**SQL**

What Is SQL?

Introduction to SQL language

SQL is a programming language designed to manage data stored in a relational database management system (RDBMS).

SQL stands for the structured query language. It is pronounced as /ˈɛs kjuː ˈɛl/ or /ˈsiːkwəl/.

SQL consists of a data definition language (DDL), data manipulation language (DML), and a data control language (DCL).

* The data definition language deals with the schema creation and modification

e.g., CREATE TABLE statement allows you to create a new table in the database and the ALTER TABLE statement changes the structure of an existing table.

* The data manipulation language provides the constructs to query data such as the SELECT statement and to update the data such as INSERT, UPDATE, and DELETE statements.
* The data control language consists of the statements that deal with the user authorization and security such as GRANT and REVOKE statements.

SQL Standard

SQL was one of the first commercial database languages since 1970. Since then different database vendors implemented SQL in their products with some variations. To bring greater conformity between the vendors, the American Standards Institute (ANSI) published the first SQL standard in 1986.

ANSI then updated the SQL standard in 1992, known as SQL92 and SQL2, and again in 1999 as SQL99 and SQL3. Every time, ANSI added new features and commands into the SQL language.

The SQL Standard is now maintained by both ANSI and International Standards Organization as ISO/IEC 9075 standard. The latest release standard is SQL: 2011.

The SQL standard formalizes SQL syntax structures and behaviors across database products. It becomes even more important to the open-source databases such

as MySQL and PostgreSQL where the RDBMS are developed mainly by the communities rather than big corporations.

SQL Dialects

The community constantly requests new features and capabilities that do not exist in the SQL standard yet, therefore, even with the SQL standard in place, there are many SQL dialects in various database products.

Because ANSI and ISO have not yet developed these important features, RDBMS vendors (or communities) are free to invent their own new syntax structure.

The following are the most popular dialects of SQL:

* PL/SQL stands for procedural language/SQL. It is developed by Oracle for the Oracle Database.
* Transact-SQL or T-SQL is developed by Microsoft for Microsoft SQL Server.
* PL/pgSQL stands for Procedural Language/PostgreSQL that consists of SQL dialect and extensions implemented in PostgreSQL
* MySQL has its own procedural language since version 5. Note that MySQL was acquired by Oracle.

We will explain the SQL syntax structures and behaviors that are valid across the databases. We also will discuss the exceptions if they exist in a particular database.

SQL Syntax

SQL is a declarative language, therefore, its syntax reads like a natural language. An SQL statement begins with a verb that describes the action, for example, SELECT, INSERT, UPDATE or DELETE. Following the verb are the subject and predicate.

A predicate specifies conditions that can be evaluated as true, false, or unknown. See the following SQL statement:

SELECT first\_name FROM employees WHERE

YEAR(hire\_date) = 2000;

As you see, it reads like a normal sentence.

Get the first names of employees who were hired in 2000.

The SELECT first\_name, FROM employees, and WHERE are clauses in the SQL statement. Some clauses are mandatory e.g., the SELECT and FROM clauses whereas others are optional such as the WHERE clause.

**SQL Syntax**



Because SQL was designed specifically for the non-technical people in mind, it is very simple and easy to understand. To write an SQL statement, you just need to tell what you want instead of how you want it like other imperative languages such as PHP, Java, and C++.

SQL is a user-friendly language because it is mainly for the users who perform ad-hoc queries and generate reports.

Nowadays, SQL is used by the highly technical people like data analysts, data scientists, developers, and database administrators.

## SQL commands

SQL is made up of many commands. Each SQL command is typically terminated with a semicolon (;). For example, the following are two different SQL commands separated by a semicolon (;).

SELECT first\_name, last\_name FROM employees;

DELETE FROM employees WHERE

hire\_date < '1990-01-01';

SQL uses the semicolon (;) to mark the end of a command.

Each command is composed of tokens that can be literals, keywords, identifiers, or expressions. Tokens are separated by space, tabs, or newlines.

## Literals

Literals are explicit values which are also known as constants. SQL provides three kinds of literals: string, numeric, and binary.

1. String literal consists of one or more alphanumeric characters surrounded by single quotes, for example:

'John'

'1990-01-01'

'50'

50 is a number. However, if you surround it with single quotes e.g., '50', SQL treats it as a string literal.

Typically, SQL is case sensitive with respect to string literals, so the value 'John' is not the same as 'JOHN'.

1. Numeric literals are the integer, decimal, or scientific notation, for example:

200

-5 6.0221415E23

1. SQL represents binary value using the notation x'0000', where each digit is hexadecimal value,

for example:

x'01'

x'0f0ff'

## Keywords

SQL has many keywords that have special meanings such as SELECT, INSERT, UPDATE, DELETE, and DROP. These keywords are the reserved words, therefore, you cannot use them as the name of tables, columns, indexes, views, stored procedures, triggers, or other database objects.

## Identifiers

Identifiers refer to specific objects in the database such as tables, columns, indexes, etc. SQL is case-insensitive with respect to keywords and identifiers.

The following statements are equivalent.

Select \* From employees; SELECT \* FROM EMPLOYEES;

select \* from employees;

SELECT \* FROM employees;

To make the SQL commands more readable and clear, we will use the SQL keywords in

uppercase and identifiers in lower case.

## Comments

To document SQL statements, you use the SQL comments. When parsing SQL statements with comments, the database engine ignores the characters in the comments.

A comment is denoted by two consecutive hyphens ( --) that allow you to comment the remaining line. See the following example.

SELECT employee\_id, salary FROM employees WHERE salary < 3000;-- employees with low salary

This is an SQL comment.

-- employees with low salary

To document the code that can span multiple lines, you use the multiline C-style notation ( /\*\*/)

as the shown in the following statement:

/\* increase 5% for employees whose salary is less than 3,000 \*/ UPDATE employees

SET

salary = salary \* 1.05 WHERE

salary < 3000;

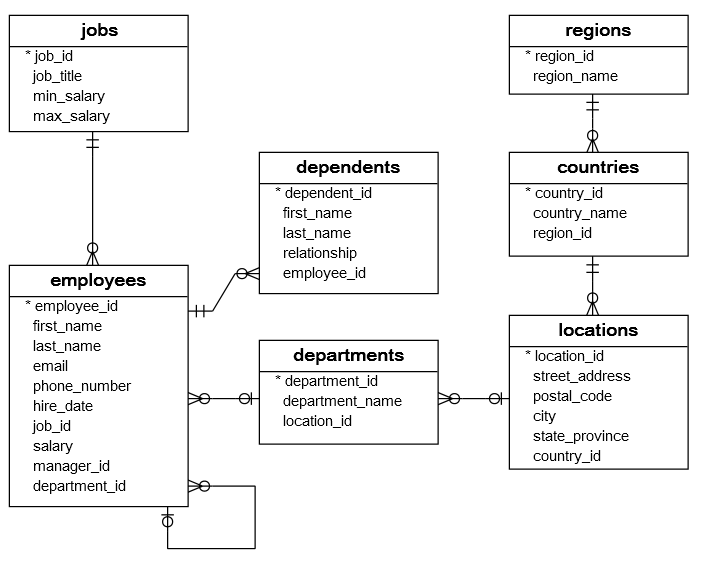
We have introduced you to the SQL syntax that helps you understand each component of an SQL

statement.

To learn the SQL language more effectively, you need to have a good sample database to practice with. We will introduce you to a simple SQL database.

SQL Sample Database

The following database diagram illustrates the HR sample database:



The HR sample database has seven tables:

* The *employees* table stores the data of employees.
* The *jobs* table stores the job data including job title and salary range.
* The *departments* table stores department data.
* The *dependents* table stores the employee’s dependents.
* The *locations* table stores the location of the departments of the company.
* The *countries* table stores the data of countries where the company is doing business.
* The *regions* table stores the data of regions such as Asia, Europe, America, and the Middle East and Africa. The countries are grouped into regions.

The following picture shows the table names and their records.

|  |  |
| --- | --- |
| Table | Rows |
| employees | 40 |
| dependents | 30 |
| departments | 11 |
| jobs | 11 |
| locations | 7 |
| countries | 25 |
| regions | 4 |

Typically, you need to install a Relational Database Management System (RDBMS) to work with SQL.

If you have worked with an RDBMS such as MySQL, PostgreSQL, Oracle Database, and SQL Server, you can use the following script to create the sample database in one of these databases.

In case you don’t have a database system to practice, you can quickly use our SQL online tool to execute the SQL statements in your web browser.

## MySQL

The following SQL script creates the HR sample database in MySQL: Create HR Sample Database in MySQL

The following script allows you to insert data into the tables in MySQL: Load HR Data in MySQL

## PostgreSQL

The following script creates the HR sample database structure in PostgreSQL. Create HR Sample Database in PostgreSQL

The following script allows you to insert data into the tables in PostgreSQL: Load HR Data in PostgreSQL

## Microsoft SQL Server

The following script creates the HR sample database structure in Microsoft SQL Server.

Create HR Sample Database in SQL Server

The following script allows you to insert data into the tables: Load HR Data in SQL Server

## Oracle Database (>12c)

The following script creates the HR sample database structure in Oracle Database 12c.

Create HR Sample Database in Oracle

The following script inserts data into the tables in the Oracle database: Load HR Data in Oracle Database

## SQLite

The following script creates the HR sample database structure in SQLite.

Create HR Sample Database in SQLite

The following script inserts data into the tables in the SQLite: Create HR Sample Database in SQLite

## Removing tables

The following is the script that drops all tables in case you want to refresh the sample database.

Drop All Tables

## DDL:

The SQL CREATE TABLE Statement

The CREATE TABLE statement is used to create a new table in a database. Syntax

CREATE TABLE *table\_name* (

*column1 datatype*, *column2 datatype*, *column3 datatype*,

....

);

The column parameters specify the names of the columns of the table.

The datatype parameter specifies the type of data the column can hold (e.g. varchar, integer, date, etc.).

## SQL 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)

);

The PersonID column is of type int and will hold an integer.

The LastName, FirstName, Address, and City columns are of type varchar and will hold characters, and the maximum length for these fields is 255 characters.

The empty "Persons" table will now look like this:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| PersonID | LastName | FirstName | Address | City |
|  |  |  |  |  |

The SQL DROP TABLE Statement

The DROP TABLE statement is used to drop an existing table in a database. Syntax

DROP TABLE *table\_name*;

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

## SQL DROP TABLE Example

The following SQL statement drops the existing table "Shippers":

DROP TABLE Shippers;

## SQL TRUNCATE TABLE

The TRUNCATE TABLE statement is used to delete the data inside a table, but not the table itself.

