source date.

***day_of_week**:* The day of the week. This is the case-insensitive name of a day in the date language of your session. You can also specify day-name abbreviations, in which case any characters after the recognized abbreviation are ignored. For example, if you're using English, you can use the following values (again, the case of the characters is ignored):

Day Name	Abbreviation
Sunday	Sun
Monday	Mon
Tuesday	Tue
Wednesday	Wed
Thursday	Thu
Friday	Fri
Saturday	Sat

Example: SELECT NEXT_DAY('3-15-2020','Thursday') "NEXT DAY" ;

Output: 3-19-2020

2.5 Report aggregated data using group functions

2.5.1 Aggregate Functions

In database management an aggregate function is a function where the values of multiple rows are grouped together as input on certain criteria to form a single value of more significant meaning.

a) **Sum()**

SELECT SUM returns the sum of the data values.
Syntax: SELECT SUM(column-name) FROM table-name

b) **Avg()**

SELECT AVG returns the average of the data values.
Syntax: SELECT AVG(column-name) FROM table-name

c) **Min()**

SELECT MIN returns the minimum value for a column.
Syntax: SELECT MIN(column-name) FROM table-name

d) Max()

SELECT MAX returns the maximum value for a column.
Syntax: SELECT MAX(column-name) FROM table-name

e) Count()

SELECT COUNT returns a count of the number of data values.
Syntax: SELECT COUNT(column-name) FROM table-name

Now let us understand each Aggregate function with an example:

Id	Name	Salary
1	A	80
2	B	40
3	C	60
4	D	70
5	E	60
6	F	Null

Count():
Count(*): Returns total number of records .i.e 6.
Count(salary): Return number of Non Null values over the column salary. i.e 5.
Count(Distinct Salary): Return number of distinct Non Null values over the column salary .i.e. 4

Sum():
sum(salary): Sum all Non Null values of Column salary i.e., 310
sum(Distinct salary): Sum of all distinct Non-Null values i.e., 250.

Avg():
Avg(salary) = Sum(salary) / count(salary) = 310/5
Avg(Distinct salary) = sum(Distinct salary) / Count(Distinct Salary) = 250/4

Min() and Max():

Min(salary): Minimum value in the salary column except NULL i.e., 40.
Max(salary): Maximum value in the salary i.e., 80.

2.5.2 GROUP BY clause

Group by clause is used to group the results of a ***SELECT*** query based on one or more columns. It is also or often used with SQL aggregate functions to group the result from one or more tables.

Syntax for using Group by in a statement.

SELECT column_name, function(column_name) FROM table_name [WHERE condition]

GROUP BY column_name

Example of Group by in a Statement

Consider the following **Emp** table.

Eid	**Name**	**Age**	**Salary**
401	Anu	22	9000
402	Shane	29	8000
403	Rohan	34	6000
404	Scott	44	9000
405	Tiger	35	8000

Here we want to find **name** and **age** of employees grouped by their **salaries** or in other words, we will be grouping employees based on their salaries, hence, as a result, we will get a data set, with unique salaries listed, alongside the first employee's name and age to have that salary.

Group by is used to group different row of data together based on any one column.

SQL query for the above requirement will be,

SELECT name, age FROM Emp GROUP BY salary

Result will be,

Name	**Age**
Rohan	34
Shane	29
Anu	22

Example of Group by in a Statement with WHERE clause

Consider the above **Emp** table

SQL query will be,

SELECT name, salary FROM Emp WHERE age > 25 GROUP BY salary

Result will be.

Name	Salary
Rohan	6000
Shane	8000
Scott	9000

2.5.2 HAVING clause

Having clause is used with SQL Queries to give more precise condition for a statement. It is used to mention condition in ***Group by*** based SQL queries, just like ***WHERE*** clause is used with ***SELECT*** query.

Syntax for HAVING clause is,

```
SELECT column_name, function(column_name)
FROM table_name
WHERE column_name condition
GROUP BY column_name
HAVING function(column_name) condition
ORDER BY column_name
```

Example of SQL Statement using HAVING

Consider the following **Sale** table.

Oid	order_name	previous_balance	customer
11	ord1	2000	Alex
12	ord2	1000	Adam
13	ord3	2000	Abhi
14	ord4	1000	Adam
15	ord5	2000	Alex

Suppose we want to find the **customer** whose **previous_balance** sum is more than **3000**.

We will use the below SQL query,

SELECT * FROM sale GROUP BY customer HAVING sum(previous_balance) > 3000

Result will be,

oid	order_name	previous_balance	customer
11	ord1	2000	Alex

The main objective of the above SQL query was to find out the name of the customer who has had a **previous_balance** more than **3000**, based on all the previous sales made to the customer, hence we get the first row in the table for customer Alex.

2.6 SQL JOIN

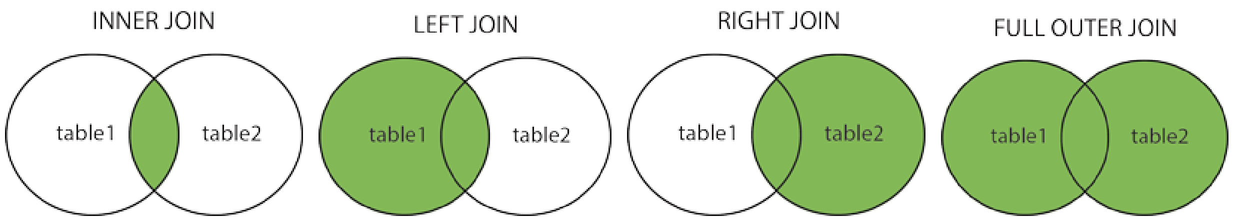
As the name shows, JOIN means *to combine something*. In case of SQL, JOIN means **"to combine two or more tables"**.

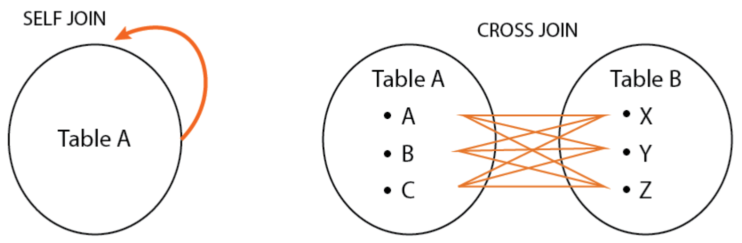
The SQL JOIN clause takes records from two or more tables in a database and combines it together.

