Advanced Higher
Computing Science

Database Development

SQL Implemtation

Name:______________________________

IMPLEMENTATION: DATABASE STRUCTURE

DATA DEFINITION LANGUAGE (DDL)

SQL is a standard language for accessing databases. There are two types of SQL command:

- Data Definition Language (DDL)
- Data Manipulation Langugage (DML)

Data Definition Language is used to define the structure of a database. DDL statements enable the creation and deletion of databases and tables.

The main DLL SQL statements you will use are

CREATE – Create a database or table
DROP – Delete a database or table

Database and Field Names

When using the Create statement, database and field names should be surrounded by the \`single quote\`.

This can be found on the keyboard, to the left of the 1 key.

CREATE

The CREATE statement is used to create a new database:

```
CREATE DATABASE IF NOT EXISTS `PeopleDB`;
```

CREATE is also used to create a new table (entity) together with its attributes:

```
CREATE TABLE IF NOT EXISTS `Persons`

(`PersonID` INT,  `LastName` VARCHAR(25), `FirstName`
VARCHAR(20), `Address` VARCHAR(60), `City` VARCHAR(20),

PRIMARY KEY (`PersonID`));
```

DROP

The DROP statement is used to delete a database or table:

```
DROP DATABASE IF EXISTS `PeopleDB`;
```

```
DROP TABLE IF EXISTS `Persons`;
```

In each of the CREATE and DROP examples above the, ‘IF EXISTS’ and ‘IF NOT EXISTS’ options do not have to be included.

However, without them, error messages will be returned if the statement cannot complete the command because of an attempt to create duplicate databases/tables or delete databases/tables that don’t exist.

CONSTRAINTS

When creating a table, there are a number of constraints that can be set on fields to make them perform in a particular way.

Constraints that can be used include:

- Primary Key
- Foreign Key
- Not Null
- Auto Increment
- Check

Primary Key

The Primary Key constraint is used to specify the primary key in a table. The primary key field specified must already have been defined in the field list.

```
CREATE TABLE IF NOT EXISTS `Persons`

(`PersonID` INT,  `LastName` VARCHAR(25), `FirstName`
VARCHAR(20), `Address` VARCHAR(60), `City` VARCHAR(20),

PRIMARY KEY (`PersonID`));
```

A compound key can be created by specifying two or more fields within the primary key parameters.

```
PRIMARY KEY (`orderNumber`,`productID`)
```

Foreign Key

The foreign key constraint is used to specify any fields that link to the primary key in another table.

```
CREATE TABLE IF NOT EXISTS `OrderProduct`(`orderNumber` INT,
`productID` VARCHAR(10),

PRIMARY KEY (`ordNum`,`productID`),

FOREIGN KEY(`productID`) REFERENCES `Product`(`productID`),

FOREIGN KEY(`ordNum`) REFERENCES `CustOrder`(`orderNumber`));
```

Each foreign key specified must identify the field in the current table that is a foreign key.

It must also reference the table and primary key field that the foreign key value will come from.

Not Null

The Not Null constraint is used to specify that a field is required.

```
CREATE TABLE IF NOT EXISTS `Persons`

(`PersonID` INT NOT NULL,  `LastName` VARCHAR(25) NOT
NULL, `FirstName` VARCHAR(20) NOT NULL, `Address`
VARCHAR(60) NOT NULL, `City` VARCHAR(20) NOT NULL,

PRIMARY KEY (`PersonID`));
```

The above code would specify that all fields are required.

Auto Increment

The Auto Increment constraint specifies that a field will automatically assign a number. The number assigned is incremental each time a new record is added.

```
CREATE TABLE IF NOT EXISTS `Persons`

(`PersonID` INT NOT NULL AUTO_INCREMENT,  `LastName` VARCHAR(25), `FirstName` VARCHAR(20), `Address` VARCHAR(60), `City` VARCHAR(20),

PRIMARY KEY (`PersonID`));
```

In the above code, the PersonID field will automatically be assigned a unique number for each new record.

Check

The Check constraint is used to validate the data that is entered into a field.

