Advanced Higher

SQL Operations

Candidates need to implement relational databases using SQL Data Definition Language (DDL) and Data Manipulation Language (DML) in the Advanced Higher course.

DDL

`CREATE` statement — used to create a database and the structure of each table in the database

`DROP` statement — used to remove individual tables from a database or even the entire database

DML

`INSERT` statement — used to populate a table by adding records (this was introduced at National 5)

`UPDATE` statement — used to edit values stored in database records (this was introduced at National 5 and extended at Higher)

`DELETE` statement — used to remove records from a database table (this was introduced at National 5)

In addition, Advanced Higher candidates should be able to describe, exemplify and implement SQL `SELECT` statements that make use of:

the `HAVING` clause

logical operators `IN`, `NOT`, `ANY`, `BETWEEN`, `EXISTS` in the `WHERE` or `HAVING` clause

a subquery in the `WHERE` clause

SQL data types

When using the SQL `CREATE` statement, SQL data types must be used.

Data type	Sample	SQL implementation	Comment
integer	32, -846	int	using a size parameter is optional; it is used to restrict the maximum display width
float	3.14	float(size, d)	the size parameter specifies the total number of digits displayed, while d specifies the number of digits after the decimal point
varchar	ABC123D	varchar(size)	the size parameter is mandatory, to restrict number of characters possible between 0 and 65535
date	2019-05-23	date	format is YYYY-MM-DD
time	09:12:47	time	format is hh:mm:ss

Information about each of these data types and examples of SQL statements are on the following pages.

CREATE statement

A database is defined as being a structured set of data. The first step in building an SQL database is to create the database structure using `CREATE DATABASE`.

```
CREATE DATABASE databaseName;
```

Once a database has been created, the structure for each table in the database needs to be built using `CREATE TABLE`.

```
CREATE TABLE tableName (
     fieldName1 dataType,
     fieldName2 dataType,
     ......
);
```

Validation constraints

The following can be specified for individual fields:

PRIMARY KEY: uniquely identifies each record in the table

```
fieldName dataType PRIMARY KEY
or
PRIMARY KEY (fieldName1)
or
PRIMARY KEY (fieldName1, fieldName2, ...)
```

FOREIGN KEY: links two tables together by referencing the primary key of another table

```
fieldName dataType FOREIGN KEY REFERENCES tableName
(fieldName)
or
FOREIGN KEY(fieldName) REFERENCES tableName (fieldName)
```

NOT NULL: ensures that a field always contains a value and is not left empty

```
fieldName dataType NOT NULL
```

CHECK: ensures that all values in a field satisfy a specific condition

```
fieldName dataType CHECK(fieldName condition)
```

AUTO-INCREMENT: automatically generates a unique number when a new record is inserted

```
fieldName dataType AUTO_INCREMENT
```

Additional notes on constraints

PRIMARY and FOREIGN KEY constraints

Some dialects of SQL allow the `PRIMARY` or `FOREIGN KEY` constraint to be applied in the clause used to identify the data type for the field; other dialects require the `PRIMARY` or `FOREIGN KEY` constraint to be applied in a separate clause.

Users should refer to the relevant documentation or reference guide to check the syntax for the version of SQL they are using.

If the primary or foreign key consists of multiple columns, users **must** specify them in a separate clause at the end of the `CREATE TABLE` statement.

CHECK constraint

Standard SQL provides the `CHECK` constraint, as described and exemplified in this appendix. However, the `CHECK` constraint is not provided in all dialects of SQL (for example, MS Access and MySQL do not support the use of `CHECK)`.

In the case of MySQL, the `CHECK` constraint is ignored and the intended data validation is not carried out. To implement the `CHECK` constraint in MySQL, triggers or views must be used.

Note: candidates should implement triggers or views within their project solution, as required; however, these constraints are not assessed in the Advanced Higher Computing Science course.

Users should refer to the relevant documentation or reference guide to check the syntax for the version of SQL they are using.