## Syntax

TRUNCATE TABLE *table\_name*;

**SQL ALTER TABLE Statement**

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

To add a column in a table, use the following syntax:

ALTER TABLE *table\_name*

ADD *column\_name datatype*;

The following SQL adds an "Email" column to the "Customers" table:

## Example

ALTER TABLE Customers ADD Email varchar(255);

**ALTER TABLE - DROP COLUMN**

To delete a column in a table, use the following syntax (notice that some database systems don't allow deleting a column):

ALTER TABLE *table\_name*

DROP COLUMN *column\_name*;

The following SQL deletes the "Email" column from the "Customers" table:

## Example

ALTER TABLE Customers DROP COLUMN Email;

**ALTER TABLE - RENAME COLUMN**

To rename a column in a table, use the following syntax:

## Example

ALTER TABLE *table\_name*

RENAME COLUMN *old\_name* to *new\_name*;

**ALTER TABLE - ALTER/MODIFY DATATYPE**

To change the data type of a column in a table, use the following syntax:

## Example

**SQL Server / MS Access:**

ALTER TABLE *table\_name*

ALTER COLUMN *column\_name datatype*;

## My SQL / Oracle (prior version 10G):

ALTER TABLE *table\_name*

MODIFY COLUMN *column\_name datatype*;

**Oracle 10G and later:**

ALTER TABLE *table\_name*

MODIFY *column\_name datatype*;

## SQL ALTER TABLE Example

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 |

Now we want to add a column named "DateOfBirth" in the "Persons" table. We use the following SQL statement:

## Example

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. For a complete reference of all the data types available in MS Access, MySQL, and SQL Server, go to our complete Data Types reference.

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:

## Example

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:

## Example

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 | |

|  |  |  |
| --- | --- | --- |
|  | **RENAME Command**  RENAME is a DDL command which is used to change the name of the database table.  **Syntax of RENAME command Example** |  |
| 1. | RENAME **TABLE** Old\_Table\_Name **TO** New\_Table\_Name; |  |
|  | **Example** |  |
| 1. | RENAME **TABLE** Student **TO** Student\_Details ; |  |
|  | This query changes the name of the table from Student to Student\_Details. |  |
|  | **DML Commands in SQL** |  |
|  | **DML is an abbreviation of Data Manipulation Language.** |  |
|  | The DML commands in Structured Query Language change the data present in database. We can easily access, store, modify, update and delete the existing records database using DML commands. | the SQL from the |
|  | **Following are the four main DML commands in SQL:** |  |
|  | 1. SELECT Command |  |
|  | 2. INSERT Command |  |
|  | 3. UPDATE Command |  |
|  | 4. DELETE Command |  |

## SELECT DML Command

SELECT is the most important data manipulation command in Structured Query Language. The SELECT command shows the records of the specified table. It also shows the particular record of a particular column by using the WHERE clause.

## Syntax of SELECT DML command

1. **SELECT** column\_Name\_1, column\_Name\_2, ….., column\_Name\_N **FROM** Name\_of\_table;

Here, **column\_Name\_1, column\_Name\_2, ….., column\_Name\_N** are the names of those columns whose data we want to retrieve from the table.

If we want to retrieve the data from all the columns of the table, we have to use the following SELECT command:

1. **SELECT** \* **FROM** table\_name;

## Examples of SELECT Command

**Example 1: This example shows all the values of every column from the table.**

1. **SELECT** \* **FROM** Student;

This SQL statement displays the following values of the student table:

|  |  |  |
| --- | --- | --- |
| **Student\_ID** | **Student\_Name** | **Student\_Marks** |
| BCA1001 | Abhay | 85 |
| BCA1002 | Anuj | 75 |
| BCA1003 | Bheem | 60 |
| BCA1004 | Ram | 79 |
| BCA1005 | Sumit | 80 |

**Example 2: This example shows all the values of a specific column from the table.**

1. **SELECT** Emp\_Id, Emp\_Salary **FROM** Employee;

This SELECT statement displays all the values of **Emp\_Salary** and **Emp\_Id** column of **Employee** table:

|  |  |
| --- | --- |
| **Emp\_Id** | **Emp\_Salary** |
| 201 | 25000 |
| 202 | 45000 |
| 203 | 30000 |
| 204 | 29000 |
| 205 | 40000 |

**Example 3: This example describes how to use the WHERE clause with the SELECT DML command.**

Let's take the following Student table:

|  |  |  |
| --- | --- | --- |
| **Student\_ID** | **Student\_Name** | **Student\_Marks** |
| BCA1001 | Abhay | 80 |
| BCA1002 | Ankit | 75 |
| BCA1003 | Bheem | 80 |
| BCA1004 | Ram | 79 |

|  |  |  |
| --- | --- | --- |
| BCA1005 | Sumit | 80 |

If you want to access all the records of those students whose marks is 80 from the above table, then you have to write the following DML command in SQL:

1. **SELECT** \* **FROM** Student **WHERE** Stu\_Marks = 80;

The above SQL query shows the following table in result:

|  |  |  |
| --- | --- | --- |
| **Student\_ID** | **Student\_Name** | **Student\_Marks** |
| BCA1001 | Abhay | 80 |
| BCA1003 | Bheem | 80 |
| BCA1005 | Sumit | 80 |

## INSERT DML Command

INSERT is another most important data manipulation command in Structured Query Language, which allows users to insert data in database tables.

## Syntax of INSERT Command

1. **INSERT INTO** TABLE\_NAME ( column\_Name1 , column\_Name2 , column\_Name3 , colu

mn\_NameN ) **VALUES** (value\_1, value\_2, value\_3, value\_N ) ;

**Examples of INSERT Command**

**Example 1: This example describes how to insert the record in the database table.**

Let's take the following student table, which consists of only 2 records of the student.

|  |  |  |  |
| --- | --- | --- | --- |
| **Stu\_Id** | **Stu\_Name** | **Stu\_Marks** | **Stu\_Age** |
| 101 | Ramesh | 92 | 20 |
| 201 | Jatin | 83 | 19 |

Suppose, you want to insert a new record into the student table. For this, you have to write the following DML INSERT command:

1. **INSERT INTO** Student (Stu\_id, Stu\_Name, Stu\_Marks, Stu\_Age) **VALUES** (104, Anmol, 89, 19);

## UPDATE DML Command

UPDATE is another most important data manipulation command in Structured Query Language, which allows users to update or modify the existing data in database tables.

## Syntax of UPDATE Command

1. **UPDATE** Table\_name **SET** [column\_name1= value\_1, ….., column\_nameN = value\_N] **WHE RE** CONDITION;

Here, 'UPDATE', 'SET', and 'WHERE' are the SQL keywords, and 'Table\_name' is the name of the table whose values you want to update.

## Examples of the UPDATE command

**Example 1: This example describes how to update the value of a single field.**

Let's take a Product table consisting of the following records:

|  |  |  |  |
| --- | --- | --- | --- |
| **Product\_Id** | **Product\_Name** | **Product\_Price** | **Product\_Quantity** |
| P101 | Chips | 20 | 20 |
| P102 | Chocolates | 60 | 40 |
| P103 | Maggi | 75 | 5 |
| P201 | Biscuits | 80 | 20 |
| P203 | Namkeen | 40 | 50 |

Suppose, you want to update the Product\_Price of the product whose Product\_Id is P102. To do this, you have to write the following DML UPDATE command:

1. **UPDATE** Product **SET** Product\_Price = 80 **WHERE** Product\_Id = 'P102' ;

**Example 2: This example describes how to update the value of multiple fields of the database table.**

Let's take a Student table consisting of the following records:

|  |  |  |  |
| --- | --- | --- | --- |
| **Stu\_Id** | **Stu\_Name** | **Stu\_Marks** | **Stu\_Age** |
| 101 | Ramesh | 92 | 20 |
| 201 | Jatin | 83 | 19 |
| 202 | Anuj | 85 | 19 |
| 203 | Monty | 95 | 21 |
| 102 | Saket | 65 | 21 |
| 103 | Sumit | 78 | 19 |
| 104 | Ashish | 98 | 20 |

Suppose, you want to update Stu\_Marks and Stu\_Age of that student whose Stu\_Id is 103 and

202. To do this, you have to write the following DML Update command:

1. **UPDATE** Student **SET** Stu\_Marks = 80, Stu\_Age = 21 **WHERE** Stu\_Id = 103 AND Stu\_Id = 2 02;

## DELETE DML Command

DELETE is a DML command which allows SQL users to remove single or multiple existing records from the database tables.

This command of Data Manipulation Language does not delete the stored data permanently from the database. We use the WHERE clause with the DELETE command to select specific rows from the table.

## Syntax of DELETE Command

1. **DELETE FROM** Table\_Name **WHERE** condition;

**Examples of DELETE Command**

## Example 1: This example describes how to delete a single record from the table.

Let's take a Product table consisting of the following records:

|  |  |  |  |
| --- | --- | --- | --- |
| **Product\_Id** | **Product\_Name** | **Product\_Price** | **Product\_Quantity** |
| P101 | Chips | 20 | 20 |
| P102 | Chocolates | 60 | 40 |
| P103 | Maggi | 75 | 5 |
| P201 | Biscuits | 80 | 20 |
| P203 | Namkeen | 40 | 50 |

Suppose, you want to delete that product from the Product table whose Product\_Id is P203. To do this, you have to write the following DML DELETE command:

1. **DELETE FROM** Product **WHERE** Product\_Id = 'P202' ;

## Example 2: This example describes how to delete the multiple records or rows from the database table.

Let's take a Student table consisting of the following records:

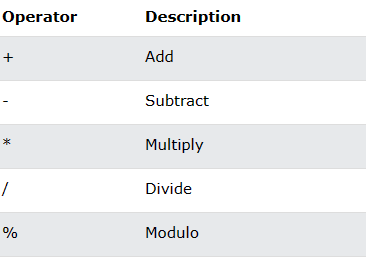
|  |  |  |  |
| --- | --- | --- | --- |
| **Stu\_Id** | **Stu\_Name** | **Stu\_Marks** | **Stu\_Age** |
| 101 | Ramesh | 92 | 20 |
| 201 | Jatin | 83 | 19 |
| 202 | Anuj | 85 | 19 |
| 203 | Monty | 95 | 21 |
| 102 | Saket | 65 | 21 |
| 103 | Sumit | 78 | 19 |
| 104 | Ashish | 98 | 20 |

Suppose, you want to delete the record of those students whose Marks is greater than 70. To do this, you have to write the following DML Update command:

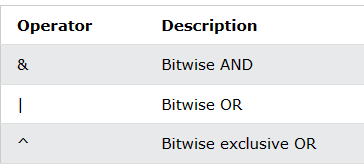
1. **DELETE FROM** Student **WHERE** Stu\_Marks > 70 ;

