Hyperiondev

LESSON

Introduction to SQL

Visit our website

Introduction

WELCOME TO THE INTRODUCTION TO SQL LESSON!

In this lesson, you will be introduced to the database language SQL, and the concepts of Data Definition vs Data Manipulation language (DDL vs DML) and associated commands. We'll show you how to create and manipulate tables and indexes, query data and make queries more specific with criteria, and create joins. The lesson concludes with a summary of the functionality available to support large objects, and how to create triggers, stored procedures, and functions. At the end there is a task for you to put into practice what you have learned.

INTENDED LEARNING OUTCOMES

You will know that you have succeeded at this task if, by the time you complete it, you are able to:

- Write program code for database access for a computer application using SQL
- Test programs for a computer application that accesses a database using SQL

Refer to the self-assessment table provided after the programming task near the end of the document for a checklist to assess your own degree of achievement of each of these learning outcomes. Try to gauge your own learning and address any weak areas before submitting your task, which will become part of your formative assessment portfolio.

This lesson addresses Unit Standard 114048: Create database access for a computer application using structured query language.

Specific Outcomes:

1. **SO1: Review the requirements for database access for a computer application using SQL (Range: Piecemeal development, Data sharing, Integration, Abstraction, Data Independence, Data models, Data Definition Language, Data Manipulation Language, Data Control, Language (at least 4))**
2. **SO2: Design database access for a computer application using SQL. (Range: Simple indexes, Multi-level indexes, Index-sequential file, Tree structures, SQL CREATE INDEX and DROP INDEX statements (at least 3))**
3. **SO3: Write program code for database access for a computer application using SQL. (Range: Row types, User-defined types, User**

defined routines, Reference types, Collection types, Support for large objects, Stored procedures, Triggers (at least 3))

4. **SO4: Test programs for a computer application that accesses a database using SQL. (Range: Logic tests; Access functions; Debugging.)**
5. **SO5: Document programs for a computer application that accesses a database using SQL. (Range: Clarity; Simplicity; Ease of Use)**

CONCEPT: STRUCTURED QUERY LANGUAGE (SQL)

SQL (Structured Query Language) was introduced in Unit 3. Here we will look at this query language in more detail.

According to Encyclopaedia Britannica (2020), SQL is a "language designed for eliciting information from databases". Coronel et al. (2011) expand on this definition, adding that SQL's commands allow a user to create tables of data, from which the data can be manipulated and sorted. We can, therefore, simplify by saying that SQL is a language that we use to turn data into information.

Like many coding languages, SQL's vocabulary is quite similar to English, which makes it a fairly easy language to learn.

SQL statements fit into two general categories you saw defined in a previous lesson (Oracle, 2017):

1. **SQL as a data definition language (DDL):** includes commands to create, change and drop database objects (such as tables) as well as define access rights to those database objects. They also allow you to add comments to the data dictionary.

2. **SQL as a data manipulation language (DML):** includes commands to manipulate data in existing objects like insert, update, delete and retrieve data within the database tables.

The table below lists the SQL **data definition commands** (Coronel, et al., 2011):

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

SQL Data Definition Commands (Coronel, et al., 2011, p. 221)

The table below lists the SQL **data manipulation commands** (Coronel, et al., 2011):

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 their original values
Comparison Operators	
=, <, >, <=, >=, <>	Used in conditional expressions
Logical Operators	
AND, OR, NOT	Used in conditional expressions
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
AVG	Returns the average of all values for a given column

SQL Data Manipulation Commands (Coronel, et al., 2011, pp. 221-222)

CREATING TABLES

To create new tables in SQL you use the CREATE TABLE statement. As its arguments, it expects all the columns we want in the table, as well as their data types. The syntax of the CREATE TABLE statement is shown below:

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

In the above, the constraints are optional and are used to specify rules for data in a table. Constraints that are commonly used in SQL are:

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

To create a table called Employee, for example, that contains five columns — EmployeeID, LastName, FirstName, Address, and PhoneNumber — you would do the following:

```
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. Note that these are all within brackets with a semicolon at the end. This is what indicates that the line is concluded.

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. Primary keys must contain unique values, and cannot contain null values. A table can only contain one primary key, however, the primary key may consist of single or multiple columns (as you may remember from the previous lesson).

You can add a **primary key** as the index 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 (ID,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: ID 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:

1. Do not specify the column names where you intend to insert the data. This can be done if you are adding values for all of the columns of the table. However, you should ensure that the order of the values is in the same order as the columns in the table. The syntax will be as follows:

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

   So, 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);
```