Applying multiple constraints

It is possible to apply several constraints to one field, for example:

```
fieldName dataType NOT NULL PRIMARY KEY
```

DROP statement

The DROP statement is used to drop or delete a whole database. Be careful when using this statement, as all the tables and data stored in them are removed and cannot be restored. This statement is often exploited by cyber criminals in SQL injections.

The DROP statement can be used to permanently remove an entire database.

```
DROP DATABASE databaseName;
```

It can also be used to delete individual tables from a database. Used in this format, the statement results in the complete loss of all data stored in the named table.

```
DROP TABLE tableName;
```

Note: The DROP statement is not supported in MS Access.

HAVING clause of a SELECT statement

The SQL HAVING clause is used in combination with the GROUP BY clause or an aggregate function, to restrict the returned rows to only those where the HAVING condition is true.

HAVING is used to filter records that work on summarised GROUP BY results. It was added to the SQL language because the WHERE clause cannot be used with aggregate functions. The HAVING clause is applied to grouped records, but WHERE is applied to individual records. Only groups that meet the HAVING criteria will be returned.

HAVING can also be used in combination with WHERE and ORDER BY clauses, for example:

the WHERE clause is used to restrict the rows that are returned from the tables(s)
the ORDER BY clause is used to sequence the rows in the answer table
the HAVING clause is used to filter summarised and/or aggregated data or grouped data

Note: using HAVING requires a GROUP BY clause to be present.

```
SELECT list of field names
FROM list of table names
WHERE condition
GROUP BY list of field names
HAVING condition
ORDER BY list of field names;
```

Logical operators

Logical operators are used, together with the comparison operators =, <, >, <=, >= and `LIKE`, in the `WHERE` clause of a `SELECT` query to form a condition that restricts the rows returned from the tables. At National 5, logical operators `AND` and `OR` were introduced. At Advanced Higher, five specialist operators are introduced.

NOT

This returns a record from the underlying tables when the specified condition is **not** true.

```
SELECT list of field names
FROM list of table names
WHERE NOT condition;
```

BETWEEN

This selects values that fall within a specified range of (inclusive) values.

```
SELECT list of field names
FROM list of table names
WHERE fieldname BETWEEN value1 AND value2;
```

IN

This allows multiple values to be specified as an alternative to multiple OR conditions.

```
SELECT list of field names
FROM list of table names
WHERE fieldName IN (value1, value2, ......);
```

subquery

```
SELECT list of field names
FROM list of table names
WHERE fieldName IN (SELECT statement);
```

ANY

This returns true if any of the subquery values meet the condition specified in the main query.

subquery

```
SELECT list of field names
FROM list of table names
WHERE fieldName operator ANY (SELECT statement);
```

EXISTS

This tests for the existence of records within the subquery and returns true when the subquery returns one or more records (this is very useful to obtain records that do not meet a certain condition).

subquery

```
SELECT list of field names
FROM list of table names
WHERE EXISTS (SELECT statement);
```

subquery

```
SELECT list of field names
FROM list of table names
WHERE NOT EXISTS (SELECT statement);
```

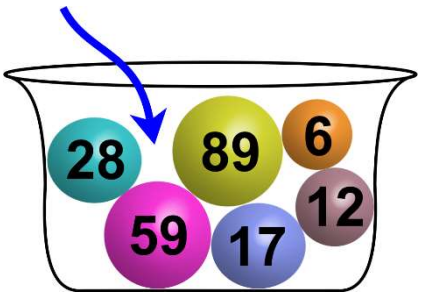
Additional notes on operators

ANY operator

The images below provide pictorial explanations of the SQL `ANY` operator.

Query 1: using the `ANY` operator, generates TRUE and so returns data to the main query.

`>ANY` means greater than at least one value, that is, greater than the minimum.

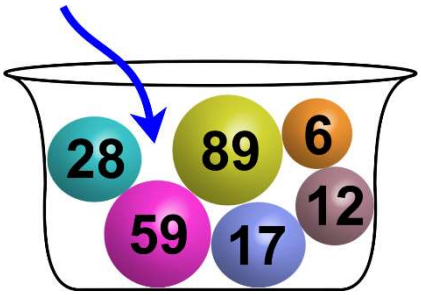
`WHERE 90 > ANY(` 

`);`

So `>ANY(28,59,89,17,6,12)` means greater than 6.
As 90>6 is true, data is returned.