## SQL Operators

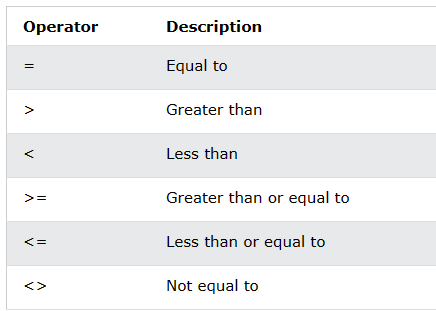
**SQL Arithmetic Operators**



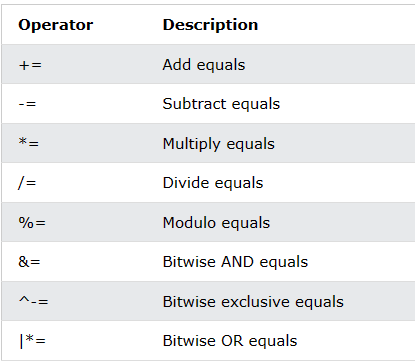
## SQL Bitwise Operators



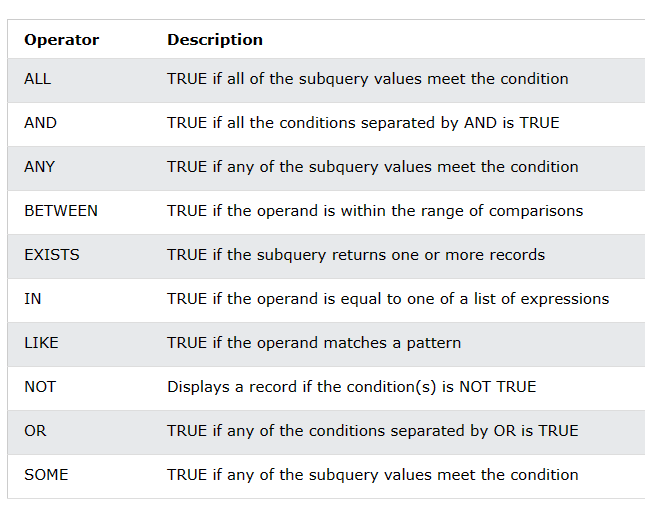
**SQL Comparison Operators**



## SQL Compound Operators



**SQL Logical Operators**



## SQL Create Constraints

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*,

....

);

**SQL Constraints**

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

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 - Prevents actions that would destroy links between tables
* CHECK - Ensures that the values in a column satisfies a specific condition
* DEFAULT - Sets a default value for a column if no value is specified
* CREATE INDEX - Used to create and retrieve data from the database very quickly

## SQL NOT NULL Constraint

By default, a column can hold NULL values.

The NOT NULL constraint enforces a column to NOT accept NULL values.

This enforces a field to always contain a value, which means that you cannot insert a new record, or update a record without adding a value to this field.

## SQL NOT NULL on CREATE TABLE

The following SQL ensures that the "ID", "LastName", and "FirstName" columns will NOT accept NULL values when the "Persons" table is created:

## Example[Get your own SQL Server](https://www.w3schools.com/spaces/)

CREATE TABLE Persons ( ID int NOT NULL,

LastName varchar(255) NOT NULL, FirstName varchar(255) NOT NULL, Age int

);

**SQL NOT NULL on ALTER TABLE**

To create a NOT NULL constraint on the "Age" column when the "Persons" table is already created, use the following SQL:

## SQL Server / MS Access:

ALTER TABLE Persons

ALTER COLUMN Age int NOT NULL;

**My SQL / Oracle (prior version 10G):**

ALTER TABLE Persons

MODIFY COLUMN Age int NOT NULL;

## Oracle 10G and later:

ALTER TABLE Persons MODIFY Age int NOT NULL;

**SQL UNIQUE Constraint**

The UNIQUE constraint ensures that all values in a column are different.

Both the UNIQUE and PRIMARY KEY constraints provide a guarantee for uniqueness for a column or set of columns.

A PRIMARY KEY constraint automatically has a UNIQUE constraint.

However, you can have many UNIQUE constraints per table, but only one PRIMARY KEY constraint per table.

## SQL UNIQUE Constraint on CREATE TABLE

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

## SQL Server / Oracle / MS Access:

CREATE TABLE Persons (

ID int NOT NULL UNIQUE,

LastName varchar(255) NOT NULL, FirstName varchar(255),

Age int

);

**MySQL:**

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:

## MySQL / SQL Server / Oracle / MS Access:

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:

## MySQL / SQL Server / Oracle / MS Access:

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:

## MySQL / SQL Server / Oracle / MS Access:

ALTER TABLE Persons

ADD CONSTRAINT UC\_Person UNIQUE (ID,LastName);

**DROP a UNIQUE Constraint**

To drop a UNIQUE constraint, use the following SQL:

## MySQL:

ALTER TABLE Persons DROP INDEX UC\_Person;

**SQL Server / Oracle / MS Access:**

ALTER TABLE Persons

DROP CONSTRAINT UC\_Person;

## SQL PRIMARY KEY Constraint

The PRIMARY KEY constraint uniquely identifies each record in a table. Primary keys must contain UNIQUE values, and cannot contain NULL values.

A table can have only ONE primary key; and in the table, this primary key can consist of single or multiple columns (fields).

## SQL PRIMARY KEY on CREATE TABLE

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

## MySQL:

CREATE TABLE Persons ( ID int NOT NULL,

LastName varchar(255) NOT NULL, FirstName varchar(255),

Age int,

PRIMARY KEY (ID)

);

**SQL Server / Oracle / MS Access:**

CREATE TABLE Persons (

ID int NOT NULL PRIMARY KEY,

LastName varchar(255) NOT NULL, FirstName varchar(255),

Age int

);

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

## MySQL / SQL Server / Oracle / MS Access:

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:

## MySQL / SQL Server / Oracle / MS Access:

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:

## MySQL / SQL Server / Oracle / MS Access:

ALTER TABLE Persons

ADD CONSTRAINT PK\_Person PRIMARY KEY (ID,LastName);

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

## DROP a PRIMARY KEY Constraint

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

## MySQL:

ALTER TABLE Persons DROP PRIMARY KEY;

**SQL Server / Oracle / MS Access:**

ALTER TABLE Persons

DROP CONSTRAINT PK\_Person;

## SQL FOREIGN KEY Constraint

The FOREIGN KEY constraint is used to prevent actions that would destroy links between tables.

A FOREIGN KEY is a field (or collection of fields) in one table that refers to the PRIMARY KEY in another table.

The table with the foreign key is called the child table, and the table with the primary 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 prevents invalid data from being inserted into the foreign key column, because it has to be one of the values contained in the parent table.

## SQL FOREIGN KEY on CREATE TABLE

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

## MySQL:

CREATE TABLE Orders (

OrderID int NOT NULL, OrderNumber int NOT NULL, PersonID int,

PRIMARY KEY (OrderID),

FOREIGN KEY (PersonID) REFERENCES Persons(PersonID)

);

**SQL Server / Oracle / MS Access:**

CREATE TABLE Orders (

OrderID int NOT NULL PRIMARY KEY,

OrderNumber int NOT NULL,

PersonID int FOREIGN KEY REFERENCES Persons(PersonID)

);

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

## MySQL / SQL Server / Oracle / MS Access:

CREATE TABLE Orders (

OrderID int NOT NULL, OrderNumber int NOT NULL, PersonID int,

PRIMARY KEY (OrderID),

CONSTRAINT FK\_PersonOrder 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:

## MySQL / SQL Server / Oracle / MS Access:

ALTER TABLE Orders

ADD FOREIGN KEY (PersonID) REFERENCES Persons(PersonID);

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

## MySQL / SQL Server / Oracle / MS Access:

ALTER TABLE Orders

ADD CONSTRAINT FK\_PersonOrder

FOREIGN KEY (PersonID) REFERENCES Persons(PersonID);

**DROP a FOREIGN KEY Constraint**

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

## MySQL:

ALTER TABLE Orders

DROP FOREIGN KEY FK\_PersonOrder;

**SQL Server / Oracle / MS Access:**

ALTER TABLE Orders

DROP CONSTRAINT FK\_PersonOrder;

## 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 column it will allow only certain values for this column.

If you define a CHECK constraint on a table it can limit the values in certain columns based on values in other columns in the row.

## SQL CHECK on CREATE TABLE

The following SQL creates a CHECK constraint on the "Age" column when the "Persons" table is created. The CHECK constraint ensures that the age of a person must be 18, or older:

## MySQL:

CREATE TABLE Persons ( ID int NOT NULL,

LastName varchar(255) NOT NULL, FirstName varchar(255),

Age int,

CHECK (Age>=18)

);

**SQL Server / Oracle / MS Access:**

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:

## MySQL / SQL Server / Oracle / MS Access:

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:

## MySQL / SQL Server / Oracle / MS Access:

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

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

## MySQL / SQL Server / Oracle / MS Access:

ALTER TABLE Persons

ADD CONSTRAINT CHK\_PersonAge CHECK (Age>=18 AND City='Sandnes');

**DROP a CHECK Constraint**

To drop a CHECK constraint, use the following SQL:

## SQL Server / Oracle / MS Access:

ALTER TABLE Persons

DROP CONSTRAINT CHK\_PersonAge;

**MySQL:**

ALTER TABLE Persons

DROP CHECK CHK\_PersonAge;

## SQL CREATE INDEX Statement

The CREATE INDEX statement is used to create indexes in tables.

Indexes are used to retrieve data from the database more quickly than otherwise. The users cannot see the indexes, they are just used to speed up searches/queries.

**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.

## CREATE INDEX Syntax

Creates an index on a table. Duplicate values are allowed:

CREATE INDEX *index\_name*

ON *table\_name* (*column1*, *column2*, ...);

## CREATE UNIQUE INDEX Syntax

Creates a unique index on a table. Duplicate values are not allowed:

CREATE UNIQUE INDEX *index\_name*

ON *table\_name* (*column1*, *column2*, ...);

**Note:** The syntax for creating indexes varies among different databases. Therefore: Check the syntax for creating indexes in your database.

## CREATE INDEX Example

The SQL statement below creates an index named "idx\_lastname" on the "LastName" column in the "Persons" table:

CREATE INDEX idx\_lastname ON Persons (LastName);

If you want to create an index on a combination of columns, you can list the column names within the parentheses, separated by commas:

CREATE INDEX idx\_pname

ON Persons (LastName, FirstName);

## DROP INDEX Statement

The DROP INDEX statement is used to delete an index in a table.

