Hyperiondev

TASK

SQL

Visit our website

Introduction

WELCOME TO THE SQL TASK!

In this task, you will learn a language used for manipulating and managing databases: Structured Query Language, commonly referred to as SQL. In addition, you will be introduced to a popular open-source relational database: MariaDB.

INTRODUCTION TO SQL

SQL is a database language that is composed of commands that enable users to create databases or table structures, perform various types of data manipulation and data administration, as well as query the database to extract useful information. SQL is supported by all relational database management system (RDBMS) software. SQL is portable, which means that a user does not have to relearn the basics when moving from one RDBMS to another because all RDBMSs will use SQL in almost the same way.

SQL is easy to learn as its vocabulary is relatively simple. Its basic command set has a vocabulary of fewer than 100 words. It is also a declarative language, which means that the user specifies what must be done and not how it should be done. Users do not need to know the physical data storage format or the complex activities that take place when an SQL command is executed in order to issue a command.

SQL functions fit into **two** general categories:

1. **Data definition language (DDL)** includes commands that enable users to define, edit, and delete data structures and schemas stored within a database management system. Using commands like CREATE, ALTER, and DROP, DDL can create new databases and tables, modify column definitions or table properties, add/remove constraints, and delete entire data structures when no longer needed. It provides fine-grained control over the metadata systems that organise and categorise data.

2. **Data manipulation language (DML)** includes commands that enable users to access, modify, and manipulate data stored in a database. Key DML operations map to the **CRUD** acronym – **create** new entries, **read/retrieve** existing records, **update** or edit stored values, and **delete** data. With commands like SELECT, INSERT, UPDATE, and DELETE, DML provides powerful tools for curating, shaping, cleansing, and managing large

datasets, such as query filtering on specific criteria. Manipulating data dynamically is vital for impactful analytics.

The commands in each category are given in the tables below. We will explore some of these commands in the rest of this task.

The table below lists the SQL **data definition** (DDL) commands:

Command	Description
`CREATE SCHEMA AUTHORIZATION`	Creates a database schema
`CREATE TABLE`	Creates a new table in the user's database schema
`NOT NULL`	Ensures that a column will not have null values
`UNIQUE`	Ensures that a column will not have duplicate values
`PRIMARY KEY`	Defines a primary key for a table
`FOREIGN KEY`	Defines a foreign key for a table
`DEFAULT`	Defines a default value for a column when no value is given
`CHECK`	Used to validate data in an attribute
`CREATE INDEX`	Creates an index for the table
`CREATE VIEW`	Creates a dynamic subset of rows or columns from one or more tables
`ALTER TABLE`	Modifies a table (adds, modifies, or deletes attributes or constraints)
`CREATE TABLE AS`	Creates a new table based on a query in the user's database schema
`DROP TABLE`	Permanently deletes a table
`DROP INDEX`	Permanently deletes an index
`DROP VIEW`	Permanently deletes a view

(Adapted from: Rob & Coronel, 2009, p. 225)

The table below lists the SQL **data manipulation** (DML) commands:

Command	Description
`INSERT`	Inserts rows into a table
`SELECT`	Select attributes from rows in one or more tables or views
`WHERE`	Restricts the selection of rows based on a conditional expression
`GROUP BY`	Groups the selected rows based on one or more attributes
`HAVING`	Restricts the selection of grouped rows based on a condition
`ORDER BY`	Orders the selected rows based on one or more attributes
`UPDATE`	Modifies an attribute's values in one or more tables' rows
`DELETE`	Deletes one or more rows from a table
`COMMIT`	Permanently saves data changes
`ROLLBACK`	Restores data to its original values
Comparison Operators	`=, <, >, <=, >=, <>`
Logical Operators	`AND, OR, NOT`
Special Operators	Used in conditional expressions
`BETWEEN`	Checks whether an attribute value is within a range
`IS NULL`	Checks whether an attribute value is null
`LIKE`	Checks whether an attribute value matches a given string pattern
`IN`	Checks whether an attribute value matches any value within a value list
`EXISTS`	Checks whether a subquery returns any rows
`DISTINCT`	Limits values to unique values
Aggregate Functions	Used with SELECT to return mathematical summaries on columns
`COUNT`	Returns the number of rows with non-null values for a given column
`MIN`	Returns the minimum attribute value found in a given column
`MAX`	Returns the maximum attribute value found in a given column
`SUM`	Returns the sum of all values for a given column
`AVG`	Returns the average of all values for a given column