2. The other way is to specify both the column names and the values to be inserted. The syntax will be as follows:

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

   Therefore, to add an entry to the Employee table using this way, you would do the following:

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

SELECT

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;
```

column1, column2, ... are the column names of the table you want to select data from. The following example 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, however, you use the following syntax:

```
SELECT * FROM table_name;
```

The asterisk (*) means that we want to fetch all of the columns.

You can also use the ORDER BY command to sort the results in ascending or descending order. 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 by the FirstName column:

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

WHERE

Similar to an *if-statement*, 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 strings, however, you do not need to enclose numeric fields in quotes. 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 which contains the most sold albums of all time:

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 those of them 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 commands, that allows you a quick way to check commonly used queries. These are:

- 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

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;
```

FUNCTIONS

SQL has many functions that do all sorts of helpful stuff. 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

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

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

JOINS

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 column they both share.

Look at the two tables below:

VideoGame

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

GameDeveloper

ID	Name	Country
1	Blizzard	USA
2	Nintendo	Japan

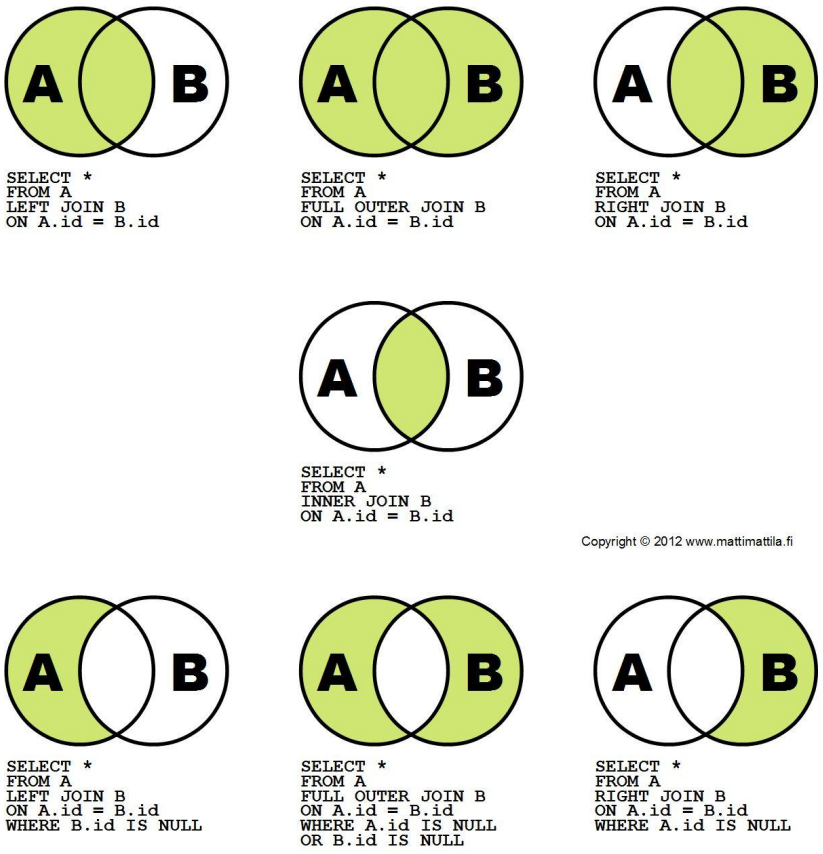
The VideoGame table contains information about various video games, while the Game Developer table contains information about the developers of the games. The VideoGame table has a DeveloperID column that holds the game developer ID and represents the ID of the respective developer from the GameDeveloper table. This logically links the two tables and allows us to 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;
```

The INNER JOIN is the simplest and most common type of JOIN, however, there are many other different types of JOINs in SQL, namely:

- 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 left or right table



Graphical representation of common SQL joins (Mattila, 2012)

ALIASES

Notice that in the VideoGame and GameDeveloper tables each has a column 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 GameDeveloper table:

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

This SQL statement shortens the query drastically by setting aliases to the table names:

```
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;
```

UPDATE

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 wanted 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 St	New York
2	Jackson Peters	124 River Rd	Berlin
3	Thomas Hardy	455 Hanover Sq	London
4	Kelly Martins	55 Loop St	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 St', City= 'Los Angeles'
WHERE CustomerID = 1;
```

DELETING ROWS

Deleting a row is a simple process. All that you need to do is select the right table and row 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 rows in a table without deleting the table:

```
DELETE * FROM table_name;
```

DELETING TABLES

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

```
DROP TABLE table_name;
```

For example, if we what 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;
```

SUPPORT FOR LARGE OBJECTS

Blobs

Blobs are very similar to the text datatype. They contain all the same lengths and very similar wording.

The key difference between text and blobs is that text data types will convert the data in your table to/from the character set that they have assigned to them. Blobs, however, convert their data from the object inserted into a binary data type.

So what does this mean and why are blobs useful?

Because of how the data in a blob field is converted, we are actually able to insert data that we may once have not been able to. A key example of this would be a picture.

Typically databases are known for only storing text/numerical data, but what about pictures? Every social media platform you go onto will usually have a profile picture, but how would they link this image to your account? This is where the blob field comes into play; it converts your photo into binary and, from there, inserts it into a database field. Once all that has been completed, it will then find the image and show it on the website.

Below is an example of where we could use the blob type.

```
CREATE TABLE users (
  user_id INT PRIMARY KEY AUTO_INCREMENT,
  user_name VARCHAR(25),
  profile_picture MEDIUMBLOB
);
```

Blobs Storage size

Blobs can store different size files (remember that it's not just limited to images, it can store almost any file type). Depending on the usage - where we might need the blob - we can decide on what blob type we will use.

Blob Type	File Storage Size	Example
TINYBLOB	>= 255 bytes	A pixel on an image
BLOB	>= 64 kb	Simple text files
MEDIUMBLOB	>= 16MB	Profile picture
LONGBLOB	>= 4GB	A Movie

TRIGGERS

What are triggers in SQL?

Think of triggers as a way of provoking certain actions based on a particular event taking place on a table or view. The timing of the event also plays an important role in the way in which a trigger is executed. Let us imagine for a second that we want to ensure that a certain action takes place when we perform a change in a table. There are typically two options for our triggering time: (a) BEFORE and (b) AFTER, where the 'before' keyword signifies that we wish for the change to take place before our SQL command runs, and 'after' signifies firing the trigger after a SQL command has run. In addition to these two special triggering-time keywords, we have the INSTEAD OF trigger which can be used only on views and not tables, like BEFORE and AFTER.

The life-cycle of trigger statements

When we start exploring the different types of triggers (as we will in the sections below), you will notice that the life-cycle of a trigger involves two key statements: **(a)** the **Create Trigger** and the **(b) Drop Trigger** statements. It is through these two statements that we can effect any change in our tables or views using triggers. So, let us have a look at the two statements individually and assess the level of importance they each bear.

The CREATE TRIGGER Statement

You will have noticed by now that triggers do not exist in isolation, and that in fact, they exist because of either tables or views. To begin with, the reason we would need to have a trigger is so we may better manage the processes going on in both our tables and views. Pretty much like querying a table (where statements like SELECT would be used), setting up a trigger will require us to firstly create the trigger. To achieve this, we use the **CREATE TRIGGER** statement which can then hold all the business logic/procedures we would like to implement.

Below is an example of the basic syntax for creating a trigger:

```
CREATE TRIGGER trigger_name
ON table_name
AS
{sql_statements}
```

What we have in the example above is a very fundamental CREATE TRIGGER statement which should be fired on a given table. Please note that since the above is just a template for the implementation of the CREATE TRIGGER statement, the **trigger_name** as well as the **table_name** and **sql_statements** have been labelled in accordance with the general structure of the CREATE TRIGGER statement.

The DROP TRIGGER Statement

Let us say you now wish to take down a trigger you had created for a certain table or view. To achieve this, you would have to evoke the DROP TRIGGER statement which will then ensure that the trigger you had created is taken down effectively, without causing any harm to the data in our table or view. Please see an example of the basic syntax for a DROP TRIGGER statement:

```
DROP TRIGGER [IF EXISTS] trigger_name
ON {DATABASE | ALL SERVER}
```

Notice that when we now drop a trigger, we introduce the IF EXISTS conditional which checks whether the specified trigger being sought to be dropped actually exists, and then will proceed to drop it if the conditional finds the trigger does exist.

The three types of triggers

- **Data manipulation language (DML) triggers**
  The easiest way to understand data manipulation triggers is by thinking of already existing data (tables), on which INSERT, UPDATE, and DELETE may be used. The example below covers a typical structure for a data manipulation language trigger using what is known as **T-SQL** or **Transact-SQL**. Think of T-SQL as a way of enforcing certain checks and balances on existing data, so that the data best reflects the business processes.