Query 2: using the `ANY` operator, generates FALSE and so does not return data to the main query.

`<ANY` means less than at least one value, that is, less than the maximum.

`WHERE 90 < ANY(` 

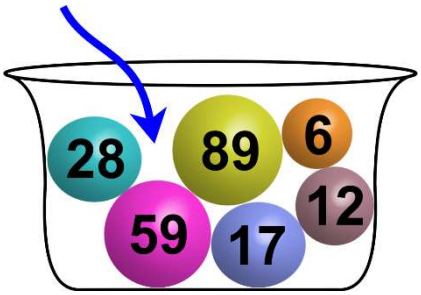
`);`

So `<ANY(28,59,89,17,6,12)` means less than 89.
As 90<89 is false, no data is returned.

Query 3: using the `ANY` operator, generates TRUE and so returns data to the main query.

`<ANY` means less than at least one value, that is, less than the maximum.

WHERE 4 < ANY(



);

So `<ANY(28,59,89,17,6,12)` means less than 89.
As 4<89 is false, data is returned.

EXISTS operator

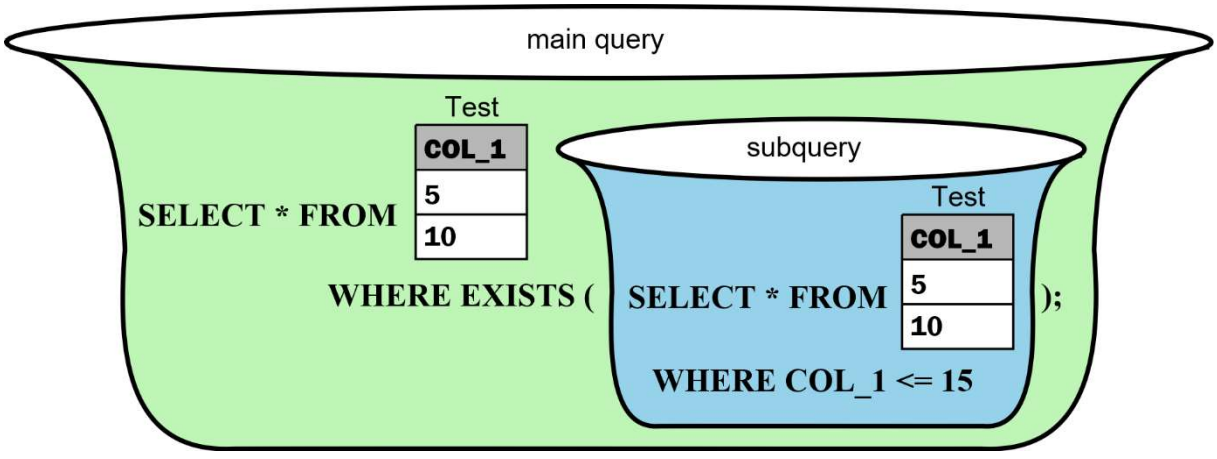
The images below provide pictorial explanations of the SQL EXISTS operator.

Query 4: general format of an SQL query that uses the EXISTS operator.

```
WHERE EXISTS (     subquery      );
```