(Adapted from: Rob & Coronel, 2009, p. 225)

CREATING TABLES

To create new tables in SQL, you use the `CREATE TABLE` statement. Pass all the columns you want in the table, as well as their data types, as arguments to the `CREATE TABLE` function. The table will be organised by columns and rows, like a spreadsheet. Each column is called a field, and has a field name which functions like the column heading. Each field holds data on a specific topic, where the field name usually indicates the topic. In the example table below, the fields are `StudentNumber`, `Name`, `Surname`, `CellNumber`, and `Address`. Each row, on the other hand, is called a record, and each record holds a full set of data for a particular entity/thing/person/etc. In the example table below, the entities we're storing data about are students, and each row holds a record of a single student's data.

StudentNumber	Name	Surname	CellNumber	Address
4f8149817	Liano	Charook	082 283 9009	3 Maple Str
5e9285991	Bianca	Manan	072 329 5571	17 Willow Ave
9b7744992	Cameron	Devilliers	072 410 9077	9 Birch Lane

The syntax of the `CREATE TABLE` statement is shown below:

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

Note the optional constraint arguments. They are used to specify rules for data in a table. Constraints that are commonly used in SQL include:

- `NOT NULL`**:** Ensures that a column does not have a null value.
- `UNIQUE`**:** Ensures that all values in a column are different.
- `DEFAULT`**:** Sets a default value for a column when no value is specified.
- `INDEX`**:** Creates an index linked to a specific column or set of columns that is used to create and retrieve data from the database very quickly. This works the same way as an index in a book, where keywords in the index enable us to quickly find those topics in the book, except that in a database the 'keyword' is a field name and indexing it helps the computer find and access the data in that field more quickly. For example, If you frequently run queries

to retrieve employees based on their `employee_id`, you should create an index on this column for faster lookup.

Let's look at another example. To create a table called `Employee` that contains five columns (`EmployeeID`, `LastName`, `FirstName`, `Address`, and `PhoneNumber`), you would use the following SQL:

```
CREATE TABLE Employee (
    EmployeeID int,
    LastName varchar(255),
    FirstName varchar(255),
    Address varchar(255),
    PhoneNumber varchar(255)
);
```

The `EmployeeID` column is of type `int` and will, therefore, hold an integer value. The `LastName`, `FirstName`, `Address`, and `PhoneNumber` columns are of type `varchar` and will, therefore, hold characters. The number in brackets indicates the maximum number of characters, which in this case is 255.

The `CREATE TABLE` statement above will create an empty `Employee` table that will look like this:

EmployeeID	LastName	FirstName	Address	PhoneNumber

When creating tables, it's advisable to add a **primary key** to one of the columns as this will help keep entries unique and will speed up select queries. A primary key is a field which holds data that will definitely be different for each record, and can be used as an identifier for records.

Examples of primary keys in everyday life include things like a national ID number, social security number, insurance policy number, or employee or student number. Primary keys must contain unique values and cannot contain null values, or they would not be able to function as the unique identifiers for records. A table can only contain one primary key; however, the primary key may consist of a single column or a combination of multiple columns.

You can add a primary key when creating the `Employee` table as follows:

```
CREATE TABLE Employee (
    EmployeeID int NOT NULL,
    LastName varchar(255) NOT NULL,
    FirstName varchar(255),
    Address varchar(255),
    PhoneNumber varchar(255),
    PRIMARY KEY (EmployeeID)
);
```

To name a primary key constraint and define a primary key constraint on multiple columns, you use the following SQL syntax:

```
CREATE TABLE Employee (
    EmployeeID int NOT NULL,
    LastName varchar(255) NOT NULL,
    FirstName varchar(255),
    Address varchar(255),
    PhoneNumber varchar(255),
    CONSTRAINT PK_Employee PRIMARY KEY (EmployeeID, LastName)
);
```

In the example above, there is only one primary key named `PK_Employee`. However, the value of the primary key is made up of two columns: `EmployeeID` and `LastName`.

INSERTING ROWS

The table that we have just created is empty and needs to be populated with rows or records. We can add entries to a table using the `INSERT INTO` command.

There are two ways to write the `INSERT INTO` command: inserting values for specific columns and inserting values for all columns. It's generally a good practice to explicitly specify the columns for which you are providing values during an INSERT statement. Explicit column specification makes the code more readable and reduces the risk of errors, especially when the table schema changes or new columns are added.