Check can be used for a **range check** or for **length check** when used with the length function.

```
CREATE TABLE Persons (

    ID int NOT NULL, LastName varchar(255) NOT NULL,

    FirstName varchar(255), Age int,

    CHECK (Age>=18 AND Age<=65)

);
```

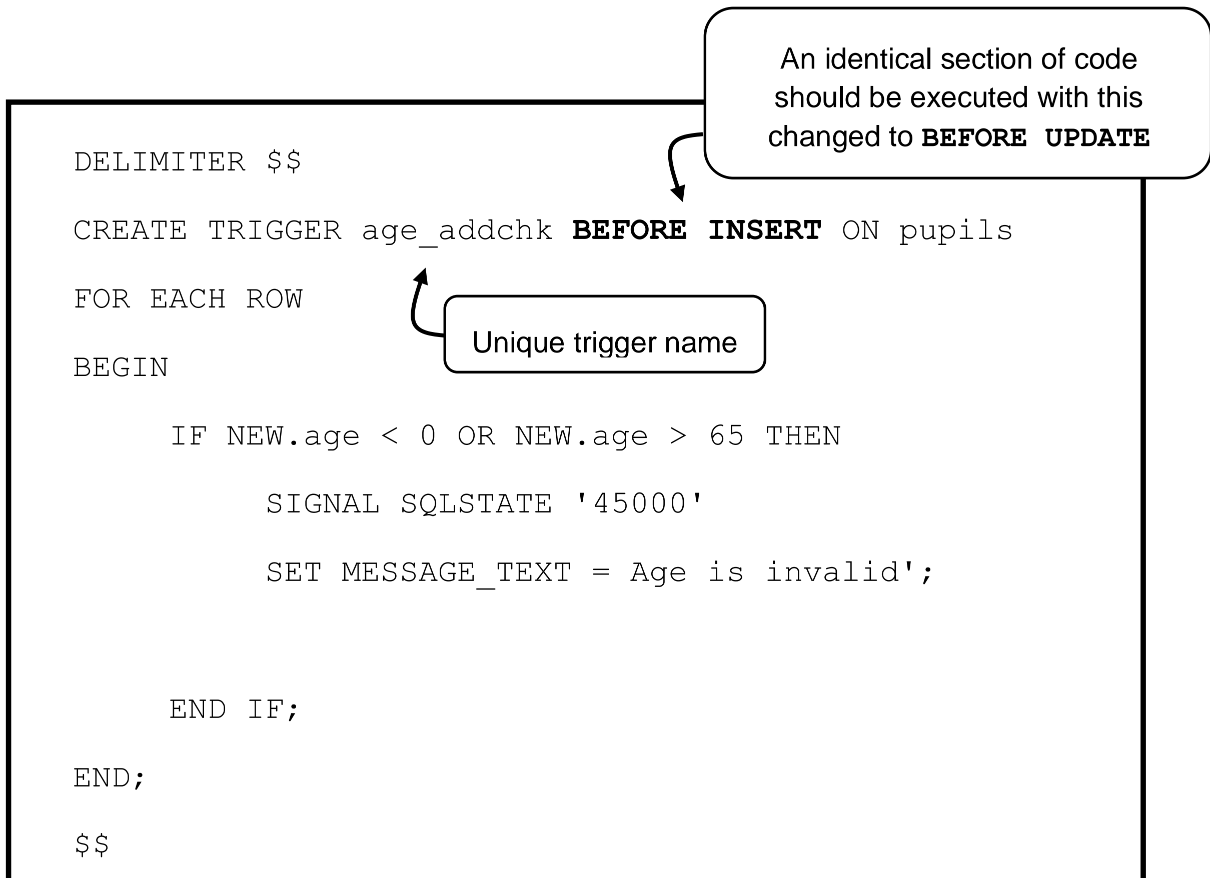
```
CREATE TABLE Persons (

    ID int NOT NULL, LastName varchar(255) NOT NULL,

    FirstName varchar(255), Phone varchar(11),

    CHECK (LENGTH(Phone=11))

);
```

NOTE: Check **does not work** in earlier MySQL versions. See **Triggers** for validation

Triggers (Check alternative)

Since the check constraint does not work in earlier versions of MySQL, an alternative method for validating data is to use triggers.

A trigger is a method of making the MySQL check the value being entered during an insert or update operation.



```
DELIMITER $$

CREATE TRIGGER age_addchk BEFORE INSERT ON pupils

FOR EACH ROW

BEGIN

     IF NEW.age < 0 OR NEW.age > 65 THEN

          SIGNAL SQLSTATE '45000'

          SET MESSAGE_TEXT = Age is invalid';


     END IF;

END;

$$
```


Only **one trigger** is allowed **for each operation** (insert or delete) **per table**. So if you want to check on multiple fields, the check must take place within the same trigger.