## MS Access:

DROP INDEX *index\_name* ON *table\_name*;

**SQL Server:**

DROP INDEX *table\_name*.*index\_name*;

## DB2/Oracle:

DROP INDEX *index\_name*;

**MySQL:**

ALTER TABLE *table\_name*

DROP INDEX *index\_name*;

## 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. The data type is a guideline for SQL to understand what type of data is expected inside of each column, and it also identifies how SQL will interact with the stored data.

**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!**

## MySQL Data Types (Version 8.0)

In MySQL there are three main data types: string, numeric, and date and time. String Data Types

|  |  |
| --- | --- |
| Data type | Description |
| CHAR(size) | A FIXED length string (can contain letters, numbers, and special characters). The size parameter specifies the column length in  characters - can be from 0 to 255. Default is 1 |
| VARCHAR(size) | A VARIABLE length string (can contain letters, numbers, and  special characters). The size parameter specifies the maximum string length in characters - can be from 0 to 65535 |
| BINARY(size) | Equal to CHAR(), but stores binary byte strings.  The size parameter specifies the column length in bytes. Default is 1 |
| VARBINARY(size) | Equal to VARCHAR(), but stores binary byte strings.  The size parameter specifies the maximum column length in bytes. |
| TINYBLOB | For BLOBs (Binary Large Objects). Max length: 255 bytes |
| TINYTEXT | Holds a string with a maximum length of 255 characters |
| TEXT(size) | Holds a string with a maximum length of 65,535 bytes |
| BLOB(size) | 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(val1, val2, val3, ...) | A string object that can have only one value, chosen from 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. The values are sorted in the order you enter them |
| SET(val1, val2, val3, ...) | A string object that can have 0 or more values, chosen from a list  of possible values. You can list up to 64 values in a SET list |

## Numeric Data Types

|  |  |
| --- | --- |
| **Data type** | **Description** |
| BIT(size) | A bit-value type. The number of bits per value is specified  in size. The size parameter can hold a value from 1 to 64. The default value for size is 1. |
| TINYINT(size) | A very small integer. Signed range is from -128 to 127. Unsigned range is from 0 to 255. The size parameter specifies  the maximum display width (which is 255) |
| BOOL | Zero is considered as false, nonzero values are considered as  true. |
| BOOLEAN | Equal to BOOL |
| SMALLINT(size) | A small integer. Signed range is from -32768 to 32767. Unsigned range is from 0 to 65535. The size parameter  specifies the maximum display width (which is 255) |
| MEDIUMINT(size) | A medium integer. Signed range is from -8388608 to 8388607. Unsigned range is from 0 to 16777215. The size parameter  specifies the maximum display width (which is 255) |
| INT(size) | A medium integer. Signed range is from -2147483648 to 2147483647. Unsigned range is from 0 to 4294967295. The size parameter specifies the maximum display width  (which is 255) |
| INTEGER(size) | Equal to INT(size) |
| BIGINT(size) | A large integer. Signed range is from -9223372036854775808 to 9223372036854775807. Unsigned range is from 0 to 18446744073709551615. The size parameter specifies the  maximum display width (which is 255) |
| FLOAT(size, d) | A floating point number. The total number of digits is specified in size. The number of digits after the decimal point is specified in the d parameter. This syntax is deprecated in MySQL 8.0.17, and it will be removed in future MySQL  versions |
| FLOAT(p) | A floating point number. MySQL uses the p value to determine whether to use FLOAT or DOUBLE for the resulting data type. If p is from 0 to 24, the data type becomes FLOAT().  If p is from 25 to 53, the data type becomes DOUBLE() |
| DOUBLE(size, d) | A normal-size floating point number. The total number of  digits is specified in size. The number of digits after the decimal point is specified in the d parameter |
| DOUBLE PRECISION(size, d) |  |

|  |  |
| --- | --- |
| DECIMAL(size, d) | An exact fixed-point number. The total number of digits is specified in size. The number of digits after the decimal point is specified in the d parameter. The maximum number  for size is 65. The maximum number for d is 30. The default value for size is 10. The default value for d is 0. |
| DEC(size, d) | Equal to DECIMAL(size,d) |

**Note**: All the numeric data types may have an extra option: UNSIGNED or ZEROFILL. If you add the UNSIGNED option, MySQL disallows negative values for the column. If you add the ZEROFILL option, MySQL automatically also adds the UNSIGNED attribute to the column.

## Date and Time Data Types

|  |  |
| --- | --- |
| **Data type** | **Description** |
| DATE | A date. Format: YYYY-MM-DD. The supported range is from '1000-01-  01' to '9999-12-31' |
| DATETIME(*fsp*) | A date and time combination. Format: YYYY-MM-DD hh:mm:ss. The supported range is from '1000-01-01 00:00:00' to '9999-12-31 23:59:59'. Adding DEFAULT and ON UPDATE in the column definition to get  automatic initialization and updating to the current date and time |
| TIMESTAMP(*fsp*) | 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:mm:ss. The supported range is from '1970-01-01 00:00:01' UTC to '2038-01-09 03:14:07' UTC. Automatic initialization and updating to the current date and time can be specified using DEFAULT CURRENT\_TIMESTAMP and ON UPDATE CURRENT\_TIMESTAMP  in the column definition |
| TIME(*fsp*) | A time. Format: hh:mm:ss. The supported range is from '-838:59:59' to  '838:59:59' |
| YEAR | A year in four-digit format. Values allowed in four-digit format: 1901 to  2155, and 0000. |
|  | MySQL 8.0 does not support year in two-digit format. |

**SQL Server Data Types** **String Data Types**

|  |  |  |  |
| --- | --- | --- | --- |
| **Data type** | **Description** | **Max size** | **Storage** |
| char(n) | Fixed width character string | 8,000 characters | Defined width |
| varchar(n) | Variable width character  string | 8,000 characters | 2 bytes + number of  chars |
| varchar(max) | Variable width character  string | 1,073,741,824  characters | 2 bytes + number of  chars |
| text | Variable width character  string | 2GB of text data | 4 bytes + number of  chars |
| nchar | Fixed width Unicode string | 4,000 characters | Defined width x 2 |

|  |  |  |  |
| --- | --- | --- | --- |
| nvarchar | Variable width Unicode  string | 4,000 characters |  |
| nvarchar(max  ) | Variable width Unicode  string | 536,870,912 characters |  |
| ntext | Variable width Unicode  string | 2GB of text data |  |
| binary(n) | Fixed width binary string | 8,000 bytes |  |
| varbinary | Variable width binary string | 8,000 bytes |  |
| varbinary(ma  x) | Variable width binary string | 2GB |  |
| image | Variable width binary string | 2GB |  |

## Numeric Data Types

|  |  |  |
| --- | --- | --- |
| **Data type** | **Description** | **Storage** |
| bit | Integer that can be 0, 1, or NULL |  |
| tinyint | Allows whole numbers from 0 to 255 | 1 byte |
| smallint | Allows whole numbers between -32,768 and 32,767 | 2 bytes |
| int | Allows whole numbers between -2,147,483,648 and 2,147,483,647 | 4 bytes |
| bigint | Allows whole numbers between - 9,223,372,036,854,775,808 and  9,223,372,036,854,775,807 | 8 bytes |
| decimal(p,s) | Fixed precision and scale numbers. | 5-17 bytes |
|  | Allows numbers from -10^38 +1 to 10^38 –1. |  |
|  | The p parameter indicates the maximum total number of digits that can be stored (both to the left and to the right of the decimal point). p must be a value from 1 to 38. Default is 18. |  |
|  | The s parameter indicates the maximum number of digits stored to the right of the decimal point. s must be a value from 0 to p. Default value is 0 |  |
| numeric(p,s) | Fixed precision and scale numbers. | 5-17 bytes |
|  | Allows numbers from -10^38 +1 to 10^38 –1. |  |
|  | The p parameter indicates the maximum total number of digits that can be stored (both to the left and to the right of the decimal point). p must be a value from 1 to 38. Default is 18. |  |
|  | The s parameter indicates the maximum number of digits stored to the right of the decimal point. s must be a value from 0 to p. Default value is 0 |  |
| smallmoney | Monetary data from -214,748.3648 to 214,748.3647 | 4 bytes |
| money | Monetary data from -922,337,203,685,477.5808 to 922,337,203,685,477.5807 | 8 bytes |

|  |  |  |
| --- | --- | --- |
| float(n) | Floating precision number data from -1.79E + 308 to 1.79E + 308. | 4 or 8 bytes |
|  | The n parameter indicates whether the field should hold 4 or 8 bytes. float(24) holds a 4-byte field and float(53) holds an 8-byte field. Default value of n is 53. |  |
| real | Floating precision number data from -3.40E + 38 to 3.40E + 38 | 4 bytes |

**Date and Time Data Types**

|  |  |  |
| --- | --- | --- |
| **Data type** | **Description** | **Storage** |
| datetime | From January 1, 1753 to December 31, 9999 with an accuracy of 3.33 milliseconds | 8 bytes |
| datetime2 | From January 1, 0001 to December 31, 9999 with an accuracy of 100 nanoseconds | 6-8 bytes |
| smalldatetime | From January 1, 1900 to June 6, 2079 with an accuracy of  1 minute | 4 bytes |
| date | Store a date only. From January 1, 0001 to December 31, 9999 | 3 bytes |
| time | Store a time only to an accuracy of 100 nanoseconds | 3-5 bytes |
| datetimeoffset | The same as datetime2 with the addition of a time zone offset | 8-10 bytes |
| timestamp | Stores a unique number that gets updated every time a row gets created or modified. The timestamp value is based upon an internal clock and does not correspond to real time. Each table may have only one timestamp variable |  |

## Other Data Types

|  |  |  |
| --- | --- | --- |
| **Data type** | **Description** | **Storage** |
| Text | Use for text or combinations of text and numbers. 255 characters maximum |  |
| Memo | Memo is used for larger amounts of text. Stores up to 65,536 characters. **Note:** You cannot sort a memo field. However, they are searchable |  |
| Byte | Allows whole numbers from 0 to 255 | 1 byte |
| Integer | Allows whole numbers between -32,768 and 32,767 | 2 bytes |
| Long | Allows whole numbers between -2,147,483,648 and 2,147,483,647 | 4 bytes |
| Single | Single precision floating-point. Will handle most decimals | 4 bytes |
| Double | Double precision floating-point. Will handle most  decimals | 8 bytes |

|  |  |  |
| --- | --- | --- |
| Currency | Use for currency. Holds up to 15 digits of whole dollars, plus 4 decimal places. **Tip:** You can choose which country's currency to use | 8 bytes |
| AutoNumber | AutoNumber fields automatically give each record its own number, usually starting at 1 | 4 bytes |
| Date/Time | Use for dates and times | 8 bytes |
| Yes/No | A logical field can be displayed as Yes/No, True/False, or On/Off. In code, use the constants True and False (equivalent to -1 and 0). **Note:** Null values are not allowed  in Yes/No fields | 1 bit |
| Ole Object | Can store pictures, audio, video, or other BLOBs (Binary Large Objects) | up to 1GB |
| Hyperlink | Contain links to other files, including web pages |  |
| Lookup Wizard | Let you type a list of options, which can then be chosen from a drop-down list | 4 bytes |