1. **Inserting values for specific columns:**

   If you only want to insert data into specific columns and want to specify both the column names and the corresponding values, you can use this method. The syntax is as follows:

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

   For instance, to add an entry to the `Employee` table using this approach, you would write:

```
INSERT INTO Employee (EmployeeID, LastName, FirstName, Address,
PhoneNumber)
       VALUES (1234, 'Smith', 'John', '25 Oak Rd', '0837856767');
```

   This method is useful when you want to insert data into specific columns and leave other columns with **default values** or **NULL**. For instance, as the FirstName and Address fields don't have a **NOT NULL** constraint, they don't necessarily have to be included when inserting. The FirstName and Address fields for this record will be **NULL**:

```
INSERT INTO Employee (EmployeeID, LastName, PhoneNumber)
      VALUES (1234, 'Smith', '0837856767');
```

   To insert multiple rows using this method, you can provide multiple sets of values within parentheses, separated by commas. Each set of values represents a row to be inserted. Here's an example:

```
INSERT INTO Employee (EmployeeID, LastName, FirstName, Address,
PhoneNumber)
      VALUES (1234, 'Smith', 'John', '25 Oak Rd', '0837856767'),
             (5678, 'Brown', 'Robert', '5 Pine Str', '0821116789'),
             (9112, 'Davies', 'Emily', '25 Maple Ave', '0876543210');
```

2. **Inserting values for all columns:**

   If you want to add values for all the columns in the table and the order of the values matches the exact order of the columns, you can simply not specify the column names. The syntax for this approach is as follows:

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

For example, to add an entry to the **Employee** table, you would do the following:

```
INSERT INTO Employee
      VALUES (1234, 'Smith', 'John', '25 Oak Rd', '0837856767');
```

To insert multiple rows using this method, you can simply provide multiple sets of values within parentheses, separated by commas. Each set of values represents a row to be inserted. For instance:

```
INSERT INTO Employee
      VALUES (1234, 'Smith', 'John', '25 Oak Rd', '0837856767'),
             (5678, 'Brown', 'Robert', '5 Pine Str', '0821116789'),
             (9112, 'Davies', 'Emily', '25 Sycamore Ave',
            '0876543210');
```

RETRIEVING DATA FROM A TABLE

The **SELECT** statement is used to fetch data from a database. The data returned is stored in a result table, known as the result set. The syntax of a **SELECT** statement is as follows:

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

Here, **column1, column2, ...** refer to the column names of the table from which you want to select data. The following example below selects the FirstName and LastName columns from the **Employee** table:

```
SELECT FirstName, LastName
FROM Employee;
```

If you want to select all the columns in the table, you use the following syntax:

```
SELECT * FROM Employee;
```

The asterisk (*) indicates that we want to fetch all of the columns without excluding any of them. The above **SELECT** command will show all columns and rows from the **Employee** table.

You can also order and filter the data that is returned when using the `SELECT` statement using the `ORDER BY` and `WHERE` commands.

ORDER BY

You can use the `ORDER BY` command to sort the results returned in ascending or descending order based on one or more columns. The `ORDER BY` command sorts the records in ascending order by default. You need to use the `DESC` keyword to sort the records in descending order.

The `ORDER BY` syntax is as follows:

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

The example **below** selects all Employees in the `Employee` table and sorts them in descending order (4, 3, 2, 1 for numbers, and Z to A for letters), based on the values in the FirstName column:

```
SELECT * FROM Employee
ORDER BY FirstName, LastName DESC;
```

WHERE

The `WHERE` clause allows us to filter data depending on a specific condition. The syntax of the `WHERE` clause is as follows:

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

The following SQL statement selects all the employees with the first name 'John' in the `Employee` table:

```
SELECT * FROM Employee
WHERE FirstName='John';
```

Note that SQL requires single quotes around text values; however, you do not need to enclose numeric fields in quotes. You should also note that, for conditions in

SQL, we use a single **'='**, which is equivalent to the **'=='** used in Python, JavaScript, and Java.

You can use logical operators **(AND, OR)** and comparison operators **(=, <, >, <=, >=, <>)** to make **WHERE** conditions as specific as you like.