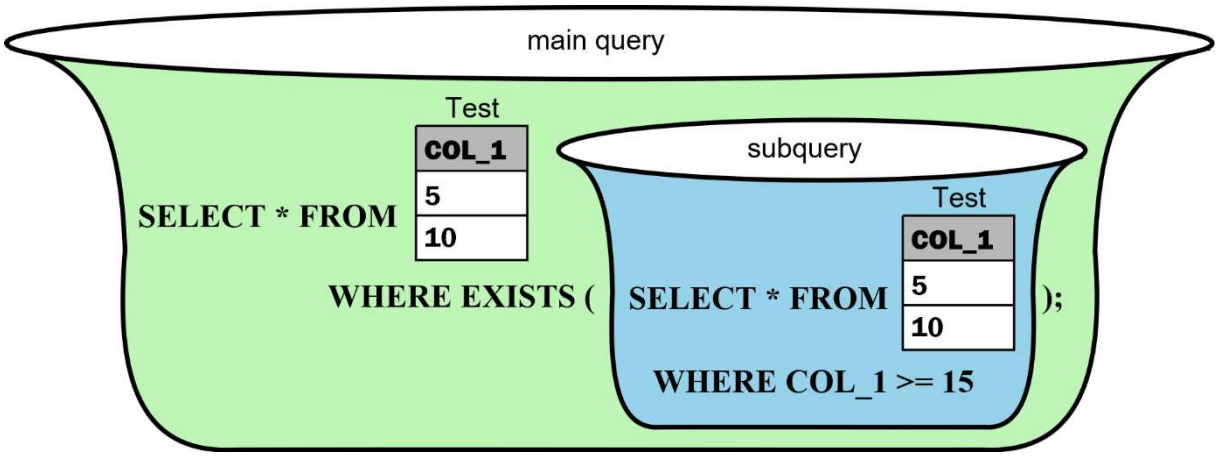
`EXISTS...`
is a comparison operator
is used in the `WHERE` clause to validate an 'it exists' condition
will tell whether a query returned results
returns a Boolean, (TRUE or FALSE)
returns TRUE if a subquery contains any rows

Query 5: using the EXISTS operator, returns TRUE.



The subquery contains more than one row, so it returns TRUE. Data is therefore returned from the main query.

Query 6: using the EXISTS operator, returns FALSE.



The subquery contains no rows, so it returns FALSE. No data is therefore returned from the main query.

Subquery in the WHERE clause of a SELECT query

A subquery is a query embedded within the `WHERE` clause of another SQL query. A subquery is sometimes referred to as an inner query or a nested query, and an SQL query is sometimes referred to as the outer query or the parent query.

The subquery executes before the main query, so the results can be passed to the main query as a condition to further restrict the data to be retrieved.

There are a few rules that subqueries must follow:

Subqueries must be enclosed within brackets.

Unless the main query has multiple fields in its `SELECT` clause, a subquery can have only one field in its `SELECT` clause.

The `BETWEEN` operator can be used within a subquery but cannot be applied to the results of a subquery returned to the main query.

Although an `ORDER BY` clause can be used with the main query, an `ORDER BY` clause cannot be used in a subquery; if it is needed, the `GROUP BY` clause can be used to perform the same function as the `ORDER BY` within a subquery.

Many subqueries return exactly one record (called single-value subqueries); the developer must check that this is the case, because an error will be generated if a subquery returns more results than expected.

Subqueries that return more than one row (called multiple-value subqueries), can only be used with multiple-value operators such as `EXISTS`, `IN` and `ANY`.

```
SELECT list of field names
FROM list of table names
WHERE fieldName OPERATOR
    (SELECT list of field names
     FROM list of table names
     WHERE condition)
ORDER BY list of field names;
```

Example queries: travel agency database

A travel agency uses a relational database to store details on a booking system.

It stores details of Scottish holiday resorts, hotels in each resort, customers and their bookings. These details are stored in four separate entities.

The attributes stored in each entity are shown below.

Resort	Hotel	Customer	Booking
resortID resortName resortType	hotelRef hotelName resortID * starRating seasonStartDate mealPlan checkInTime pricePersonNight	customerNo firstname surname address town postcode	hotelRef * customerNo * startDate numberOfNights numberInParty