If we also want to validate phone number as well as age during an insert or delete, the complex condition must be added to as follows:

```
IF NEW.age < 0 OR NEW.age > 65 OR LENGTH(NEW.phone)<11 THEN
```

Trigger code should be executed once the table is created and before data is entered.

NEW SQL COMMANDS

Advanced Higher introduces some new SQL keywords that can be used within queries.

SELECT query commands:

- HAVING clause
- Subqueries

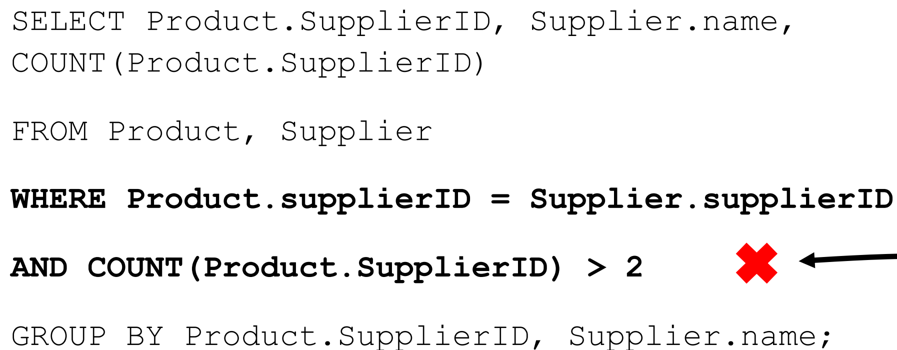
Logical Operators:

- IN
- NOT
- BETWEEN
- ANY
- EXISTS

HAVING

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

In the example below, we only want to display the names of suppliers who provide more than 2 different products. You might try to achieve this as shown below:



```
SELECT Product.SupplierID, Supplier.name,
COUNT(Product.SupplierID)

FROM Product, Supplier

WHERE Product.supplierID = Supplier.supplierID

AND COUNT(Product.SupplierID) > 2

GROUP BY Product.SupplierID, Supplier.name;
```

This does not work because aggregate functions are not allowed within a WHERE.

Instead, HAVING is used after the GROUP BY to state which results we would like to see displayed.

```
SELECT Product.SupplierID, Supplier.name,
COUNT(Product.SupplierID)

FROM Product, Supplier

WHERE Product.supplierID = Supplier.supplierID

GROUP BY Product.SupplierID, Supplier.name

HAVING COUNT(Product.SupplierID) > 2;
```

Task

Try running the SQL code above in your product supplier database.

Sub Queries

Subqueries are queries which are embedded within another query.

Previously, to display a list of products that cost more than the average product cost, you would have used two separate queries (a stored query).

Query 1

```
SELECT AVG(product.price)AS `average price`

FROM Product
```

Query 2

```
SELECT Product.name, Product.price

FROM Product, Query1

WHERE Product.price > Query1.`average price`

ORDER BY Product.price DESC
```

Using a subquery, the SQL can be combined.

```
SELECT Product.name, Product.price

FROM Product

WHERE Product.price > (SELECT AVG(product.price) FROM Product)

ORDER BY Product.price DESC
```

Task

Try running the SQL code above in your product supplier database.

Logical Operator: IN

The IN operator allows you to specify multiple values in a WHERE clause.

The IN operator is a shorthand for multiple OR conditions.

In the example below, we only want to display the names of shops in Dundee, Montrose or Paisley.

Previously, you might have written this as a complex OR condition.

```
SELECT Customer.shopName, Customer.city

FROM customer

WHERE City ='Dundee' OR City = 'Montrose' OR City ='Paisley'
```

However, the IN operator makes this much easier, especially if there are a lot of conditions.

```
SELECT Customer.shopName, Customer.city

FROM customer

WHERE City IN ('Dundee', 'Montrose','Paisley')
```

Task

Try running the SQL code above in your product supplier database.

Logical Operator: BETWEEN

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

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

In the example below, we only want to display the names of products that cost between £10 and £20.