For example, suppose you have the following table that contains the most sold albums of all time:

Albums table:

Artist	Album	Released	Genre	sales_in_millions
Michael Jackson	Thriller	1982	pop	70
AC/DC	Back in Black	1980	rock	50
Pink Floyd	The Dark Side of the Moon	1973	rock	45
Whitney Houston	The Bodyguard	1992	soul	44

You can select the records that are classified as rock and have sold under 50 million copies by simply using the **AND** operator as follows:

```
SELECT *
FROM albums
WHERE Genre = 'rock' AND sales_in_millions <= 50
ORDER BY Released
```

WHERE statements also support some special operators to customise queries further:

- **IN:** Compares the column to multiple possible values and returns true if it matches at least one.
- **BETWEEN:** Checks if a value is within an inclusive range.
- **LIKE:** Searches for a pattern match, which is specific character patterns within text data. For example, using the pattern specification 'S%' searches for all text that starts with an 'S' followed by any number of other characters (no matter which characters). The percentage sign is referred to as a wildcard, which represents any characters. An underscore represents any single character. For example, 'J__n' can be used to search any text that starts with a 'J' followed by two characters and ends with an 'n'.

For example, if we want to select the pop and soul albums from the table above, we can use:

```
SELECT * FROM albums
WHERE Genre IN ('pop', 'soul');
```

Or, if we want to get all the albums released between 1975 and 1985, we can use:

```
SELECT * FROM albums
WHERE Released BETWEEN 1975 AND 1985;
```

We can also find all albums sung by artists whose names start with an 'M':

```
SELECT * FROM albums
WHERE Artist LIKE 'M%';
```

AGGREGATE FUNCTIONS

SQL has many functions that do all sorts of helpful things. Some of the most regularly used ones are:

- **`COUNT()`:** Returns the number of rows.
- **`SUM()`:** Returns the total sum of a numeric column.
- **`AVG()`:** Returns the average of a set of values.
- **`MIN()` / `MAX()`:** Gets the minimum or maximum value from a column.
- **`GROUP BY`:** Group rows returned by a query into summary rows based on the values in one or more columns.

For example, to get the most recent year in the **`Album`** table, we can use:

```
SELECT MAX(Released)
FROM albums;
```

Or to get the number of albums released between 1975 and 1985 we can use:

```
SELECT COUNT(album)
WHERE Released BETWEEN 1975 AND 1985;
```

The **`GROUP BY`** clause is often used in conjunction with the other functions. The following SQL command calculates the total sales by genre:

```sql
SELECT Genre, SUM(sales_in_millions) AS total_sales
FROM albums
GROUP BY Genre;
```

RETRIEVING DATA ACROSS MULTIPLE TABLES

In complex databases, there are often several tables connected to each other in some way. A **`JOIN`** clause is used to combine rows from two or more tables based on a related column between them. Look at the two tables below:

`VideoGame` table:

ID	Name	DeveloperID	Genre
1	Super Mario Bros.	2	platformer
2	World of Warcraft	1	MMORPG
3	The Legend of Zelda	2	adventure

`GameDeveloper` table:

ID	Name	Country
1	Blizzard	USA
2	Nintendo	Japan

The **`VideoGame`** table contains information about various video games, while the **`GameDeveloper`** table contains information about the developers of the games. The **`VideoGame`** table has a **`DeveloperID`** column that holds the game developer's ID, which represents the ID of the respective developer from the **`GameDeveloper`** table. **`DeveloperID`** points to a specific developer with a matching ID (primary key) in the **`GameDeveloper`** table; a column that can be used to link two tables like this is called a foreign key. A foreign key in one table **must always correspond** to a primary key in another table.

From the tables above we can see that a developer called 'Blizzard' from the USA developed the game titled 'World of Warcraft'. The foreign key logically links the two tables and allows us to access and use the information stored in both of them at the same time.

If we want to create a query that returns everything we need to know about the games, we can use **INNER JOIN** to acquire the columns from both tables:

```
SELECT VideoGame.Name, VideoGame.Genre, GameDeveloper.Name,
GameDeveloper.Country
FROM VideoGame
INNER JOIN GameDeveloper
ON VideoGame.DeveloperID = GameDeveloper.ID;
```

Notice that in the **SELECT** statement above, we specify the name of the table and the column from which we want to retrieve information, and not just the name of the column as we have done previously. This is because we are getting information from more than just one table, and it is possible that tables may have columns with the same names. In this case, both the table **VideoGame** and the table **GameDeveloper** contain columns called Name. In the next section, you will see how to use aliases to further address this issue.