**SQL Keywords**

|  |  |
| --- | --- |
| **Keyword** | **Description** |
| ADD | Adds a column in an existing table |
| ADD CONSTRAINT | Adds a constraint after a table is already created |
| ALL | Returns true if all of the subquery values meet the condition |
| ALTER | Adds, deletes, or modifies columns in a table, or changes the data type  of a column in a table |
| ALTER COLUMN | Changes the data type of a column in a table |
| ALTER TABLE | Adds, deletes, or modifies columns in a table |
| AND | Only includes rows where both conditions is true |
| ANY | Returns true if any of the subquery values meet the condition |
| AS | Renames a column or table with an alias |
| ASC | Sorts the result set in ascending order |
| BACKUP  DATABASE | Creates a back up of an existing database |
| BETWEEN | Selects values within a given range |
| CASE | Creates different outputs based on conditions |
| CHECK | A constraint that limits the value that can be placed in a column |
| COLUMN | Changes the data type of a column or deletes a column in a table |
| CONSTRAINT | Adds or deletes a constraint |
| CREATE | Creates a database, index, view, table, or procedure |
| CREATE  DATABASE | Creates a new SQL database |
| CREATE INDEX | Creates an index on a table (allows duplicate values) |
| CREATE OR  REPLACE VIEW | Updates a view |
| CREATE TABLE | Creates a new table in the database |

|  |  |
| --- | --- |
| CREATE  PROCEDURE | Creates a stored procedure |
| CREATE UNIQUE  INDEX | Creates a unique index on a table (no duplicate values) |
| CREATE VIEW | Creates a view based on the result set of a SELECT statement |
| DATABASE | Creates or deletes an SQL database |
| DEFAULT | A constraint that provides a default value for a column |
| DELETE | Deletes rows from a table |
| DESC | Sorts the result set in descending order |
| DISTINCT | Selects only distinct (different) values |
| DROP | Deletes a column, constraint, database, index, table, or view |
| DROP COLUMN | Deletes a column in a table |
| DROP  CONSTRAINT | Deletes a UNIQUE, PRIMARY KEY, FOREIGN KEY, or CHECK  constraint |
| DROP DATABASE | Deletes an existing SQL database |
| DROP DEFAULT | Deletes a DEFAULT constraint |
| DROP INDEX | Deletes an index in a table |
| DROP TABLE | Deletes an existing table in the database |
| DROP VIEW | Deletes a view |
| EXEC | Executes a stored procedure |
| EXISTS | Tests for the existence of any record in a subquery |
| FOREIGN KEY | A constraint that is a key used to link two tables together |
| FROM | Specifies which table to select or delete data from |
| FULL OUTER JOIN | Returns all rows when there is a match in either left table or right table |
| GROUP BY | Groups the result set (used with aggregate functions: COUNT, MAX,  MIN, SUM, AVG) |
| HAVING | Used instead of WHERE with aggregate functions |
| IN | Allows you to specify multiple values in a WHERE clause |
| INDEX | Creates or deletes an index in a table |
| INNER JOIN | Returns rows that have matching values in both tables |
| INSERT INTO | Inserts new rows in a table |
| INSERT INTO  SELECT | Copies data from one table into another table |
| IS NULL | Tests for empty values |
| IS NOT NULL | Tests for non-empty values |
| JOIN | Joins tables |
| LEFT JOIN | Returns all rows from the left table, and the matching rows from the  right table |
| LIKE | Searches for a specified pattern in a column |
| LIMIT | Specifies the number of records to return in the result set |
| NOT | Only includes rows where a condition is not true |
| NOT NULL | A constraint that enforces a column to not accept NULL values |

# SQL

|  |  |
| --- | --- |
| OR | Includes rows where either condition is true |
| ORDER BY | Sorts the result set in ascending or descending order |
| OUTER JOIN | Returns all rows when there is a match in either left table or right table |
| PRIMARY KEY | A constraint that uniquely identifies each record in a database table |
| PROCEDURE | A stored procedure |
| RIGHT JOIN | Returns all rows from the right table, and the matching rows from the  left table |
| ROWNUM | Specifies the number of records to return in the result set |
| SELECT | Selects data from a database |
| SELECT DISTINCT | Selects only distinct (different) values |
| SELECT INTO | Copies data from one table into a new table |
| SELECT TOP | Specifies the number of records to return in the result set |
| SET | Specifies which columns and values that should be updated in a table |
| TABLE | Creates a table, or adds, deletes, or modifies columns in a table, or  deletes a table or data inside a table |
| TOP | Specifies the number of records to return in the result set |
| TRUNCATE TABLE | Deletes the data inside a table, but not the table itself |
| UNION | Combines the result set of two or more SELECT statements (only  distinct values) |
| UNION ALL | Combines the result set of two or more SELECT statements (allows  duplicate values) |
| UNIQUE | A constraint that ensures that all values in a column are unique |
| UPDATE | Updates existing rows in a table |
| VALUES | Specifies the values of an INSERT INTO statement |
| VIEW | Creates, updates, or deletes a view |
| WHERE | Filters a result set to include only records that fulfill a specified  condition |

## SQL – Joins

The SQL JOIN joins two tables based on a common column and selects records that have matching values in these columns.

## Example

-- join the Customers and Orders tables

-- based on the common values of their customer\_id columns SELECT Customers.customer\_id, Customers.first\_name, Orders.item FROM Customers

JOIN Orders

ON Customers.customer\_id = Orders.customer\_id;

Here, the SQL command joins the Customers and Orders tables based on the common values in the customer\_id columns of both tables.

The result set will consist of

* customer\_id and first\_name columns from the Customers table
* item column from the Orders table

## SQL JOIN Syntax

The syntax of the SQL JOIN statement is:

SELECT columns\_from\_both\_tables FROM table1

JOIN table2

ON table1.column1 = table2.column2

Here,

* table1 and table2 are the two tables that are to be joined
* column1 is the column in table1 that is related to column2 in table2

**Note**: There are 4 types of JOINs in SQL. But INNER JOIN and JOIN refer to the same thing.

## Example 1: SQL JOIN

-- join Customers and Orders tables based on

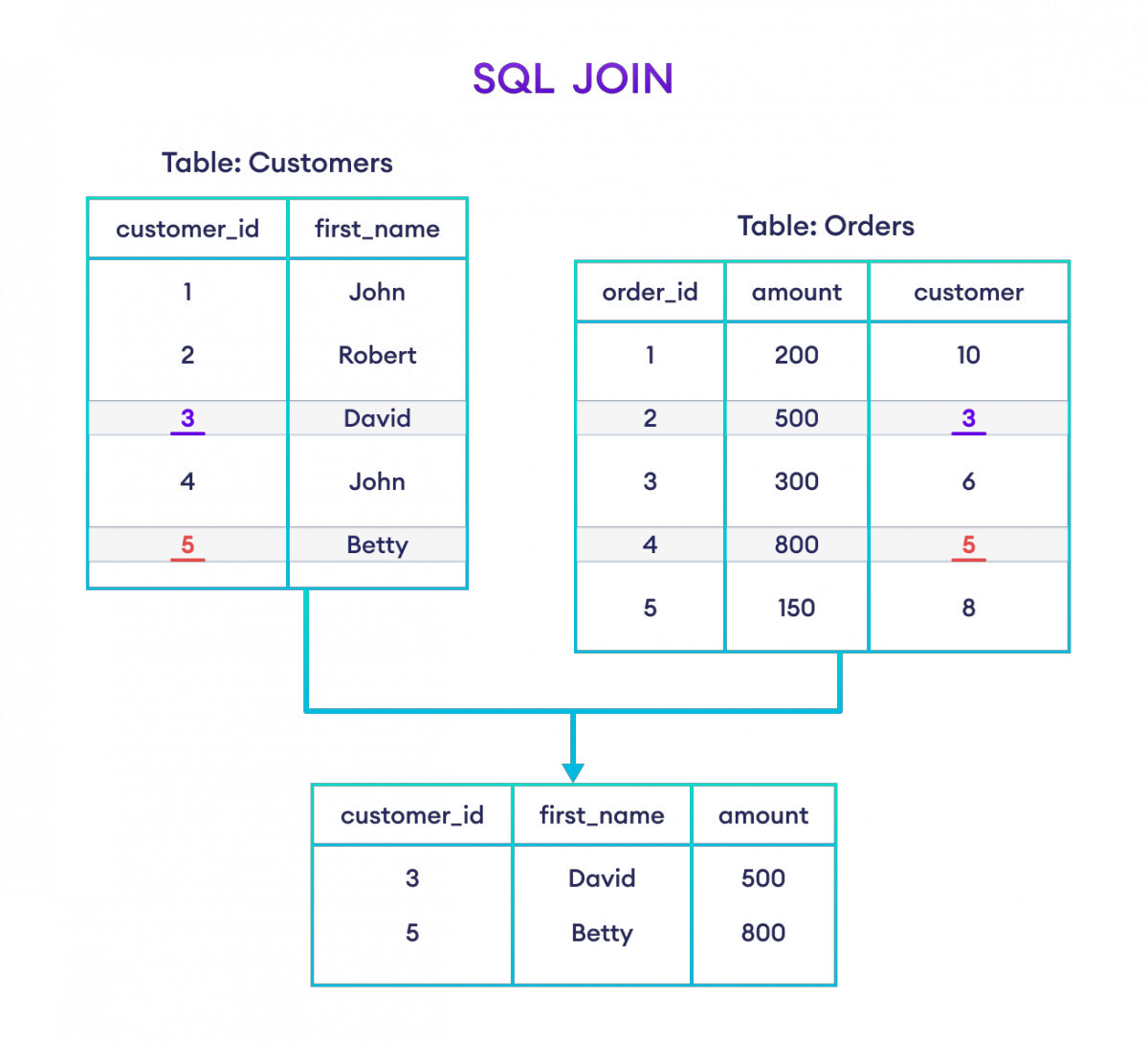
-- customer\_id of Customers and customer column of Orders

SELECT Customers.customer\_id, Customers.first\_name, Orders.amount FROM Customers

JOIN Orders

ON Customers.customer\_id = Orders.customer;

Here's how this code works:



## Example: SQL JOIN

Here, the SQL command selects the customer\_id and first\_name columns (from the Customers table) and the amount column (from the Orders table).

The result set will contain those rows where there is a match between customer\_id (of the Customers table) and customer (of the Orders table).