Types of SQL JOINs

Here are the different types of the JOINs in SQL:

a) **(INNER) JOIN**: Returns records that have matching values in both tables
b) **LEFT (OUTER) JOIN**: Return all records from the left table, and the matched records from the right table
c) **RIGHT (OUTER) JOIN**: Return all records from the right table, and the matched records from the left table
d) **FULL (OUTER) JOIN**: Return all records when there is a match in either left or right table
e) **Cross join:** produces a result set which is the number of rows in the first table multiplied by the number of rows in the second table
f) **Self-join:** is a query in which a table is joined (compared) to itself.





In the process of joining, rows of both tables are combined in a single table.

a) INNER JOIN

Syntax1: (old version)

```
SELECT table1.column1, table2.column2...
FROM table1, table2
where table1.common_field = table2.common_field;
```

Syntax2: (new version)

```
SELECT table1.column1, table2.column2...
FROM table1
INNER JOIN table2
ON table1.common_field = table2.common_field;
```

Example

Consider the following two tables:

Table 1 − CUSTOMERS Table is as follows:

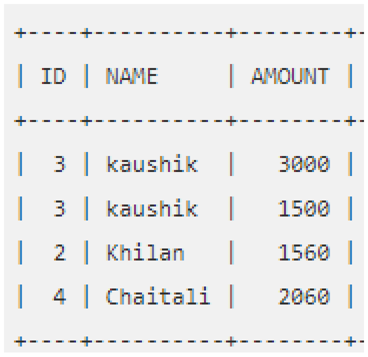
```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000    |
|  2 | Khilan   |  25 | Delhi     |  1500    |
|  3 | kaushik  |  23 | Kota      |  2000    |
|  4 | Chaitali |  25 | Mumbai    |  6500    |
|  5 | Hardik   |  27 | Bhopal    |  8500    |
|  6 | Komal    |  22 | MP        |  4500    |
|  7 | Muffy    |  24 | Indore    | 10000    |
+----+----------+-----+-----------+----------+
```

Table 2 − ORDERS Table is as follows:

```
+-----+---------------------+-------------+--------+
| OID | DATE                | CUSTOMER_ID | AMOUNT |
+-----+---------------------+-------------+--------+
| 102 | 2009-10-08 00:00:00 |           3 |   3000 |
| 100 | 2009-10-08 00:00:00 |           3 |   1500 |
| 101 | 2009-11-20 00:00:00 |           2 |   1560 |
| 103 | 2008-05-20 00:00:00 |           4 |   2060 |
+-----+---------------------+-------------+--------+
```

Now, let us join these two tables using the INNER JOIN as follows:

```
SELECT ID, NAME, AMOUNT   FROM CUSTOMERS
   INNER JOIN ORDERS
   ON CUSTOMERS.ID = ORDERS.CUSTOMER_ID;
```



```
+----+----------+--------+
| ID | NAME     | AMOUNT |
+----+----------+--------+
|  3 | kaushik  |   3000 |
|  3 | kaushik  |   1500 |
|  2 | Khilan   |   1560 |
|  4 | Chaitali |   2060 |
+----+----------+--------+
```

b) LEFT JOINS

The SQL **LEFT JOIN** returns all rows from the left table, even if there are no matches in the right table. This means that if the ON clause matches 0 (zero) records in the right table; the join will still return a row in the result, but with NULL in each column from the right table.

This means that a left join returns all the values from the left table, plus matched values from the right table or NULL in case of no matching join predicate.

Syntax

The basic syntax of a **LEFT JOIN** is as follows:

```
SELECT table1.column1, table2.column2...
FROM table1
LEFT JOIN table2
ON table1.common_field = table2.common_field;
```

Example

Consider the above two tables (Table 1 – CUSTOMERS, Table 2 ORDERS)

Now, let us join these two tables using the LEFT JOIN as follows.

```
SELECT  ID, NAME, AMOUNT, DATE
  FROM CUSTOMERS
  LEFT JOIN ORDERS
  ON CUSTOMERS.ID = ORDERS.CUSTOMER_ID;
```

This would produce the following result :

```
+----+----------+--------+---------------------+
| ID | NAME     | AMOUNT | DATE                |
+----+----------+--------+---------------------+
|  1 | Ramesh   |   NULL | NULL                |
|  2 | Khilan   |   1560 | 2009-11-20 00:00:00 |
|  3 | kaushik  |   3000 | 2009-10-08 00:00:00 |
|  3 | kaushik  |   1500 | 2009-10-08 00:00:00 |
|  4 | Chaitali |   2060 | 2008-05-20 00:00:00 |
|  5 | Hardik   |   NULL | NULL                |
|  6 | Komal    |   NULL | NULL                |
|  7 | Muffy    |   NULL | NULL                |
+----+----------+--------+---------------------+
```

c) RIGHT JOIN

The SQL **RIGHT JOIN** returns all rows from the right table, even if there are no matches in the left table. This means that if the ON clause matches 0 (zero) records in the left table; the join will still return a row in the result, but with NULL in each column from the left table.

This means that a right join returns all the values from the right table, plus matched values from the left table or NULL in case of no matching join predicate.

Syntax

The basic syntax of a **RIGHT JOIN** is as follow.

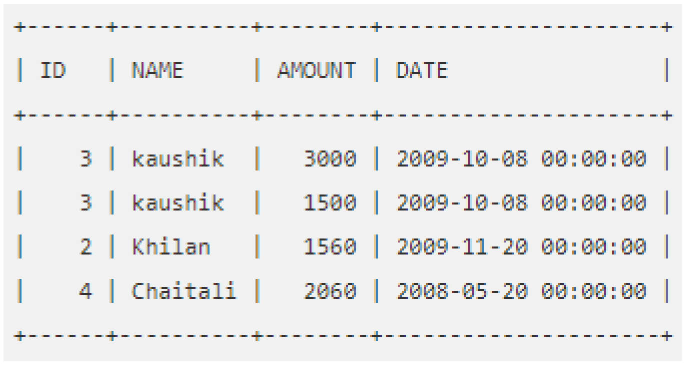
```
SELECT table1.column1, table2.column2...
FROM table1
RIGHT JOIN table2
ON table1.common_field = table2.common_field;
```

Example

Consider the above two tables (Table 1 – CUSTOMERS, Table 2 ORDERS)
Now, let us join these two tables using the RIGHT JOIN as follows.

```
SELECT ID, NAME, AMOUNT, DATE FROM CUSTOMERS
  RIGHT JOIN ORDERS
  ON CUSTOMERS.ID = ORDERS.CUSTOMER_ID;
```