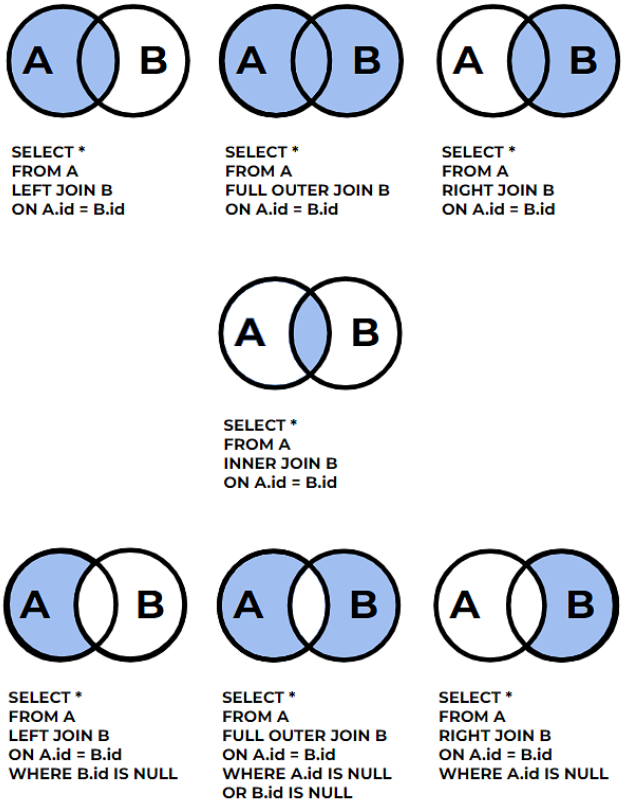
Also, notice that we use the **ON** clause to specify how we link the foreign key in one table to the corresponding primary key in the other table. The query above would result in the following dataset being returned:

VideoGame.Name	VideoGame.Genre	GameDeveloper.Name	GameDeveloper.Country
Super Mario Bros.	platformer	Nintendo	Japan
World of Warcraft	MMORPG	Blizzard	USA
The Legend of Zelda	adventure	Nintendo	Japan

The **INNER JOIN** is the simplest and most common type of **JOIN**. However, there are many other different types of join in SQL. Let's take a look at these and what they do.

- **INNER JOIN:** Returns records that have matching values in both tables.
- **LEFT JOIN:** Returns all records from the left table and the matched records from the right table.
- **RIGHT JOIN:** Returns all records from the right table and the matched records from the left table.
- **FULL JOIN:** Returns all records when there is a match in either the left or right table.

The figure below provides a graphical representation of the above joins plus some extra ones that are less common. Look carefully at the Venn diagrams and ensure you understand exactly which records would be returned from tables A and B if the given SQL code was executed.



ALIASES

Notice that in the **`VideoGame`** and **`GameDeveloper`** tables, there are two columns called **`Name`**. This can become confusing. Aliases are used to give a table or column a temporary name. An alias only exists for the duration of the query and is often used to make column names more readable.

The alias column syntax is:

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

The alias table syntax is:

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

The following SQL statement creates an alias for the **Name** column from the **GameDeveloper** table:

```
SELECT Name AS Developer
FROM GameDeveloper;
```

See how aliases have been used below:

```
SELECT games.Name, games.Genre, devs.Name AS Developer, devs.Country
FROM VideoGame AS games
INNER JOIN GameDeveloper AS devs
ON games.DeveloperID = devs.ID;
```

As you can see, this starts to get quite difficult to follow and read as the query statements grow. In SQL, the **WITH** clause, also known as a Common Table Expression (CTE), allows you to define a temporary result set that you can reference within the context of a larger SQL query. It enhances the readability and maintainability of complex queries by breaking them down into smaller, named, and reusable components. The **WITH** clause example is focused solely on retrieving developer names from the **GameDeveloper** table:

```
WITH GameInfo AS (
      SELECT games.ID, games.Name AS GameName, games.Genre, devs.Name AS
Developer
    FROM VideoGame AS games
    INNER JOIN GameDeveloper AS devs ON games.DeveloperID = devs.ID
)
```

The GameInfo CTE includes information about the video games and their developers by joining the **VideoGame** and **GameDeveloper** tables. The main query then selects distinct developer names from the CTE:

```
SELECT DISTINCT Developer
FROM GameInfo;
```

This approach allows you to create a CTE that encompasses the logic of joining the tables and then to reuse the CTE in subsequent queries.