SQL CREATE statement

The SQL statements below can be used to build the structure of the travel agency database. The full data dictionary for this database is in appendix 4: data dictionary.

```
CREATE DATABASE TravelAgency;


CREATE TABLE Resort (
        resortID int NOT NULL PRIMARY KEY,
        resortName varchar(20) NOT NULL,
        resortType varchar(20) NOT NULL CHECK (resortType
        IN('coastal','city', 'island', 'country'))
);


CREATE TABLE Hotel (
     hotelRef varchar(4) NOT NULL PRIMARY KEY,
     hotelName varchar(20) NOT NULL,
     resortID int NOT NULL,
     starRating int NOT NULL CHECK(starRating >=1 AND starRating <= 5),
     seasonStartDate date,
     mealPlan varchar(17) NOT NULL CHECK(mealPlan IN('Room Only',
     'Bed and Breakfast', 'Half Board', 'Full Board')),
     checkInTime time NOT NULL,
     pricePersonNight float(6,2) NOT NULL CHECK(pricePersonNight
     >=50 AND pricePersonNight <= 250),
     FOREIGN KEY (resortID) REFERENCES Resort(resortID)
);
```

```
CREATE TABLE Customer (
        customerNo int AUTO_INCREMENT PRIMARY KEY,
        firstname varchar(20) NOT NULL,
        surname varchar(20) NOT NULL,
        address varchar(40) NOT NULL,
        town varchar(20) NOT NULL,
        postcode varchar(8) NOT NULL
);


CREATE TABLE Booking (
        hotelRef varchar(4) NOT NULL,
        customerNo int NOT NULL,
        startDate date NOT NULL,
        numberNights int NOT NULL CHECK(numberNights >=1),
        numberInParty int NOT NULL CHECK(numberInParty >=1),
        PRIMARY KEY (customerNo, hotelRef, startDate),
        FOREIGN KEY (customerNo) REFERENCES Customer(customerNo),
        FOREIGN KEY (hotelRef) REFERENCES Hotel(hotelRef)
);
```

The following example queries match the examples in appendix 5: query design.

Queries making use of the HAVING clause

Query 7: display the resort name and number of hotels in any resort that has at least two hotels.

```
SELECT resortName, COUNT(*) AS [Number of Hotels]
FROM Resort, Hotel
WHERE Resort.resortID = Hotel.resortID
GROUP BY resortName
HAVING COUNT(*) >= 2;
```

Query 8: display the full name and the total cost of all bookings for each customer. The query should only list details of customers whose total cost exceeds £2000 and should list the details of the biggest spending customer first.

```
SELECT firstName, surname, SUM(pricePersonNight *
numberNights * numberInParty) AS [Total cost of all Bookings]
FROM Customer, Booking, Hotel
WHERE Customer.customerNo = Booking.customerNo
AND Booking.hotelRef = Hotel.HotelRef
GROUP BY firstName, surname
HAVING SUM(pricePersonNight * numberNights * numberInParty)
>= 2000
ORDER BY SUM(pricePersonNight * numberNights * numberInParty)
DESC;
```

Query 9: display the average price per person, per night for each holiday resort. Display only those resorts with an average price per person, per night that exceeds £100.

```
SELECT resortName, ROUND(AVG(pricePersonNight),2) AS [Average Price]
FROM Resort, Hotel
WHERE Resort.resortID = Hotel.resortID
GROUP BY resortName
HAVING AVG(pricePersonNight) > 100;
```

Queries using logical operators

Query 10: display the name and type of non-coastal resort, together with the name and meal plan for each hotel that meets these criteria.

```
SELECT resortName, resortType, hotelName, mealPlan
FROM Resort, Hotel
WHERE Resort.resortID = Hotel.resortID
AND NOT resortType = "coastal";
```

Query 11: display the full name and total number of bookings made by each customer who has made between two and four bookings.

```
SELECT firstName, surname, COUNT(*) AS [Total Bookings]
FROM Customer, Booking
WHERE Customer.customerNo = Booking.customerNo
GROUP BY surname, firstName
HAVING COUNT(*) BETWEEN 2 AND 4;
```

Query 12: display the surname, postcode, and town of customers who live in towns that begin with the letters ‘E’ through to ‘M’. The query should list customers in alphabetical order of town.

```
SELECT surname, postcode, town
FROM Customer
WHERE town BETWEEN "E" AND "M"
ORDER BY town;
```

Query 13: display the hotel name and meal plan for hotels that offer room only, half board or full board.

```
SELECT hotelName, mealPlan
FROM Hotel
WHERE mealPlan IN ("Room Only", "Half Board", "Full Board");
```