## Types of SQL JOINs

As we mentioned, the JOIN command we performed in this article is INNER JOIN. In SQL, we have four main types of joins:

* INNER JOIN
* LEFT JOIN
* RIGHT JOIN
* FULL OUTER JOIN

## SQL JOIN and Aliases

We can use AS aliases with table names to make our query short and clean. For example,

-- use alias C for Customers table

-- use alias O for Orders table

SELECT C.customer\_id, C.first\_name, O.amount FROM Customers AS C

JOIN Orders AS O

ON C.customer\_id = O.customer;

Here, the SQL command joins the Customers and Orders tables while assigning the aliases C and O to them, respectively.

Also, we can change the column names temporarily using AS aliases. For example,

-- use alias C for Customers table

-- use alias O for Orders table

SELECT C.customer\_id AS cid, C.first\_name AS name, O.amount FROM Customers AS C

JOIN Orders AS O

ON C.customer\_id = O.customer;

Apart from giving aliases to the tables, the SQL command above also assigns aliases to the columns of the Customers table:

* customer\_id column has the alias cid
* first\_name column has the alias name

## SQL INNER JOIN

The SQL INNER JOIN command joins two tables based on a common column and selects rows that have matching values in these columns.

## Example

-- join Customers and Orders tables with their matching fields customer\_id SELECT Customers.customer\_id, Orders.item

FROM Customers INNER JOIN Orders

ON Customers.customer\_id = Orders.customer;

Here, the SQL query performs an INNER JOIN operation by joining the Customers and Orders tables. It then filters the customer\_id column of the Customers table and the item column of the Orders table into the result set.

## SQL INNER JOIN Syntax

The syntax of the SQL INNER JOIN statement is:

SELECT columns\_from\_both\_tables FROM table1

INNER JOIN table2

ON table1.column1 = table2.column2

Here,

* table1 and table2 are the two tables that are to be joined
* column1 is the column in table1 that is related to column2 in table2 INNER JOIN excludes all the rows that are not common between two tables.

Note: We can also use JOIN instead of INNER JOIN. Basically, these two clauses are the same.

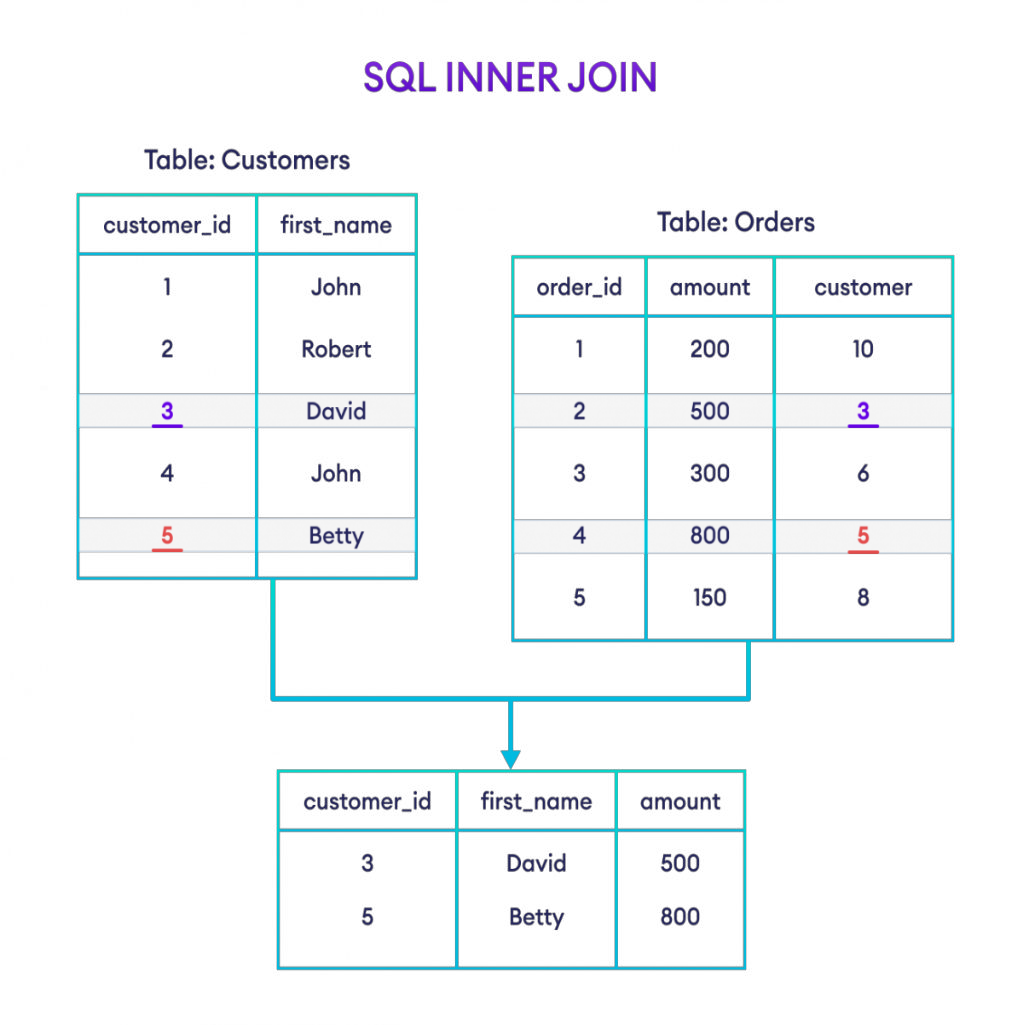
## Example 1: SQL INNER JOIN

-- join the Customers and Orders tables with matching fields customer\_id and customer SELECT Customers.customer\_id, Customers.first\_name, Orders.amount

FROM Customers INNER JOIN Orders

ON Customers.customer\_id = Orders.customer;

Here, the SQL command selects rows from both tables if the values of customer\_id (of the Customers table) and customer (of the Orders table) are a match.



## Example 2: SQL INNER JOIN

Let's look at another example,

-- join Categories and Products tables with their matching fields cat\_id SELECT Categories.cat\_name, Products.prod\_title

FROM Categories INNER JOIN Products

ON Categories.cat\_id = Products.cat\_id;

Here, the SQL command selects common rows between Categories and Products tables with the matching field cat\_id.

The result set has the cat\_name column from Categories and the prod\_title column from Products.

## INNER JOIN With WHERE Clause

Here's an example of INNER JOIN with the WHERE clause:

-- join Customers and Orders table with matching fields customer\_id and customer SELECT Customers.customer\_id, Customers.first\_name, Orders.amount

FROM Customers INNER JOIN Orders

ON Customers.customer\_id = Orders.customer

WHERE Orders.amount >= 500;

Here, the SQL command joins two tables and selects rows where the amount is greater than or equal to 500.

## SQL INNER JOIN With AS Alias

We can use AS aliases inside INNER JOIN to make our query short and clean. For example,

-- use alias C for Categories table

-- use alias P for Products table SELECT C.cat\_name, P.prod\_title FROM Categories AS C

INNER JOIN Products AS P

ON C.cat\_id= P.cat\_id;

Here, the SQL command performs an inner join on the Categories and Products tables while assigning the aliases C and P to them, respectively.

## SQL INNER JOIN With Three Tables

We can also join more than two tables using INNER JOIN. For example,

-- join three tables: Customers, Orders, and Shippings SELECT C.customer\_id, C.first\_name, O.amount, S.status FROM Customers AS C

INNER JOIN Orders AS O

ON C.customer\_id = O.customer INNER JOIN Shippings AS S

ON C.customer\_id = S.customer;

Here, the SQL command

* joins Customers and Orders tables based on customer\_id (from the Customers table) and customer (from the Orders table)
* and joins Customers and Shippingss tables based on customer\_id (from the Customers table) and customer (from the Shippings table)

The command returns those rows where there is a match between column values in both join conditions.

## SQL LEFT JOIN

The SQL LEFT JOIN joins two tables based on a common column. It selects records that have matching values in these columns and the remaining rows from the left table.

## Example

-- left join Customers and Orders tables based on their shared customer\_id columns

-- Customers is the left table

-- Orders is the right table

SELECT Customers.customer\_id, Customers.first\_name, Orders.item FROM Customers

LEFT JOIN Orders

ON Customers.customer\_id = Orders.customer\_id;

Here, the code left joins the Customers and Orders tables based on customer\_id, which is common to both tables. The result set contains:

* customer\_id and first\_name columns from the Customers table (including those whose customer\_id value is not present in the Orders table)
* item column from the Orders table

## SQL LEFT JOIN Syntax

The syntax of the SQL LEFT JOIN statement is:

SELECT columns\_from\_both\_tables FROM table1

LEFT JOIN table2

ON table1.column1 = table2.column2

Here,

* table1 is the left table to be joined
* table2 is the right table to be joined
* column1 and column2 are the related columns in the two tables