  ```
  CREATE TRIGGER trigger_name ON table_name
  FOR UPDATE
  NOT FOR REPLICATION
  ```

```
AS

BEGIN
  INSERT INTO another_table
  SELECT a_column
  FROM inserted
END
```

- **Data definition language (DDL) triggers**
  While the DML triggers are primarily concerned with enforcing changes on a table and view level, data definition language (**DDL**) triggers operate on the database and server level. This means that through DDL triggers we can effect change on a level a little higher than that which DML triggers target.

  Let us have a look at the general structure of a DDL trigger. Please see example below:

```
CREATE TRIGGER trigger_name
ON {DATABASE | ALL SERVER}
AS {sql_statement}
```

  You will notice that at this level, as mentioned above, the trigger created is solely concerned with effecting change at the database level, as opposed to the table level. How do we know this from looking at the example above? you may ask. The answer lies in the **ON** keyword, after which we have specified two possibilities, either the database or the server. Following this, we have the **AS** keyword which should then evoke SQL statements that will set in motion the specified changes.

- **Logon triggers**
  Logon triggers are used for effecting change right after a user has logged in onto a system. These types of triggers help establish strong security and ensure that access to databases is limited only to existing users of a system. Let us briefly have a look at the most basic implementation of a logon trigger, illustrated in the example below:

```
CREATE TRIGGER trigger_name
ON ALL SERVER FOR LOGON
AS
BEGIN
  {sql_conditionals}
```

```
  /*where sql_conditionals
    is all the sql if-else
    statements to check
    against a given
    user_name*/
END
```

You will notice that the logon trigger in the example sets out to carry out specific actions only when a certain username is accessing the server and database.

STORED PROCEDURES AND FUNCTIONS

A **function** in programming is a block of reusable code that performs a single or related action multiple times. Functions will usually take in data, process it and return it. Functions can be used over and over again making them a good alternative to having repeating blocks of code within your program (the same definition is true across all programming languages, not just SQL). A function in SQL would be a set of SQL statements that perform a specific task; functions are compiled and executed every time they are called within the server.

```
USE `library_db`;
DROP function IF EXISTS `book_description`;

USE `library_db`;
CREATE FUNCTION library_db.`book_description` (author_nm VARCHAR(50),
book_tl VARCHAR(50))
RETURNS VARCHAR(100) DETERMINISTIC
BEGIN
RETURN CONCAT(author_nm, ' - ', book_tl);
END;
```

This function will combine the author name and book title, and return it as one string. We pass in the word deterministic so that we always get the same answer whenever the inputs have been passed in; this is a form of exception handling to avoid unexpected results.

A **stored procedure** is a set of instructions or commands that are executed in order. In SQL a stored procedure is a prepared block of SQL code (queries or commands) that you can save, and that will be reused over and over again. Stored procedures are usually defined as a script that contains a series of commands that you can schedule to run at certain times. Stored procedures can chain multiple database statements like INSERT, UPDATE, SELECT and DELETE to perform

repetitive scripts. Stored procedures are pre-compiled, meaning that once they are compiled the compiled format is saved and executed as needed without being compiled each time.

```
USE `library_db`;
DROP procedure IF EXISTS `add_new_book`;

DELIMITER $$
USE `library_db`$$
CREATE PROCEDURE library_db.`add_new_book` (IN id_val INT, IN title_bk
VARCHAR(50), IN author_name VARCHAR(50), IN quantity INT)
BEGIN
      INSERT INTO books(id, title, author, qty)
    VALUES(id_val, title_bk, author_name, quantity);
END$$