Previously, you might have written this as a complex AND condition.

```
SELECT Product.productID, Product.name, Product.price

FROM Product

WHERE Product.price >=10 AND Product.price <=20
```

However, the BETWEEN operator makes this much easier, especially if there are a lot of conditions.

```
SELECT Product.productID, Product.name, Product.price

FROM Product

WHERE Product.price BETWEEN 10 AND 20
```

BETWEEN can also be used with text. The following SQL will identify product names which are alphabetically between King Doll and Raggedy Ann (inclusive).

```
SELECT Product.productID, Product.name, Product.price

FROM Product

WHERE Product.name BETWEEN "King Doll" AND "Raggedy Ann"
```

BETWEEN can also be used with dates.

```
SELECT *

FROM Customerorder

WHERE Customerorder.orderDate BETWEEN '2018-01-15' AND '2018-
03-15'
```

Task

Try running the SQL code above in your product supplier database.

Subquery Logical Operator: ANY

The ANY operator returns TRUE if any of the subquery values meet the condition.

The following SQL statement returns TRUE and lists the product names if it finds ANY records in the OrderProduct table that have a quantity of 100 or more:

```
SELECT Product.name

FROM Product

WHERE ProductID = ANY (SELECT ProductID FROM OrderProduct
WHERE Quantity >= 100);
```

Task

Try running the SQL code above in your product supplier database.

Subquery Logical Operator: EXISTS

The EXISTS operator is used to test for the existence of any record in a subquery.

The EXISTS operator returns true if the subquery returns one or more records.

In the example below, we want to display the names of suppliers that sell products costing less than £20

Previously, you might have written this using a GROUP BY.

```
SELECT Supplier.name

FROM Supplier, Product

WHERE Product.supplierID = Supplier.supplierID AND Price < 20

GROUP BY Supplier.name
```

The same result can also be achieved using a sub-query with the EXISTS operator.

```
SELECT Supplier.name

FROM Supplier

WHERE EXISTS (SELECT Product.name FROM Product WHERE
Product.supplierID = Supplier.supplierID AND Price < 20);
```

Task

Try running the SQL code above in your product supplier database.

Logical Operator: NOT

The NOT operator is used to display a result where a condition is not true.

NOT can be used on its own with the WHERE clause, or it can be placed in front of other logical operators to provide the opposite results.

WHERE

```
SELECT Product.productID, Product.name, Product.price

FROM Product

WHERE NOT Product.price >10
```

BETWEEN

```
SELECT Product.productID, Product.name, Product.price

FROM Product

WHERE Product.price NOT BETWEEN 10 AND 20
```

IN

```
SELECT Customer.shopName, Customer.city

FROM customer

WHERE City NOT IN ('Dundee', 'Montrose','Paisley')
```

EXISTS

```
SELECT Supplier.name

FROM Supplier

WHERE NOT EXISTS (SELECT Product.name FROM Product WHERE
Product.supplierID = Supplier.supplierID AND Price < 20);
```

Task

Try running the SQL code above in your product supplier database.

SQL QUERY DESIGN

SELECT QUERY DESIGN

Designing SELECT queries is done using tables like the ones below:

Field(s) and calculation(s)	People booked in July = SUM(numberInParty)
Table(s) and query	Booking
Search criteria	startDate between 01/07/19 and 31/07/19
Grouping	
Having	SUM(numberInParty) <=4
Sort order	startDate DESC, SUM(numberInParty) ASC

Field(s) and calculation(s)	ResortType, Number of Hotels = COUNT(*)
Table(s) and query	Resort, Hotel, Booking
Search criteria	resortType = "coastal"
Grouping	resortType
Having	COUNT(*) > 10
Sort order	COUNT(*) ASC

TESTING

GENERAL TESTING

See the *Development Process* notes on testing for details of:

- Integrative testing
- Usability testing based on prototypes
- Final testing
- End-user testing

DATABASE SPECIFIC TESTING

When testing a database, further testing should be conducted to ensure that:

- SQL implemented tables match the design

  Check that the tables that have been implemented match the design in terms of all data dictionary requirements, cardinality, optionality.

- SQL operations work correctly

  Check that all SQL queries carry out the desired functions and produce the correct output.

EVALUATION

GENERAL EVALUATION

See the *Development Process* notes on testing for details of:

- Fitness for purpose
- Maintainability
- Robustness

DATABASE SPECIFIC EVALUATION

When evaluating a database, further consideration should be given to:

- Accuracy of output

Check that the results of SELECT queries display the records expected.