This would produce the following result:



```
+------+----------+--------+---------------------+
| ID   | NAME     | AMOUNT | DATE                |
+------+----------+--------+---------------------+
|    3 | kaushik  |   3000 | 2009-10-08 00:00:00 |
|    3 | kaushik  |   1500 | 2009-10-08 00:00:00 |
|    2 | Khilan   |   1560 | 2009-11-20 00:00:00 |
|    4 | Chaitali |   2060 | 2008-05-20 00:00:00 |
+------+----------+--------+---------------------+
```

d) FULL JOIN

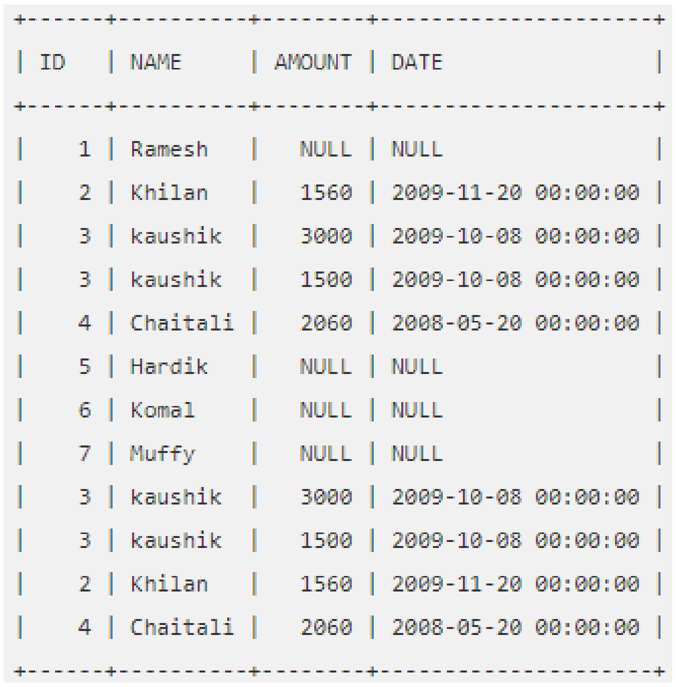
The SQL **FULL JOIN** combines the results of both left and right outer joins.
The joined table will contain all records from both the tables and fill in NULLs for missing matches on either side.

Syntax
The basic syntax of a **FULL JOIN** is as follows:

```
SELECT table1.column1, table2.column2...
FROM table1
FULL JOIN table2
ON table1.common_field = table2.common_field;
```

Example
Consider the above two tables (Table 1 – CUSTOMERS, Table 2 ORDERS)
Now, let us join these two tables using the FULL JOIN as follows.

```
SELECT ID, NAME, AMOUNT, DATE FROM CUSTOMERS
  FULL JOIN ORDERS
  ON CUSTOMERS.ID = ORDERS.CUSTOMER_ID;
```



```
+------+----------+--------+---------------------+
| ID   | NAME     | AMOUNT | DATE                |
+------+----------+--------+---------------------+
|    1 | Ramesh   |   NULL | NULL                |
|    2 | Khilan   |   1560 | 2009-11-20 00:00:00 |
|    3 | kaushik  |   3000 | 2009-10-08 00:00:00 |
|    3 | kaushik  |   1500 | 2009-10-08 00:00:00 |
|    4 | Chaitali |   2060 | 2008-05-20 00:00:00 |
|    5 | Hardik   |   NULL | NULL                |
|    6 | Komal    |   NULL | NULL                |
|    7 | Muffy    |   NULL | NULL                |
|    3 | kaushik  |   3000 | 2009-10-08 00:00:00 |
|    3 | kaushik  |   1500 | 2009-10-08 00:00:00 |
|    2 | Khilan   |   1560 | 2009-11-20 00:00:00 |
|    4 | Chaitali |   2060 | 2008-05-20 00:00:00 |
+------+----------+--------+---------------------+
```

If your Database does not support FULL JOIN (MySQL does not support FULL JOIN), then you can use **UNION ALL** clause to combine these two JOINS as shown below.

```
SELECT ID, NAME, AMOUNT, DATE
  FROM CUSTOMERS
  LEFT JOIN ORDERS
  ON CUSTOMERS.ID = ORDERS.CUSTOMER_ID
UNION ALL
  SELECT ID, NAME, AMOUNT, DATE
  FROM CUSTOMERS
  RIGHT JOIN ORDERS
  ON CUSTOMERS.ID = ORDERS.CUSTOMER_ID
```

e) CARTESIAN or CROSS JOINS

The CARTESIAN JOIN or CROSS JOIN returns the Cartesian product of the sets of records from two or more joined tables. Thus, it equates to an inner join where the join-condition always evaluates to either True or where the join-condition is absent from the statement.

Syntax

The basic syntax of the **CARTESIAN JOIN** or the **CROSS JOIN** is as follows

```
SELECT table1.column1, table2.column2...
FROM table1, table2 [, table3 ]
```

Example

Consider the following two tables:

Table 1 − CUSTOMERS Table is as follows:

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000    |
|  2 | Khilan   |  25 | Delhi     |  1500    |
|  3 | kaushik  |  23 | Kota      |  2000    |
```

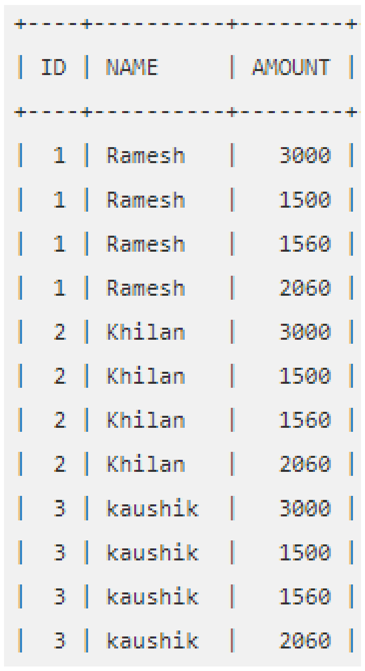
Table 2 − ORDERS Table is as follows:

```
+-----+---------------------+-------------+--------+
| OID | DATE                | CUSTOMER_ID | AMOUNT |
+-----+---------------------+-------------+--------+
| 102 | 2009-10-08 00:00:00 |           3 |   3000 |
| 100 | 2009-10-08 00:00:00 |           3 |   1500 |
| 101 | 2009-11-20 00:00:00 |           2 |   1560 |
| 103 | 2008-05-20 00:00:00 |           4 |   2060 |
+-----+---------------------+-------------+--------+
```

Now, let us join these two tables using the CROSS JOIN as follows:

```
SELECT ID, NAME, AMOUNT FROM CUSTOMERS, ORDERS;
```

This would produce the following result:



```
+----+----------+--------+
| ID | NAME     | AMOUNT |
+----+----------+--------+
|  1 | Ramesh   |   3000 |
|  1 | Ramesh   |   1500 |
|  1 | Ramesh   |   1560 |
|  1 | Ramesh   |   2060 |
|  2 | Khilan   |   3000 |
|  2 | Khilan   |   1500 |
|  2 | Khilan   |   1560 |
|  2 | Khilan   |   2060 |
|  3 | kaushik  |   3000 |
|  3 | kaushik  |   1500 |
|  3 | kaushik  |   1560 |
|  3 | kaushik  |   2060 |
```

f) SELF JOIN

A self-join is a join in which a table is joined with itself (which is also called Unary relationships), especially when the table has a FOREIGN KEY which references its own PRIMARY KEY. To join a table itself means that each row of the table is combined with itself and with every other row of the table.
The self-join can be viewed as a join of two copies of the same table. The table is not actually copied, but SQL performs the command as though it were.

The syntax of the command for joining a table to itself is almost same as that for joining two different tables. To distinguish the column names from one another, aliases for the actual the table name are used, since both the tables have the same name. Table name aliases are defined in the FROM clause of the SELECT statement.

Syntax:

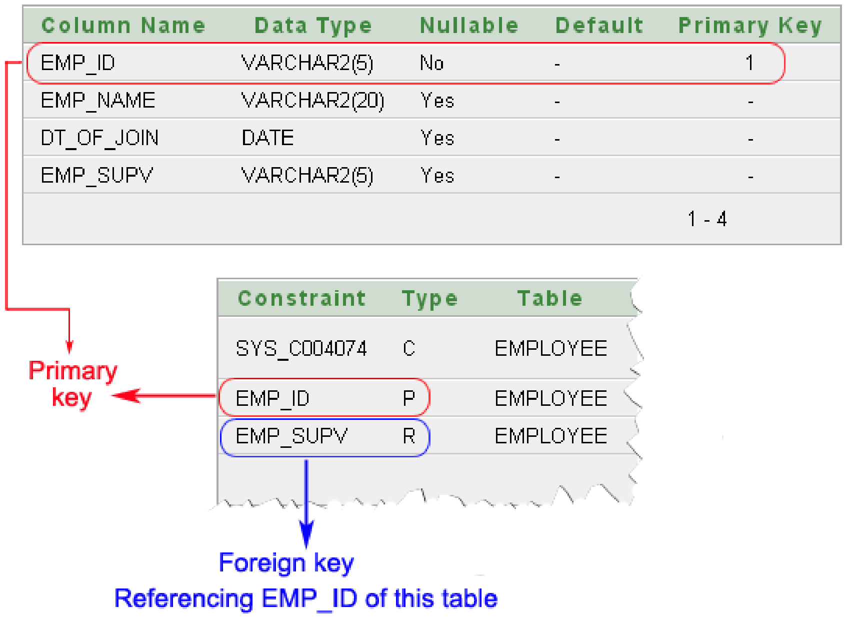
```
SELECT a.column_name, b.column_name...
FROM table1 a, table1 b
WHERE a.common_filed = b.common_field;
```

Code to create the table EMPLOYEE

SQL Code:

```
CREATE TABLE employee (
emp_id varchar (5) NOT NULL,
emp_name varchar (20) NULL,
dt_of_join date NULL,
emp_supv varchar (5) NULL,
CONSTRAINT emp_id PRIMARY KEY (emp_id),
CONSTRAINT emp_supv FOREIGN KEY (emp_supv)
REFERENCES employee (emp_id));
```

The structure of the table



Column Name	Data Type	Nullable	Default	Primary Key
EMP_ID	VARCHAR2(5)	No	-	1
EMP_NAME	VARCHAR2(20)	Yes	-	-
DT_OF_JOIN	DATE	Yes	-	-
EMP_SUPV	VARCHAR2(5)	Yes	-	-
				1 - 4

Constraint	Type	Table
SYS_C004074	C	EMPLOYEE
EMP_ID	P	EMPLOYEE
EMP_SUPV	R	EMPLOYEE

In the EMPLOYEE table displayed above, emp_id is the primary key. emp_supv is the foreign key (this is the supervisor's employee id).

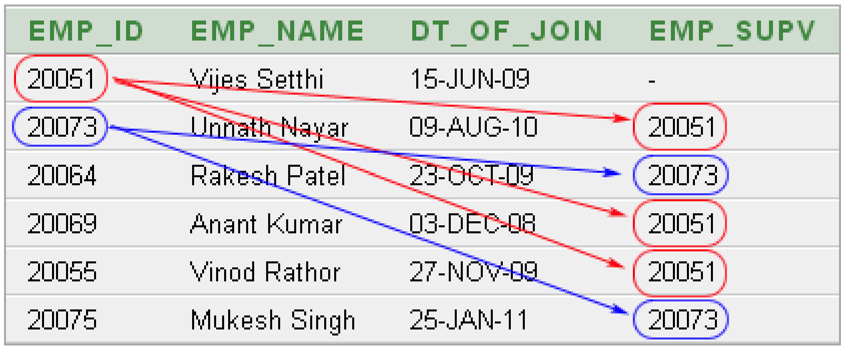
If we want a list of employees and the names of their supervisors, we'll have to JOIN the EMPLOYEE table to itself to get this list.

Unary relationship to employee

How the employees are related to themselves:

- An employee may report to another employee (supervisor).
- An employee may supervise himself (i.e. zero) to many employees (subordinates).

We have the following data into the table EMPLOYEE.



EMP_ID	EMP_NAME	DT_OF_JOIN	EMP_SUPV
20051	Vijes Setthi	15-JUN-09	-
20073	Unnath Nayar	09-AUG-10	20051
20064	Rakesh Patel	23-OCT-09	20073
20069	Anant Kumar	03-DEC-08	20051
20055	Vinod Rathor	27-NOV-09	20051
20075	Mukesh Singh	25-JAN-11	20073

The above data shows:

- Unnath Nayar's supervisor is Vijes Setthi
- Anant Kumar and Vinod Rathor can also report to Vijes Setthi.
- Rakesh Patel and Mukesh Singh are under supervison of Unnith Nayar.

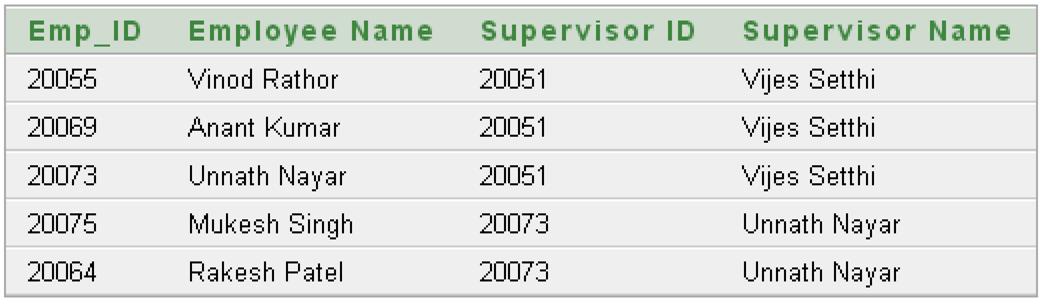
Example of SQL SELF JOIN

In the following example, we will use the table EMPLOYEE twice and in order to do this we will use the alias of the table.
To get the list of employees and their supervisor the following SQL statement has used:

SQL Code:

```
SELECT a.emp_id AS "Emp_ID",a.emp_name AS "Employee Name",
b.emp_id AS "Supervisor ID",b.emp_name AS "Supervisor Name"
FROM employee a, employee b
WHERE a.emp_supv = b.emp_id;
```

Output:



Emp_ID	Employee Name	Supervisor ID	Supervisor Name
20055	Vinod Rathor	20051	Vijes Setthi
20069	Anant Kumar	20051	Vijes Setthi
20073	Unnath Nayar	20051	Vijes Setthi
20075	Mukesh Singh	20073	Unnath Nayar
20064	Rakesh Patel	20073	Unnath Nayar

2.7 SQL NESTED QUERY

A Subquery or Inner query or a Nested query is a query within another SQL query and embedded within the WHERE clause.

A subquery is used to return data that will be used in the main query as a condition to further restrict the data to be retrieved.

Subqueries can be used with the SELECT, INSERT, UPDATE, and DELETE statements along with the operators like =, <, >, >=, <=, IN, BETWEEN, etc.

There are a few rules that subqueries must follow:

- Subqueries must be enclosed within parentheses.
- A subquery can have only one column in the SELECT clause, unless multiple columns are in the main query for the subquery to compare its selected columns.
- An ORDER BY command cannot be used in a subquery, although the main query can use an ORDER BY. The GROUP BY command can be used to perform the same function as the ORDER BY in a subquery.
- Subqueries that return more than one row can only be used with multiple value operators such as the IN operator.
- The SELECT list cannot include any references to values that evaluate to a BLOB, ARRAY, CLOB, or NCLOB.
- A subquery cannot be immediately enclosed in a set function.
- The BETWEEN operator cannot be used with a subquery. However, the BETWEEN operator can be used within the subquery.

Subqueries with the SELECT Statement

Subqueries are most frequently used with the SELECT statement. The basic **syntax:**

```
SELECT column_name [, column_name ]
FROM   table1 [, table2 ]
WHERE  column_name OPERATOR
  (SELECT column_name [, column_name ]
  FROM table1 [, table2 ]
  [WHERE])
```

Example

Consider the CUSTOMERS table having the following records:

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  35 | Ahmedabad |  2000.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
|  7 | Muffy    |  24 | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
```

Now, let us check the following subquery with a SELECT statement.

```
SELECT * FROM CUSTOMERS

WHERE ID IN (SELECT ID FROM CUSTOMERS WHERE SALARY > 4500);
```

This would produce the following result:

```
+----+----------+-----+---------+----------+

| ID | NAME     | AGE | ADDRESS | SALARY   |

+----+----------+-----+---------+----------+

|  4 | Chaitali |  25 | Mumbai  |  6500.00 |

|  5 | Hardik   |  27 | Bhopal  |  8500.00 |

|  7 | Muffy    |  24 | Indore  | 10000.00 |

+----+----------+-----+---------+----------+
```

Subqueries with the INSERT Statement

Subqueries also can be used with INSERT statements. The INSERT statement uses the data returned from the subquery to insert into another table. The selected data in the subquery can be modified with any of the character, date or number functions.

The basic syntax is as follows:

```
INSERT INTO table_name (column1, column2)
  SELECT column1, column2
  FROM table1
  WHERE VALUE OPERATOR;
```

Example

Consider a table CUSTOMERS_BKP with similar structure as CUSTOMERS table. Now to copy the complete CUSTOMERS table into the CUSTOMERS_BKP table, you can use the following syntax.

```
INSERT INTO CUSTOMERS_BKP
SELECT * FROM CUSTOMERS
WHERE ID IN (SELECT ID
FROM CUSTOMERS);
```

Subqueries with the UPDATE Statement

The subquery can be used in conjunction with the UPDATE statement. Either single or multiple columns in a table can be updated when using a subquery with the UPDATE statement.

The basic syntax is as follows.

```
UPDATE table
SET column_name = new_value
[WHERE OPERATOR [VALUE]
  (SELECT COLUMN_NAME
  FROM TABLE_NAME)
  [WHERE)]
```

Example

Assuming, we have CUSTOMERS_BKP table available which is backup of CUSTOMERS table. The following example updates SALARY by 0.25 times in the CUSTOMERS table for all the customers whose AGE is greater than or equal to 27.

```
UPDATE CUSTOMERS
SET SALARY = SALARY * 0.25
WHERE AGE IN (SELECT AGE FROM CUSTOMERS_BKP
WHERE AGE >= 27);
```

This would impact two rows and finally CUSTOMERS table would have the following records.

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  35 | Ahmedabad |   125.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  2125.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
|  7 | Muffy    |  24 | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
```

Subqueries with the DELETE Statement

The subquery can be used in conjunction with the DELETE statement like with any other statements mentioned above.

The basic syntax is as follows:

```
DELETE FROM TABLE_NAME
[ WHERE OPERATOR [ VALUE ]
  (SELECT COLUMN_NAME
  FROM TABLE_NAME)
  [ WHERE) ]
```

Example

Assuming, we have a CUSTOMERS_BKP table available which is a backup of the CUSTOMERS table. The following example deletes the records from the CUSTOMERS table for all the customers whose AGE is greater than or equal to 27.