DELIMITER;
```

This is a Stored Procedure that is used to add a new book to the library database. The IN keyword is an input parameter; by default when creating a function the keyword is always IN but for stored procedures the keyword can be IN, OUT, or INOUT. This is because we sometimes want to pass data into the procedure (IN), or get data out from the procedure (OUT) and in other cases both (INOUT).

The key differences between a stored procedure and a function are:

- A function must have a return value and it can only return a single value. A stored procedure can return nothing (0), a single value, or multiple values.
- Functions can only have input parameters, whereas stored procedures can have input and output parameters.
- We can call a function from a stored procedure, but a stored procedure can not be called from a function.
- A stored procedure allows us to use multiple SQL statements while a function will only allow a SELECT statement to be used within it.

These differences summarise the effect these routines can have on a database. A stored procedure can affect the state of the database and or alter the state of the environment parameters, while a function cannot do either.

MYSQL

MySQL is a popular open-source relational database management system owned by MySQL AB. While it is written in C and C++, it is based on SQL. It also supports many different languages and operating systems. MySQL makes using SQL easy because It has an intuitive interface that allows you to add, remove, modify, search and join tables in a database. You will learn more about MySQL in the next lesson where we will be installing it and starting to create and manipulate databases with it.

Compulsory Task

Answer the following questions:

- Go to the **w3schools website's SQL browser IDE**. This is where you can write and test your SQL code using their databases. Once you are happy with it, paste your code in a text file named **Student.txt** and save it in your task folder.
- Write the SQL code to create a table called Student. The table structure is summarised in the table below (Note that STU_NUM is 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 in question 1, write the SQL code to enter the first two rows of the table as below:

STU_NUM	STU_SNAME	STU_FNAME	STU_INITIAL	STU_STARTDATE	COURSE_CODE	PROJ_NUM
01	Snow	John	E	05-Apr-14	201	6
02	Stark	Arya	C	12-Jul-17	305	11

- Assuming all the data in the Employee table has been entered as shown below, write the SQL code that will list all attributes for a COURSE_CODE of 305.

STU_NUM	STU_SNAME	STU_FNAME	STU_INITIAL	STU_STARTDATE	COURSE_CODE	PROJ_NUM
01	Snow	Jon	E	05-Apr-14	201	6
02	Stark	Arya	C	12-Jul-17	305	11
03	Lannister	Jamie	C	05-Sep-12	101	2
04	Lannister	Cercei	J	05-Sep-12	101	2
05	Greyjoy	Theon	I	9-Dec-15	402	14
06	Tyrell	Margaery	Y	12-Jul-17	305	10
07	Baratheon	Tommen	R	13-Jun-19	201	5

- 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 all of the data inside a table, but not the table itself.

- Write the SQL code that will delete the Student table entirely.

Self-assessment table

Intended learning outcomes (SAQA US 11048)	Assess your own proficiency level for each intended learning outcome. Check each box when the statement becomes true- I can successfully:				
Write program code for database access	Understand and be able to explain the	Identify and use DDL SQL	Identify and use DML SQL	Create and manip	Query data and make

for a computer application using SQL (SO3)	difference between DDL and DML	commands	commands	ulate tables and indexes	queries more specific with criteria
	Understand and be able to use BLOBs in SQL	Understand and be able to use Triggers in SQL	Understand and be able to use functions in SQL	Understand and be able to use stored procedures in SQL	

Rate us

Share your thoughts

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

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

Click here to share your thoughts anonymously.

REFERENCES

Coronel, C., Morris, S., & Rob, P. (2011). *Database Systems* (9th ed., pp. 220-297). Boston: Cengage Learning.

Encyclopaedia Britannica. (2020). SQL | computer language. Retrieved 17 February 2020, from **https://www.britannica.com/technology/SQL**

IBM Corporation. (2010). Database SQL Programming, Triggers. (pp. 217-224). Accessed: 19 October 2021. **https://www.ibm.com/docs/el/ssw_ibm_i_71/sqlp/rbafy.pdf**

Mattila, M 2012, Visual representation of common SQL joins, flickr.com, accessed 17 February 2020,**https://www.flickr.com/photos/mattimattila/8190148857**.

Oracle. (2017). Types of SQL Statements. Retrieved 17 February 2020, from **https://docs.oracle.com/database/121/SQLRF/statements_1001.htm#SQLRF30042**

SQLServerTutorial. (2021). SQL Server Triggers. Accessed: 20 October 2021. **https://www.sqlservertutorial.net/sql-server-triggers/**