Query 14: display the name and type of resorts that are neither city nor country resorts.

```
SELECT resortName, resortType
FROM Resort
WHERE NOT resortType IN ("city", "country");
```

Queries with a subquery in the WHERE clause

Query 15: display the hotel name, star rating, and price per person for the most expensive hotel.

```
SELECT hotelName, starRating, pricePersonNight
FROM Hotel
WHERE pricePersonNight =
     (SELECT MAX(pricePersonNight)FROM Hotel);
```

Query 16: display the resort name, hotel name, and star rating of all hotels that have a below-average star rating.

```
SELECT resortName, hotelName, starRating
FROM Resort, Hotel
WHERE Resort.resortID = Hotel.resortID
AND starRating <
     (SELECT AVG(starRating)FROM Hotel);
```

Query 17: display the full name and postcode of the customer who booked the same hotel as the customer with ID 111.

```
SELECT firstName, surname, postcode
FROM Customer, Booking
WHERE Customer.customerNo = Booking.customerNo
AND NOT Customer.customerNo = 111
AND hotelRef =
     (SELECT hotelRef FROM Booking
      WHERE customerNo = 111);
```

Query 18: display the name and star rating of all hotels booked by the customer with ID 315.

```
SELECT hotelName, starRating
FROM Hotel
WHERE hotelName IN
     (SELECT hotelName FROM Hotel, Booking
      WHERE Hotel.hotelRef = Booking.hotelRef
      AND customerNo = 315);
```

Query 19: display the names and types of resort not booked by the customer with ID 315.

```
SELECT resortName, resortType
FROM Resort
WHERE resortName NOT IN
     (SELECT resortName FROM Resort, Hotel, Booking
      WHERE Resort.resortID = Hotel.resortID
      AND Hotel.hotelRef = Booking.hotelRef
      AND customerNo = 315);
```

Query 20: display the customer number, hotel reference, and booking cost for any booking that costs more than any bookings made by customers with surnames Lowden, Shawfair or Sheriffhall.

```
SELECT customerNo, Hotel.hotelRef,
pricePersonNight*numberNights*numberInParty AS [Booking Cost]
FROM Booking, Hotel
WHERE Booking.hotelRef = Hotel.hotelRef
AND pricePersonNight*numberNights*numberInParty > ANY
(SELECT pricePersonNight*numberNights*numberInParty
     FROM Booking, Hotel, Customer
     WHERE Booking.hotelRef = Hotel.hotelRef
     AND Booking.customerNo = Customer.customerNo
     AND surname IN ("Danderhall", "Lowden", "Shawfair"));
```

Query 21: display the details (hotel name, star rating, meal plan, and resort name) of all 3-star hotel bookings. The query should list the hotels in alphabetical order of meal plan.

```
SELECT hotelname, mealPlan, starRating, resortName
FROM Hotel, Resort
WHERE Hotel.resortID = Resort.resortID
AND starRating = 3
AND EXISTS
     (SELECT * FROM Booking
      WHERE Booking.hotelRef = Hotel.hotelRef)
ORDER BY mealPlan ASC;
```

Query 22: display the full name and address of customers who have never made a booking.

```
SELECT firstName, surname, address
FROM Customer
WHERE NOT EXISTS
     (SELECT * FROM Booking
     WHERE Customer.customerNo = Booking.customerNo);
```

Query 23: display the name, star rating, and total of nights booked for hotels that have:

a total number of customer nights booked that is more than the total number of nights booked by the customer with ID 290 (number of nights booked multiplied by number in party)

and

a star rating which is less than that of the hotel with the highest star rating

The query should list the hotels from lowest star rating to the highest.

```
SELECT hotelName, starRating, SUM(numberNights*numberInParty)
AS[Nights x Number in Party]
FROM Hotel, Booking
WHERE Hotel.hotelRef= Booking.HotelRef
AND numberNights*numberInParty >(
    SELECT SUM(numberNights*numberInParty) FROM Booking
    WHERE customerNo =290)
AND starRating < (SELECT MAX(starRating) FROM Hotel)
GROUP BY hotelName, starRating
ORDER BY starRating;
```