UPDATING DATA

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

To use the **UPDATE** statement you:

- Choose the table where the row you want to change is located.
- Set the new value(s) for the desired column(s).
- Select which of the rows you want to update using the **WHERE** statement. If you omit this, all rows in the table will change.

The syntax for the update statement is:

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

Take a look at the following **Customer** table:

CustomerID	CustomerName	Address	City
1	Maria Anderson	23 York Str	New York
2	Jackson Peters	124 River Rd	Berlin
3	Thomas Hardy	455 Hanover Sq	London
4	Kelly Martins	55 Loop Str	Cape Town

The following SQL statement updates the first customer (CustomerID = 1) with a new address and a new city:

```
UPDATE Customer
SET Address = '78 Oak Str', City= 'Los Angeles'
WHERE CustomerID = 1;
```

DELETING ROWS

Deleting a row is a simple process. All you need to do is select the right table and row that you want to remove. The **DELETE** statement is used to delete existing rows in a table.

The **DELETE** statement syntax is as follows:

```
DELETE FROM table_name
WHERE condition;
```

The following statement deletes the customer 'Jackson Peters' from the `Customer` table:

```
DELETE FROM Customer
WHERE CustomerName='Jackson Peters';
```

You can also delete all the data inside a table, i.e., all rows, without deleting the table:

```
DELETE FROM table_name;
```

When dealing with foreign keys, it's important to understand the concept of referential integrity. Referential integrity means ensuring that relationships between tables remain valid, meaning that a foreign key in one table must correspond to a primary key in another table. When you want to delete records from a table that is referenced by foreign keys in other tables, you need to be careful to maintain referential integrity.

Therefore, deletion is a last resort because it can lead to data inconsistency and integrity issues if not handled carefully. Always ensure that deleting records won't violate referential integrity and consider alternative approaches like those mentioned above to maintain a well-structured and consistent database

DELETING TABLES

The **`DROP TABLE`** statement is used to remove every trace of a table from a database. The syntax is as follows:

```
DROP TABLE table_name;
```

For example, if we want to delete the table **`Customer`**, we do the following:

```
DROP TABLE Customer;
```

If you want to delete the data inside a table but not the table itself, you can use the **`TRUNCATE TABLE`** statement:

```
TRUNCATE TABLE table_name;
```

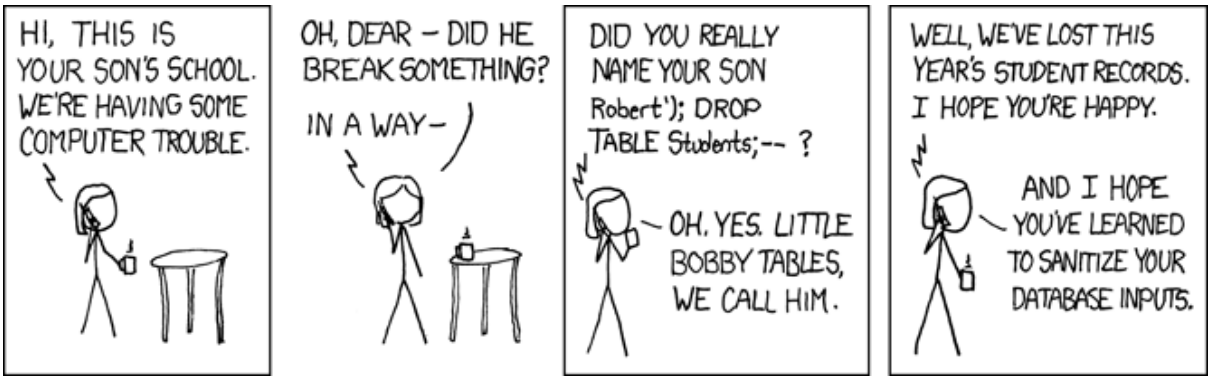
While the `TRUNCATE TABLE` and `DELETE FROM` statements perform similarly in terms of removing data from a table, there are a few important differences to consider when deciding which statement to use:

`TRUNCATE TABLE`	**`DELETE FROM table_name`**
A data definition language (DDL) command that's automatically committed.	A data manipulation language (DML) command that is not automatically committed.
Data cannot be recovered as it is a non-transactional operation (it cannot be rolled back once executed) and does not generate any undo logs.	Data can be recovered as it supports transactional operations and logs row-level details.
Removes all data rows and resets certain attributes associated with the table's structure, such as the value of the identity columns, to their initial values.	Removes all data rows while retaining attributes associated with the table's structure.
Faster speed and performance as it removes all rows as a single operation.	Slower speed and performance as it removes each row one by one.
Use case: If you have a transactional database where you want to do a bulk purge of all transactional data accumulated over time and reclaim storage space.	Use case: If you want to delete specific data, based on certain conditions, with the added flexibility to roll back if needed.