## Example: SQL LEFT Join

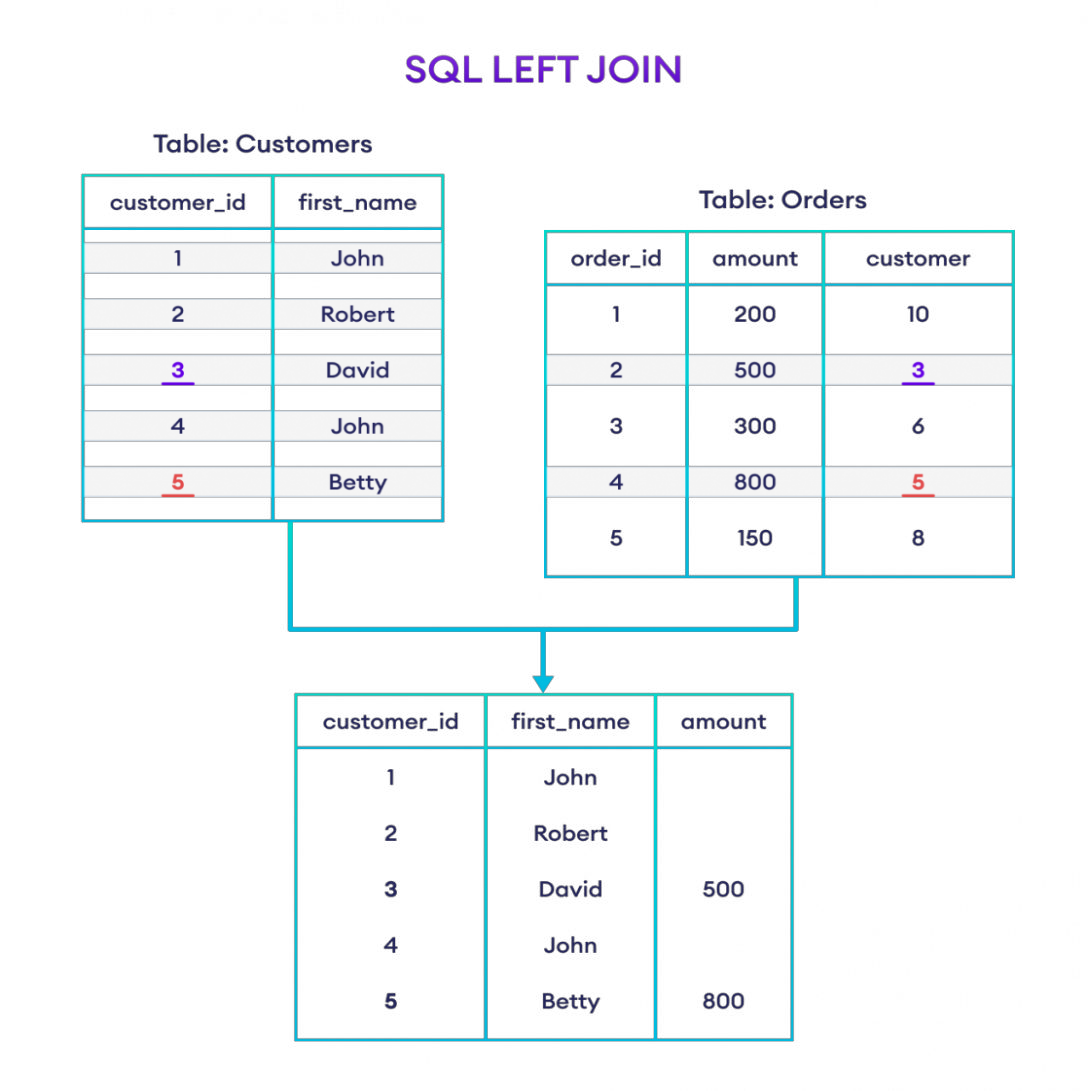
-- left join the Customers and Orders tables

SELECT Customers.customer\_id, Customers.first\_name, Orders.amount FROM Customers

LEFT JOIN Orders

ON Customers.customer\_id = Orders.customer;

Here's how this code works:



## Example: SQL LEFT JOIN

Here, the SQL command selects the customer\_id and first\_name columns (from the Customers table) and the amount column (from the Orders table).

The result set will contain those rows where there is a match between customer\_id (of the Customers table) and customer (of the Orders table), along with all the remaining rows from the Customers table.

## LEFT JOIN With WHERE Clause

The SQL LEFT JOIN statement can have an optional WHERE clause. For example,

SELECT Customers.customer\_id, Customers.first\_name, Orders.amount FROM Customers

LEFT JOIN Orders

ON Customers.customer\_id = Orders.customer WHERE Orders.amount >= 500;

Here, the SQL command joins the Customers and Orders tables and selects rows where the amount is greater than or equal to 500.

## SQL LEFT JOIN With AS Alias

We can use AS aliases inside LEFT JOIN to make our query short and clean. For example,

-- use alias C for Categories table

-- use alias P for Products table SELECT C.cat\_name, P.prod\_title FROM Categories AS C

LEFT JOIN Products AS P

ON C.cat\_id= P.cat\_id;

Here, the SQL command left joins the Categories and Products tables while assigning the aliases C and P to them, respectively.

## SQL RIGHT JOIN

The SQL RIGHT JOIN statement joins two tables based on a common column. It selects records that have matching values in these columns and the remaining rows from the right table.

## Example

-- join Customers and Orders tables

-- based on their shared customer\_id columns

-- Customers is the left table

-- Orders is the right table

SELECT Customers.customer\_id, Customers.first\_name, Orders.item FROM Customers

RIGHT JOIN Orders

ON Customers.customer\_id = Orders.customer\_id;

Here, the code right joins the Customers and Orders tables based on customer\_id, which is common to both tables. The result set contains

* customer\_id and first\_name columns from the Customers table
* item column from the Orders table (including those whose customer\_id value is not present in the Customers table)

## RIGHT JOIN SYNTAX

The syntax of the SQL RIGHT JOIN statement is:

SELECT columns\_from\_both\_tables FROM table1

RIGHT JOIN table2

ON table1.column1 = table2.column2

Here,

* table1 is the left table to be joined
* table2 is the right table to be joined
* column1 and column2 are the related columns in the two tables

## Example: SQL RIGHT JOIN

-- join Customers and Orders tables

-- based on customer\_id of Customers and customer of Orders

-- Customers is the left table

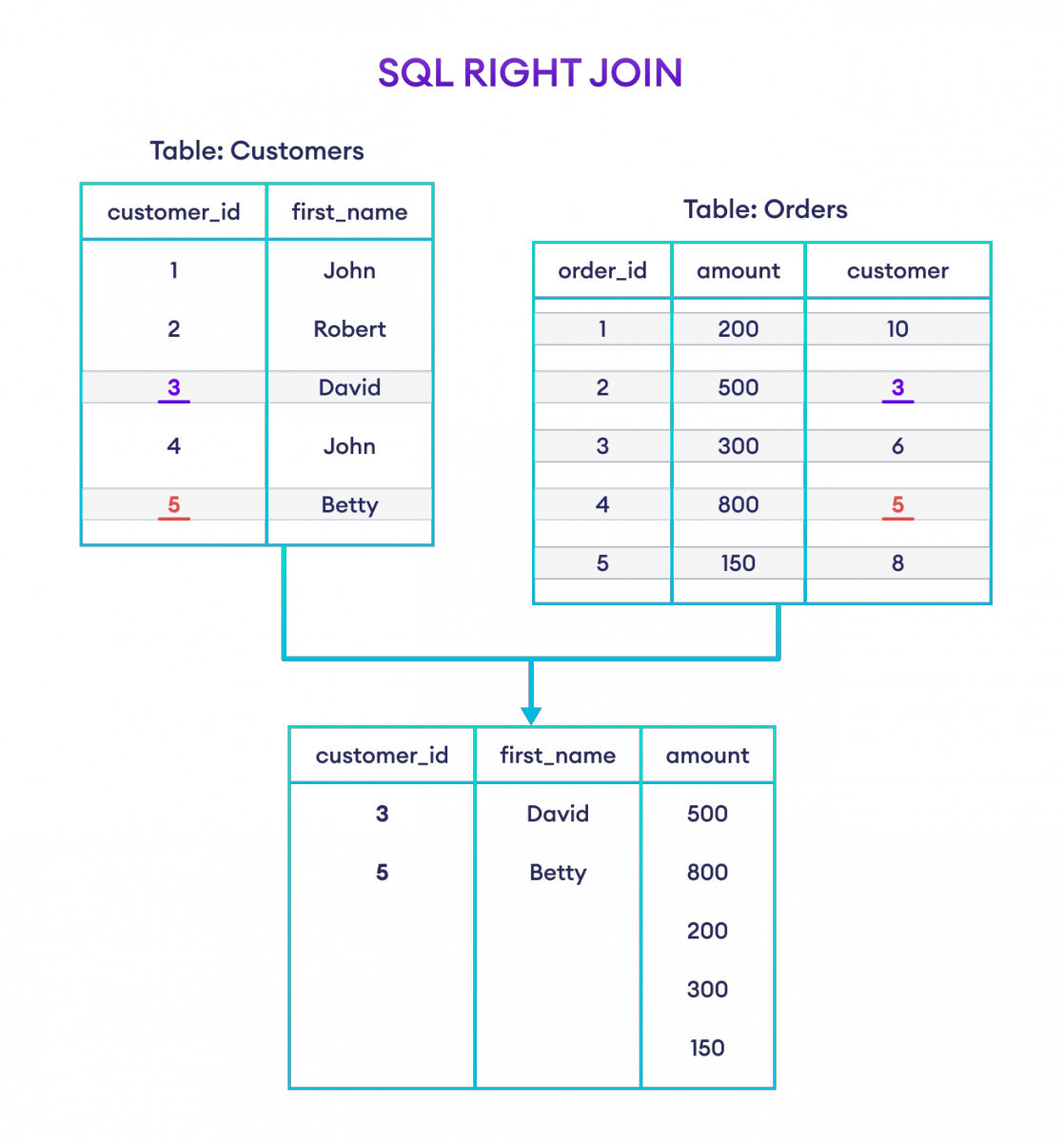
-- Orders is the right table

SELECT Customers.customer\_id, Customers.first\_name, Orders.amount FROM Customers

RIGHT JOIN Orders

ON Customers.customer\_id = Orders.customer;

Here's how this code works:



## Example: SQL RIGHT JOIN

Here, the SQL command selects the customer\_id and first\_name columns (from the Customers table) and the amount column (from the Orders table).

And, the result set will contain those rows where there is a match between customer\_id (of the Customers table) and customer (of the Orders table), along with all the remaining rows from the Orders table.

Note: RIGHT JOIN is not supported by our online SQL compiler since it's based on SQLite. However, you can get the same results by using a LEFT JOIN and swapping the left and right tables.

-- left join Orders and Customers tables

-- Orders is the left table

-- Customers is the right table

SELECT Customers.customer\_id, Customers.first\_name, Orders.amount FROM Orders

LEFT JOIN Customers

ON Orders.customer = Customers.customer\_id;

## RIGHT JOIN With WHERE Clause

The SQL RIGHT JOIN statement can have an optional WHERE clause. For example,

SELECT Customers.customer\_id, Customers.first\_name, Orders.amount FROM Customers

RIGHT JOIN Orders

ON Customers.customer\_id = Orders.customer WHERE Orders.amount >= 500;

Here, the SQL command joins the Customers and Orders tables and selects rows where the amount is greater than or equal to 500.

## SQL RIGHT JOIN With AS Alias

We can use AS aliases inside RIGHT JOIN to make our SQL code short and clean. For example,

-- use alias C for Categories table

-- use alias P for Products table

SELECT C.category\_name, P.product\_title FROM Categories AS C

RIGHT JOIN Products AS P

ON C.cat\_id = P.cat\_id;

Here, the SQL command performs a right join on the Categories and Products tables while assigning the aliases C and P to them, respectively.

## SQL FULL OUTER JOIN

The SQL FULL OUTER JOIN statement joins two tables based on a common column. It selects records that have matching values in these columns and the remaining rows from both of the tables.

## Example

-- full join Customers and Orders tables

-- based on their shared customer\_id columns

-- Customers is the left table

-- Orders is the right table

SELECT Customers.customer\_id, Customers.first\_name, Orders.item FROM Customers

FULL OUTER JOIN Orders

ON Customers.customer\_id = Orders.customer\_id;

Here, the SQL query performs a FULL OUTER JOIN on two tables, Customers and Orders. This means that the result set contains all the rows from both tables, including the ones that don't have common customer\_id values.