```
DELETE FROM CUSTOMERS
WHERE AGE IN (SELECT AGE FROM CUSTOMERS_BKP
WHERE AGE >= 27 );
```

This would impact two rows and finally the CUSTOMERS table would have the following records.

```
+----+----------+-----+---------+----------+
| ID | NAME     | AGE | ADDRESS | SALARY   |
+----+----------+-----+---------+----------+
|  2 | Khilan   |  25 | Delhi   |  1500.00 |
|  3 | kaushik  |  23 | Kota    |  2000.00 |
|  4 | Chaitali |  25 | Mumbai  |  6500.00 |
|  6 | Komal    |  22 | MP      |  4500.00 |
|  7 | Muffy    |  24 | Indore  | 10000.00 |
+----+----------+-----+---------+----------+
```

2.8 USE OF SET OPERATORS

2.9 DML STATEMENTS TO UPDATE TABLE DATA

2.9.1 Update operation

Let's take an example of a real-world problem. These days, Facebook provides an option for EDITING your status update, how do you think it works? Yes, using the Update SQL command. UPDATE command is used to update any record of data in a table.

Syntax for SINGLE RECORD or MULTIPLE RECORDS update:

```
UPDATE table_name SET column_name = new_value WHERE some_condition;
```

Example:

Let have a sample table called STUDENT

student_id	**name**	**age**
101	Adam	15
102	Alex	
103	chris	14

UPDATE student SET age=18 WHERE student_id=102;

RESULT:

S_id	S_Name	Age
101	Adam	15
102	Alex	18
103	chris	14

Syntax for ALL RECORDS update:

```
UPDATE table_name SET column_name = new_value;
```

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

Example:

UPDATE student SET age=17;

Here, all records in student table, the column "age" will be updated to 17.

2.9.2 Delete operation

Consider the following **Employee** table for DELETE operation:

Eid	**Name**	**Age**	**Salary**
401	Keza	22	9000
402	Jane	57	8000
403	Rohan	34	6000
404	Sibomana	60	9000
405	Kaliza	35	8000

The SQL **DELETE** Query is used to **delete** the existing **records** from a table. You can use WHERE clause with **DELETE** query to **delete** selected rows, otherwise all the **records** would be **deleted**. DELETE can delete one or more records in a table.

Single record delete

Deleting a single record from a table is very easy task. But it will be easy only if, during design phase proper primary key (PK) defined for the table. A PK helps to identify a single record (row) from whole table.

Syntax: delete from table_Name where condition;

Example1: DELETE FROM employee WHERE Eid = 401;

Example2: DELETE FROM employee WHERE **Age** = 22 and salary=9000;

In first example we know the in employee table there cannot be 2 employees with same employee id because E**id** is a primary key column.

Second case showing how we can use multiple columns to delete a single record (where possible).

Multiple records delete or delete with condition

Syntax:

DELETE FROM table_name WHERE condition;

Example:

DELETE from employee where age>55.

Here, we may need to delete all retired employees (having age greater than 55). In the table above, employee Jane and Sibomana will be deleted from the table.

Delete all records

It is possible to delete all rows in a table without deleting the table. This means that the table structure, attributes, and indexes will be intact: ***DELETE FROM table_name;***

Example: delete from employee. With this statement, all records will be deleted from "*employee*" table.

2.10 EXECUTION OF DATABASE STORED PROCEDURE AND INDEX

2.10.1 STORED PROCEDURE

A stored procedure is a group of one or more database statements (statement of insert, update, delete and select) stored in the database's data dictionary and called from either a remote program, another stored procedure, or the command line.

Main Parts of a Stored Procedure
Stored procedures can be thought of having three main parts:

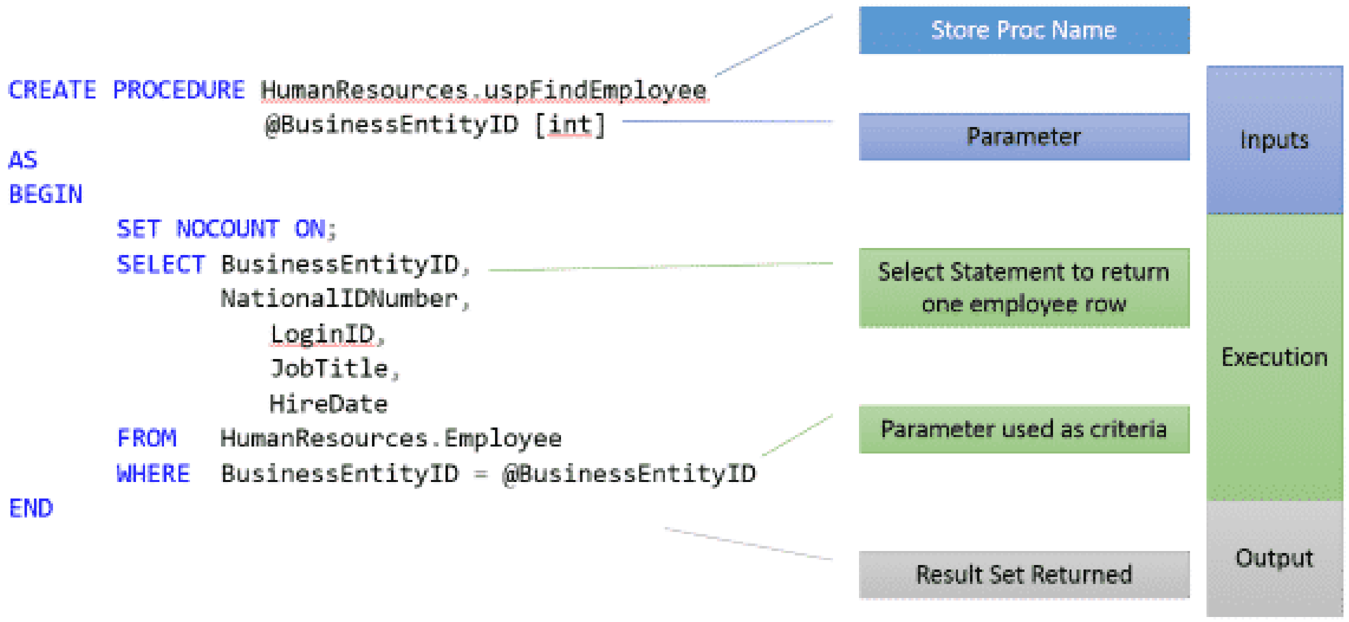
Inputs
Store procedure can accept parameter values as inputs. Depending on how the parameters are defined, modified values can be passed back to the calling program

Execution

- Stored procedures can execute SQL statements
- A stored procedure is able to call another stored procedure.

Outputs
A stored procedure can return a single values such as a number or text value or a result set (set of rows). Also, as mentioned, depending on how the inputs are defined, changed values to inputs can be propagated back to the calling procedure.

Example Stored Procedure
Here is an example of a stored procedure that takes a parameter, executes a query and return a result. Specifically, the stored procedure accepts the BusinessEntityID as a parameter and uses this to match the primary key of the HumanResources.Employee table to return the requested employee.



```
CREATE PROCEDURE HumanResources.uspFindEmployee
                 @BusinessEntityID [int]
AS
BEGIN
        SET NOCOUNT ON;
        SELECT BusinessEntityID,
               NationalIDNumber,
                   LoginID,
                   JobTitle,
                   HireDate
        FROM   HumanResources.Employee
        WHERE  BusinessEntityID = @BusinessEntityID
END
```

Store Proc Name
Parameter
Select Statement to return one employee row
Parameter used as criteria
Result Set Returned
Inputs
Execution
Output

Though this example returns a single row, due to the fact we are matching to the primary key, stored procedures can also be used to return multiple rows, or a single value.

Stored procedures can be called from within SQL server. To call this stored procedure from the SQL server command line or from another stored procedure you would use the following:

```
exec HumanResources.uspFindEmployee 3
```

Stored procedures can also be called from within a programming language. Each language, such as **PHP** or **C#**, has its specific methods for doing so.

Stored Procedures in SQL SERVER DBMS

Creating Table EMPLOYEE

CREATE TABLE employee(
 eid **INTEGER** NOT NULL **PRIMARY KEY**,
first_name **VARCHAR**(10),
 last_name **VARCHAR**(10),
 salary **int**
)
Now insert some values in the table and using select statement to select a table.
INSERT INTO employee VALUES (2, 'Keza', 'Nicole',247000);
INSERT INTO employee VALUES (4, 'Kalisa' , 'Ben', 55600);
INSERT INTO employee VALUES (5, 'Rusanganwa', 'Paul', 4000);

INSERT INTO employee VALUES (6, 'Mugisha', 'Valois', 34500);

Syntax in MySQL DBMS	Syntax in SQL SERVER DBMS
DELIMITER// CREATE PROCEDURE procedureName ( ) BEGIN SQL STATEMENT END // DELIMITER ;	CREATE PROCEDURE PROCEDURE_NAME <@Param1 data type, <@Param2 data type AS BEGIN sql operation (eg: SELECT @Param1, @Param2 from tablename) END GO

INSERT stored procedure

```
create PROCEDURE EmployeeInsert
(
@id INTEGER,
@first_name VARCHAR(10),
@last_name VARCHAR(10),
@salary int
)
AS
BEGIN
insert into employee (eid,first_name,last_name,salary) values( @id, @first_name, @last_name,
@salary)
END
```

Execute insert SP:

```
exec EmployeeInsert  @id=1, @first_name='Kigabo', @last_name='Sam', @salary=460000;
```

UPDATE stored procedure

```
create PROCEDURE EmployeeUpdate
(
@eid INTEGER,
@first_name VARCHAR(10),
@last_name VARCHAR(10),
@salary int
)
AS
BEGIN
UPDATE employee SET
First_name = @first_name, last_name = @last_name, salary = @salary
WHERE eid = @eid
END
```

Execute update SP:

```
exec EmployeeUpdate @first_name='Uwimana', @last_name='Cynthia', @salary=300000,
@eid=4
```

DELETE stored procedure

```
create PROCEDURE EmployeeDelete
(
@eid INTEGER
)
AS
BEGIN
DELETE FROM employee WHERE eid = @eid
END
```

EXECUTE DELETE SP: EXEC EmployeeDelete @EID=2;

SELECT stored procedure

```
create PROCEDURE EmployeeSelect
(
@eid INTEGER
)
AS
BEGIN
select * from employee where eid=@eid
END
```

EXECUTE SELECT SP: EXEC EmployeeSelect @EID=4;

Benefits of using stored procedures

The benefits of using stored procedures in SQL Server rather than application code stored locally on client computers include:

- They allow modular programming.
- They allow faster execution.
- They can reduce network traffic.
- They can be used as a security mechanism.

You can create a stored procedure once, store it in the database, and call it any number of times in your program. Someone who specializes in database programming may create stored procedures; this allows the application developer to concentrate on the code instead of SQL.

You can modify stored procedures independently of the program source code—the application doesn't have to be recompiled when/if the SQL is altered.

2.10.2 INDEX

Description of index

- A database index is a data structure that improves the speed of operations in a table.
- Practically, indexes are also a type of tables, which keep primary key or index field and a pointer to each record into the actual table.
- The users cannot see the indexes, they are just used to speed up queries and will be used by the Database Search Engine to locate records very fast.
- 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.
- The INSERT and UPDATE statements take more time on tables having indexes, whereas the SELECT statements become fast on those tables. The reason is that while doing insert or update, a database needs to insert or update the index values as well.
- **Note:** Updating a table with indexes takes more time than updating a table without (because the indexes also need an update). So, only create indexes on columns that will be frequently searched against.

CREATING SIMPLE AND UNIQUE INDEX

You can create a unique index on a table. A unique index means that two rows cannot have the same index value. Here is the syntax to create an Index on a table.
Creating an index involves the **CREATE INDEX** statement, which allows you to name the index, to specify the table and which column or columns to index, and to indicate whether the index is in an ascending or descending order.

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

You can use one or more columns to create an index.
For example, we can create an index on persons table using firstname.

CREATE UNIQUE INDEX fname_INDEX **ON** persons (firstname)

You can create a simple index on a table. Just omit the UNIQUE keyword from the query to create a simple index. A Simple index allows duplicate values in a table.
If you want to index the values in a column in a descending order, you can add the reserved word DESC after the column name.
CREATE UNIQUE INDEX fname_INDEX **ON** persons (firstname DESC)

ALTER COMMAND TO ADD AND DROP INDEX

DROP INDEX Statement
In MySQL

- ALTER TABLE*table_name* DROP INDEX*index_name*;

ADD an INDEX

- **ALTER TABLE tbl_name ADD UNIQUE index_name (column_list)** − This statement creates an index for which the values must be unique (except for the NULL values, which may appear multiple times).
- **ALTER TABLE tbl_name ADD INDEX index_name (column_list)**− This adds an ordinary index in which any value may appear more than once.

SHOW INDEXES CREATED ON A TABLE
To display all indexes of the persons table: SHOW INDEXES FROM PERSONS;

USING INDEX
Syntax:
SELECT select_list FROM table_name USE INDEX (index_list or index_name) WHERE condition;

- In this syntax, the USE INDEX instructs the query optimizer to use one of the named indexes to find rows in the table.
- Notice that when you recommend the indexes to use, the query optimizer may either decide to use them or not depending on the query plan that it comes up with.

Advantages of indexes

- Their use in queries usually results in much better performance.
- They make it possible to quickly retrieve (fetch) data.
- They can be used for sorting. A post-fetch-sort operation can be eliminated.
- Unique indexes guarantee uniquely identifiable records in the database.

Disadvantages of indexes

- They decrease performance on inserts, updates, and deletes.
- They take up space (this increases with the number of fields used and the length of the fields).
- Some databases will mono-case (case insensitive) values in fields that are indexed.

You should only create indexes when they are actually needed.

Take care not to add an index on something that has already been indexed. If you need a more detailed index, you can add fields to an existing index as long as it is not a unique index.

Unit 3: INTERACT WITH DATABASE

3.1 Data file formats

A file format is a standard way that information is encoded for storage in a computer file. It specifies how bits are used to encode information in a digital storage medium.

There are two different ways of storing data in a file – as text or binary data. Text-based file formats, such as XML and HTML, store data as plain text, which means the file content can be viewed in a text editor. Binary files, on the other hand, can only be opened with a program that recognizes the specific file format. While some binary files can be opened in a text editor, most of the data will appear corrupted and meaningless.

DB files can be created using database software like Microsoft Access, Filemaker Pro and others. DB files can be exported or imported in different formats like .CSV format.

A data file could be any file, but for the purpose of this list, we've listed the most common data files that relate to data used for a database, importing, and exporting.

DB file	Description
.sql	Database file. A SQL file is a file written in SQL (Structured Query Language). It contains SQL code used to modify the contents of a relational database.
.CSV	Short for "Comma separated value". Files ending in the CSV file extension are generally used to exchange data. CSV files are designed to be a way to easily export data and import it into other programs. Files in the CSV format can be imported to and exported from programs that store data in tables, such as Microsoft Excel or OpenOffice like ODS
.xls	Microsoft Excel file
.xlsx	Microsoft Excel Open XML spreadsheet file
.BAK	Backup file. A BAK file is a backup of another document or file, commonly created automatically by software programs or by the operating system. It typically contains a copy of the original file and can be restored to the original by replacing the ".bak" extension with the original extension. When a program is about to overwrite an existing file (for example, when the user saves the document they are working on), the program may first make a copy of the existing file, with .bak appended to the filename.

.CSV file

Short for comma-separated values, CSV is tabular data that has been saved as plaintext data separated by commas.
For example, if you had a table similar to the example below that data would be converted to the CSV data shown below the table.

Data1	Data2	Data3
Example1	Example2	Example3
Example1	Example2	Example3

CSV Data

```
Data1, Data2, Data3
Example1, Example2, Example3
Example1, Example2, Example3
```

As can be seen in the above example, each row is a new line, and each column is separated with a comma. Many online services, such as an online bank, allow its users to export tabular data from the website into a CSV file. These files can then be opened and viewed offline using a spreadsheet program, such as Microsoft Excel.

Why are CSV files used?

There are two primary reasons CSV files are used online:

- CSV files are plain-text files, which makes them easy for the website developer to create.
- Because the CSV is plain-text it makes the data easy to import into any spreadsheet program or database regardless of what type of computer or software program you are using.

How to Create A CSV File?

A CSV is a text file, so it can be created and edited using any text editor. More frequently, however, a CSV file is created by exporting (File Menu -> Export) a spreadsheet or database in the program that created it. Below are steps to create a CSV file in Notepad, Microsoft Excel, OpenOffice Calc, and Google Docs.

Steps to create a CSV file in Notepad

To create a CSV file with a text editor, first choose your favorite text editor, such as Notepad or vim, and open a new file. Then enter the text data you want the file to contain, separating each value with a comma and each row with a new line.

```
Title1,Title2,Title3
one,two,three
example1,example2,example3
```

Save this file with the extension .csv. You can then open the file using Microsoft Excel or another spreadsheet program. It would create a table of data similar to the following:

Title1	**Title2**	**Title3**
One	Two	three
example1	example2	example3

Steps to create a CSV file in Microsoft Excel

- To create a **CSV** file using Microsoft Excel, launch Excel and then open the file you want to save in **CSV** format.
- Once open, click File and choose Save As. Under Save as type, select CSV (Comma delimited).
- After you save the file, you are free to open it up in a text editor to view it or edit it manually.

Steps to create a CSV file in Open Office

- To create a CSV file using OpenOffice Calc, launch Calc and open the file you want to save as a CSV file. For example, below is the data contained in our example Calc worksheet.

Item	Cost	Sold	Profit
Keyboard	$10.00	$16.00	$6.00
Monitor	$80.00	$120.00	$40.00
Mouse	$5.00	$7.00	$2.00
		Total	$48.00

- Once open, click **File**, choose the **Save As** option, and for the **Save as type** option, select ***Text CSV (.csv) (*.csv)***.
- After you save the file, if you were to open the CSV file in a text editor, such as Notepad, the CSV file should resemble the example below.

```
Item,Cost,Sold,Profit
Keyboard,$10.00,$16.00,$6.00
Monitor,$80.00,$120.00,$40.00
Mouse,$5.00,$7.00,$2.00
,,Total,$48.00
```

Just as in our Excel example, the two commas at the beginning of the last line are necessary to make sure the fields correspond from row to row. Do not remove them!

Steps to create a CSV file inGoogle Docs

Open Google Docs and open the spreadsheet file you want to save as a CSV file.
Click **File**, **Download as**, and then select **CSV (current sheet)**.

3.2 Correlate data between external format and database

Correlate definition: to bring into reciprocal relation; establish in orderly connection. Example to correlate expenses and income.

To correlate data between external format and database, we keep considering:

- Analyzing data types compatibility: Numeric, Date, String
- Analyzing size of data.

3.3: Execute import of data from external source

PRACTICE!