Take note:

The behaviour of the `TRUNCATE TABLE` statement can vary across different database providers. While some database systems support rollback, others may not, e.g., in the Oracle DB. Therefore, it's important to consult the documentation specific to your database to understand whether rollback is supported.

Here's a little SQL humour related to deleting tables. Sanitising database inputs consists of removing any unsafe characters from user inputs. See if you can understand what happens in the cartoon below and why. What does **--** mean in SQL (see panel 3), and why is this relevant? See if you can find out!



*(Source: **XKCD**)*

MariaDB

MariaDB is an open-source RDBMS written in C and C++ that is a fork of MySQL. Think of MariaDB as a smart organiser for digital information. It's a free and open tool that's great at handling lots of data quickly and reliably. People can contribute to making it better, ensuring it stays up-to-date and efficient. Many websites and apps use MariaDB because it's built to handle large amounts of information with speed and stability.

Learning MariaDB is like gaining a key skill in managing digital information. It's particularly useful for web development and software applications where efficiently handling and storing data is essential. Understanding MariaDB provides a solid foundation for working with databases, which are crucial in today's tech-driven world.

Extra resource

If you'd like to know more about SQL, we highly recommend reading the book **Database Design (2nd ed.)** by Adrienne Watt. Chapters 15, 16, and Appendix C of this book provide more detail regarding what has been covered in this task.

Compulsory Task 1

Answer the following questions:

- Go to the **JDoodle Online SQL IDE**. This is where you can write and test your SQL code using their databases. Once you are happy with your code, paste it into a text file and save the file in your task folder as **Student.txt**.
- Write the SQL code to create a table called `Student`. The table structure is summarised in the table below.

  Note that `STU_NUM` must be set up as the **primary key**.

Attribute Name	Data Type
STU_NUM	CHAR(6)
STU_SNAME	VARCHAR(15)
STU_FNAME	VARCHAR(15)
STU_INITIAL	CHAR(1)
STU_STARTDATE	DATE
COURSE_CODE	CHAR(3)
PROJ_NUM	INT(2)

- After you have created the table, write the SQL code to enter the following rows of data into the table as below:

STU_NUM	STU_SNAME	STU_FNAME	STU_INITIAL	STU_STARTDATE	COURSE_CODE	PROJ_NUM
01	Snow	Jon	E	2014-04-05	201	6
02	Stark	Arya	C	2017-07-12	305	11
03	Lannister	Jamie	C	2012-09-05	101	2
04	Lannister	Cercei	J	2012-09-05	101	2

05	Greyjoy	Theon	I	2015-12-09	402	14
06	Tyrell	Margaery	Y	2017-07-12	305	10
07	Baratheon	Tommen	R	2019-06-13	201	5

- Write the SQL code that will return all records which have a `COURSE_CODE` of 305.

- Write the SQL code to change the course code to 304 for the person whose student number is 07.

- Write the SQL code to delete the row of the person named Jamie Lannister, who started on 5 September 2012, whose course code is 101 and project number is 2. Use logical operators to include all of the information given in this problem.

- Write the SQL code that will change the `PROJ_NUM` to 14 for all those students who started before 1 January 2016 and whose course code is at least 201.

- Write the SQL code that will delete the `Student` table entirely. Hint: Use `DROP TABLE`.

Rate us

Share your thoughts

HyperionDev strives to provide internationally excellent course content that helps you achieve your learning outcomes.

Think that the content of this task, or this course as a whole, can be improved? Do you think we've done a good job?

Click here to share your thoughts anonymously.

REFERENCES

Patel, J. (2013). *PHP and MySQL: Practice it learn it*. eBookIt.com.

Rob, P., & Coronel, C. (2009). *Database systems: Design, implementation, and management* (8th ed.). Thomson Course Technology.

W3Schools. (n.d.). *SQL joins*. Retrieved April 15, 2019, from **https://www.w3schools.com/sql/sql_join.asp**