## FULL OUTER JOIN SYNTAX

The syntax of the SQL FULL OUTER JOIN statement is:

SELECT columns FROM table1

FULL OUTER JOIN table2

ON table1.column1 = table2.column2;

Here,

* table1 and table2 are the tables to be joined
* column1 and column2 are the related columns in the two tables

## Example: SQL OUTER Join

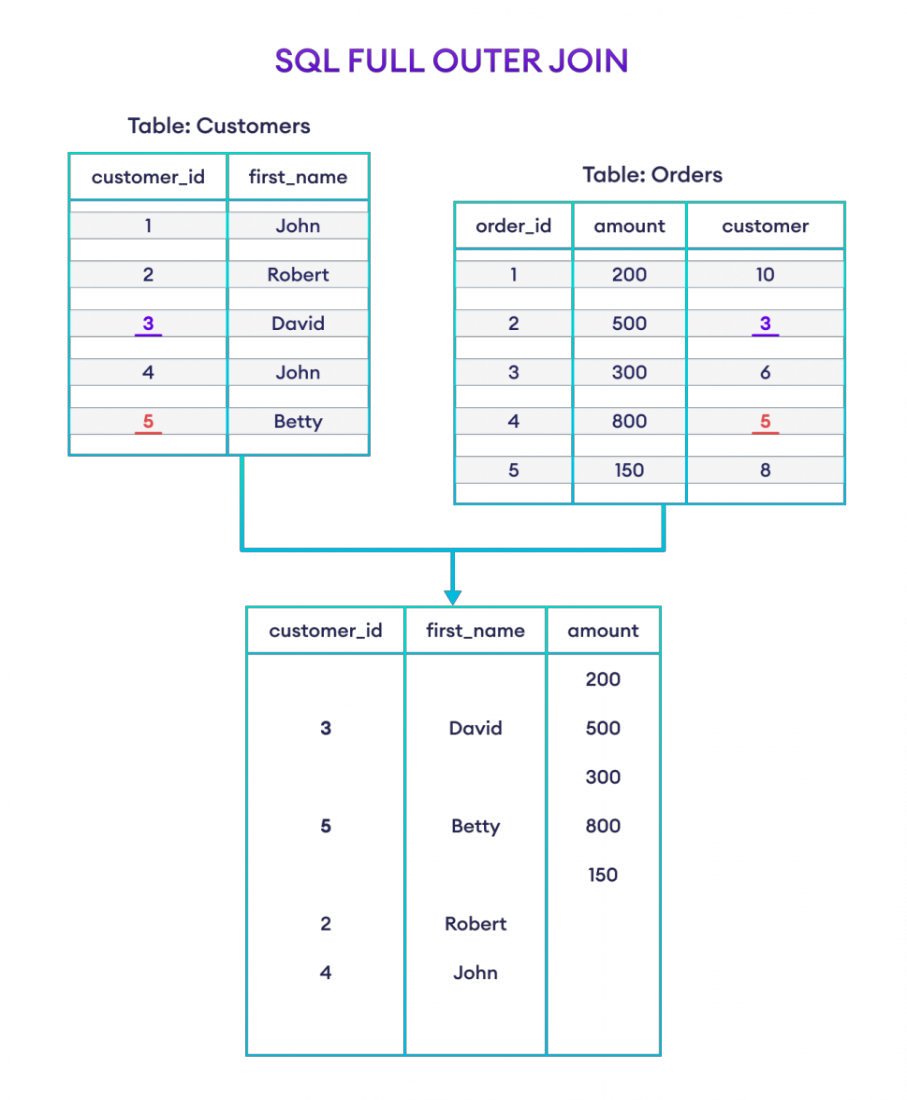
SELECT Customers.customer\_id, Customers.first\_name, Orders.amount FROM Customers

FULL OUTER JOIN Orders

ON Customers.customer\_id = Orders.customer;

Here, the SQL command selects the customer\_id and first\_name columns (from the Customers table) and the amount column (from the Orders table).

The result set will contain all rows of both the tables, regardless of whether there is a match between customer\_id (of the Customers table) and customer (of the Orders table).



## FULL OUTER JOIN With WHERE Clause

The SQL FULL OUTER JOIN statement can have an optional WHERE clause. For example,

SELECT Customers.customer\_id, Customers.first\_name, Orders.amount FROM Customers

FULL OUTER JOIN Orders

ON Customers.customer\_id = Orders.customer WHERE Orders.amount >= 500;

Here, the SQL command joins two tables and selects rows where the amount is greater than or equal to 500.

## SQL FULL OUTER JOIN With AS Alias

We can use AS aliases inside FULL OUTER JOIN to make our query short and clean. For example,

-- use alias C for Categories table

-- use alias P for Products table

SELECT C.category\_name, P.product\_title FROM Categories AS C

FULL OUTER JOIN Products AS P

ON C.category\_id = P.cat\_id;

Here, the SQL command performs a full outer join on the Categories and Products tables while assigning the aliases C and P to them, respectively.

## SQL Aggregate Function

An SQL aggregate function calculates on a set of values and returns a single value. For example, the average function ( AVG) takes a list of values and returns the average.

Because an aggregate function operates on a set of values, it is often used with the GROUP BY clause of the SELECT statement. The GROUP BY clause divides the result set into groups of values and the aggregate function returns a single value for each group.

The following illustrates how the aggregate function is used with the GROUP BY clause:

SELECT c1, aggregate\_function(c2) FROM table

GROUP BY c1;

The following are the commonly used SQL aggregate functions:

* + AVG() – returns the average of a set.
  + COUNT() – returns the number of items in a set.
  + MAX() – returns the maximum value in a set.
  + MIN() – returns the minimum value in a set
  + SUM() – returns the sum of all or distinct values in a set

Except for the COUNT() function, SQL aggregate functions ignore null. You can use aggregate functions as expressions only in the following:

* + The select list of a SELECT statement, either a subquery or an outer query.
  + A HAVING clause

## AVG()

The AVG() function returns the average values in a set. The following illustrates the syntax of the AVG() function:

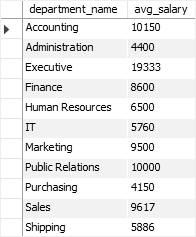
AVG( ALL | DISTINCT)

The ALL keyword instructs the AVG() function to calculate the average of all values while the DISTINCT keyword forces the function to operate on distinct values only. By default, the ALL option is used.

The following example shows how to use the AVG() function to calculate the average salary of each department:

SELECT department\_name, ROUND(AVG(salary), 0) avg\_salary FROM employees INNER JOIN departments USING (department\_id) GROUP BY department\_name ORDER BY department\_name;

Output:



## MIN()

The MIN() function returns the minimum value of a set. The following illustrates the syntax of the MIN() function:

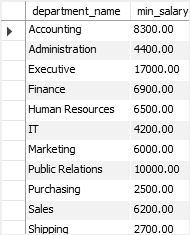
MIN(column | expression)

For example, the following statement returns the minimum salary of the employees in each

department:

SELECT department\_name, MIN(salary) min\_salary FROM employees INNER JOIN departments USING (department\_id) GROUP BY department\_name ORDER BY department\_name;

Output:



## MAX()

The MAX() function returns the maximum value of a set. The MAX() function has the following syntax:

MAX(column | expression)

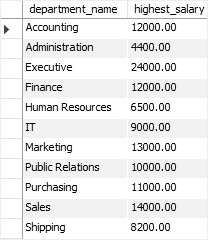
For example, the following statement returns the highest salary of employees in each department:

SELECT department\_name, MAX(salary) highest\_salary FROM employees

INNER JOIN departments USING (department\_id) GROUP BY department\_name

ORDER BY department\_name;

Output:



## COUNT()

The [COUNT()](https://www.sqltutorial.org/sql-aggregate-functions/sql-count/) function returns the number of items in a set. The following shows the syntax of the COUNT() function:

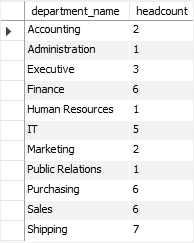
COUNT ( [ALL | DISTINCT] column | expression | \*)

For example, the following example uses the COUNT(\*) function to return the headcount of

each department:

SELECT department\_name, COUNT(\*) headcount FROM employees INNER JOIN departments USING (department\_id) GROUP BY department\_name ORDER BY department\_name;

Output:



## SUM()

The SUM() function returns the sum of all values. The following illustrates the syntax of the SUM() function:

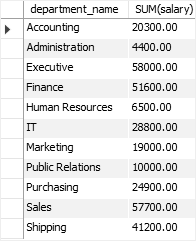
SUM(ALL | DISTINCT column)

For example, the following statement returns the total salary of all employees in each

department:

SELECT department\_id, SUM(salary) FROM employees GROUP BY department\_id;

Output:



You have learned the most commonly used SQL aggregate functions including AVG(), COUNT(), MIN(), MAX(), and SUM() functions.

## Introduction to SQL AVG function

The SQL AVG function is an aggregate function that calculates the average value of a set. The following illustrates the syntax of the SQL AVG function:

AVG([ALL|DISTINCT] expression)

If we use the ALL keyword, the AVG function takes all values in the calculation. By default, the

AVG function uses ALL whether we specify it or not.

If we specify the DISTINCT keyword explicitly, the AVG function will take the unique values only in the calculation.

For example, we have a set of (1,2,3,3,4) and apply the AVG(ALL) to this set, the AVG function will perform the following calculation:

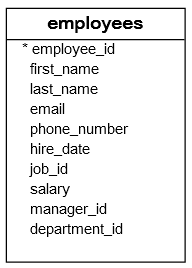
(1+2+3+3+4)/5 = 2.6

However, the AVG(DISTINCT) will process as follows:

(1+2+3+4)/4 = 2.5

SQL AVG function examples

We will use the employees table in the sample database to demonstrate how the SQL AVG function works. The following picture illustrates the structure of the employees table:



To calculate the average salary of all employees, you apply the AVG function to the salary column as follows:

SELECT AVG(salary) FROM employees;

Output:



Let’s apply the DISTINCT operator to see if the result changes:

SELECT AVG(DISTINCT salary) FROM employees;

Output:



It changed because some employees have the same salary.

To round the result to 2 decimal places, you use the ROUND function as follows:

SELECT ROUND(AVG(DISTINCT salary), 2) FROM employees;

Output



To calculate the average value of a subset of values, we add a WHERE clause to the SELECT statement. For instance, to calculate the average salary of employees in the department id 5, we use the following query:

SELECT AVG(DISTINCT salary) FROM employees WHERE department\_id = 5;

Output:



The following statement returns the average salary of employees who hold the job id 6:

SELECT AVG(salary) FROM employees WHERE job\_id = 6;

Output:

