Questions matter

Teaching Guide: Relational databases and structured query language (SQL)

This resource is designed to help you to understand the key concepts surrounding section *3.7.1 Relational databases* and *3.7.2 Structured query language (SQL)* of the 8525 GCSE Computer Science specification.

Some sections (these are boxed like this) go beyond what is required by the specification, but this is a guide for teachers and not limited to just what students need to know.

Version 2.0
September 2025

Contents

You can use the title links to jump directly to the different sections of this teaching guide.

3.7.1 Relational databases

Databases

A database is a collection of data items, e.g. strings, dates, numbers, with some sort of structure to it. A database of students might hold, for each student:

- the student's last and first names (strings)
- the student's date of birth (date)
- the student's score in a test (integer).

There must be some way of associating one data item for one student with all the other data items **for that student**, or there's no way to get useful information out of the database.

In a programming language this association may be done in a number of ways. For example, the names, date of birth and test score for each student might be held in a `list` in Python, in a `struct` in C# and in a `Structure` in VB.NET.

When shown on paper or screen all the related data items are usually shown in a **row** (each data item, shown in a column, is known as a **field**):

James	*Stuart*	*12/06/2007*	*34*

and since the student's full name could be *James Stuart* or *Stuart James* each column is labelled with a descriptive name (the **fieldname**), and so it looks like a **table**:

LastName	*FirstName*	*DateOfBirth*	*TestScore*
James	Stuart	12/06/2007	34

The details of more students and their scores could be added to the table:

LastName	*FirstName*	*DateOfBirth*	*TestScore*
James	Stuart	12/06/2007	34
Peters	Ivan	14/09/2008	21
Fields	Meera	12/01/2008	21
Sohda	Ruki	14/01/2007	64
Stuart	James	06/12/2007	45
Sohda	Ruki	14/02/2007	39
Fields	Meera	12/01/2008	19
Stuart	James	12/06/2007	15

Some possible problems are now obvious: is the *Stuart James* in the first row the same student as the *James Stuart* in the last row, and it's just that someone entered the names in the wrong order? Is the *James Stuart* in the last row the same student as the *James Stuart* in the fifth row, but that someone entered the date in day/month/year order for one and month/day/year for the other?

Although the data above is an artificial example, it shows the problems of keeping data like this:

- *inconsistency:* repeating data items that are related (student last and first names, together with their date of birth) can lead to issues of data inconsistency
- *redundancy:* the related data is duplicated
- a student's details can't be added before they have a test score.

This leads to an important rule with databases:

every set of related data items (last name, first name and date of birth of a student in the example) **must only be stored once** in the database[1]

When all the data is kept in one spreadsheet, or one file[2], it is not usually possible to satisfy this important rule.

The results of more than one test could be stored in the same row as the student data, e.g.:

LastName	*FirstName*	*DateOfBirth*	*TestScore1*	*TestScore2*	*TestScore3*	*...*
James	Stuart	12/06/2007	34	19	27	
Peters	Ivan	14/09/2008	21	41	16	
Fields	Meera	12/01/2008	21			
Sohda	Ruki	14/02/2007	64	49	14	
Stuart	James	06/12/2007	45	43		
Sohda	Ruki	14/02/2007	39	11	21	
Fields	Meera	12/01/2008	19	16	17	
Stuart	James	12/06/2007	15			

However, this starts to get messy since some students have three tests, some two and some only one (and new column names need creating).

[1] This doesn't mean that there cannot have two identical scores (the two 21s above), or two identical dates of birth (the two 14/02/2007s above) in the database: a single data item isn't related to anything.

[2] Known as a flat file database

Relational databases

To solve the problems raised by a flat file database, *relational databases* were developed. In such a database all the related data items for a student, a test score or any other item is kept in one place. Technically this is called a *relation* (hence the name *relational database*) but the name *table* is most common[3].

So, to create a relational database for the student test records there must be one table to hold all the student details:

LastName	*FirstName*	*DateOfBirth*
James	Stuart	12/06/2007
Peters	Ivan	14/09/2008
Fields	Meera	12/01/2008
Sohda	Ruki	14/02/2007
Stuart	James	06/12/2007
Sohda	Ruki	14/02/2007
Fields	Meera	12/01/2008
Stuart	James	12/06/2007

Changes to make	*Row*
	1
	2
	3
	4
	5
These two rows are duplicates: remove them	6
	7
	8

At this stage, any duplications can be eliminated (remove the repeated records for *Meera Fields* and *Ruki Sodha* on rows 6 and 7) and the data in the remaining rows verified[4] to see if *James Stuart*, born *06/12/2007* is the same student as *James Stuart*, born *12/06/2007;* it is assumed they are the same student and the correct date of birth is *12/06/2007* so row 5 is removed.

LastName	*FirstName*	*DateOfBirth*
James	Stuart	12/06/2007
Peters	Ivan	14/09/2008
Fields	Meera	12/01/2008
Sohda	Ruki	14/02/2007
Stuart	James	06/12/2007
Stuart	James	12/06/2007

Changes to make	*Row*
	1
	2
	3
	4
Same student, date of birth is 12/06/2007	5
	8

This is the final *Student* table:

LastName	*FirstName*	*DateOfBirth*
James	Stuart	12/06/2007
Peters	Ivan	14/09/2008
Fields	Meera	12/01/2008
Sohda	Ruki	14/02/2007
Stuart	James	12/06/2007

Row
1
2
3
4
8

[3] Table refers to the actual implementation of the relation

[4] Probably by asking the student/students themselves.

The numbers in the right-hand column (initially 1 to 8) are **not** adjusted as data is removed from the table: these are used in the next stage of producing the relational database.

Keeping track of the scores

The original data was:

LastName	*FirstName*	*DateOfBirth*	*TestScore*
James	Stuart	12/06/2007	34
Peters	Ivan	14/09/2008	21
Fields	Meera	12/01/2008	21
Sohda	Ruki	14/01/2007	64
Stuart	James	06/12/2007	45
Sohda	Ruki	14/02/2007	39
Fields	Meera	12/01/2008	19
Stuart	James	12/06/2007	15

The student data is now replaced with the number in the right-hand column of the student relation (the *Student* table) and produce the *StudentTest* table:

StudentNumber	*TestScore*
1	34
2	21
3	21
4	64
8	45
4	39
3	19
8	15

Row
1
2
3
4
5
6
7
8

This is the *StudentTest* table and, although there is more than one test score for some students (students 3, 4 and 8), those students' details are **not** repeated.

This solves the first and second problems with the original data (*inconsistency* and *redundancy*).

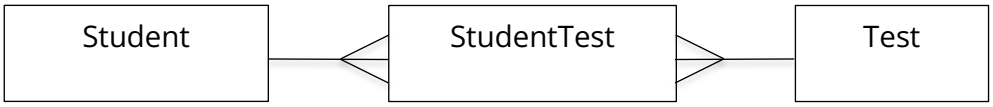
Since the data for a student is held separately (in a *Student* table) from their scores (which are in a *StudentTest* table) a student's details can be added to the *Student* table without their having any test scores which solves the third of the problems with the original data.

The linking between the relations in the database can be shown using a diagram:

called an *entity-relationship diagram*[5]. The ends of the line connecting two entities say something about the relationship, in this case that *one* student (single line end) has (or can have) *many* test scores (crow's feet end): a *one-to-many* relationship.

Although *one-to-one* and *many-to-many* relationships can exist, the first is unusual in databases and the second is implemented in databases using two one-to-many relationships and a linking table, as shown in the next paragraph.

To hold more details about the test that each score was for, adding these details (subject, date, maximum mark etc) to the *StudentTest* table would be recreating all the problems of the initial design and so a new *table* is created, called *Test*, and the relationship between the entities then becomes:



This can be written as "*One student can take many tests, and one test can be taken many times: a student test score is the score that a particular student achieved on a particular test."*

The *StudentTest* relation will then need changing to include a link to the *Test* as well as the existing link to the *Student*.

[5] The word *entity* usually refers to the 'real-world' thing that is being stored in the database.

Linking entities or tables

When linking two tables together, it is vital that one row in the table **at the *one* end** of the relationship can be uniquely identified, and this was the purpose of the numbers in the right-hand columns (labelled *Row*) that were added when converting from a flat file to a relational database: this is known as a *primary key* and **must** be unique for each record (row). If each student is allocated a student number when they enrol, this would be an ideal primary key.

Sometimes the primary key for a record may be made up of more than one of the other items of data (the **fields**) in that **record** (it is then known as a **composite primary key**) but **it must not be possible** for there to be any duplications of the primary key.

For example, choosing the combination of *last name*, *first name* and *date of birth* as a primary key could lead to problems: while it's very unlikely that there would be two or more students with the same names and date of birth **it is possible**. For this reason, primary keys are often artificially created (usually by the database program): when a new record is added the database program will use the next available number[6].

A particular value of the *primary key* is used to link from a record in the table (entity) at the *many* end of the relationship to just one record in the table (entity) at the *one* end of the relationship; it is known as a *foreign key* in the *many* table.

Another name for the *primary key* is *entity identifier* (a particular value identifies one, and only one entity), and often the field name for the primary key in a table is created just by adding *ID* to the name of the table (or entity), e.g. *StudentID, StudentTestID*.

[6] It's important that, when a record is deleted, its primary key is never reused. Although these can be thought of as row or record numbers, once rows or records are deleted this isn't strictly true: this is why the right-hand column was not renumbered in the example.

The original flat file database now comprises the *Student* table:

StudentID	*LastName*	*FirstName*	*DateOfBirth*
1	*James*	Stuart	12/06/2007
2	*Peters*	Ivan	14/09/2008
3	*Fields*	Meera	12/01/2008
4	*Sohda*	Ruki	14/02/2007
8	*Stuart*	James	12/06/2007

and the *StudentTest* table:

StudentTestID	*StudentID*	*TestScore*
1	*1*	*34*
2	*2*	*21*
3	*3*	*21*
4	*4*	*64*
5	*8*	*45*
6	*4*	*39*
7	*3*	*19*
8	*8*	*15*

If a *Test* table were added to the database, this might be:

TestID	*Subject*	*Date of test*	*MaximumMark*
1	*Physics*	12/06/2025	*40*
2	*Chemistry*	13/05/2025	*25*
3	*Computer Science*	19/05/2025	*100*

and then the *primary key* of the *Test* table could be added to the *StudentTest* table:

StudentTestID	*StudentID*	*TestID*		*TestScore*
1	*1*	*1*		*34*
2	*2*	*2*		*21*
3	*3*	*2*		*21*
4	*4*	*3*		*64*
5	*8*	*3*		*45*
6	*4*	*1*		*39*
7	*3*	*2*		*19*
8	*8*	*2*		*15*

The test score with *StudentTestID 5* is for the student with *StudentID 8* (*James Stuart*, born on *12/06/2007*) who took the test with *TestID 3* (*Computer Science* on *19/05/2025)* getting *45* marks (out of a possible *100*).

Glossary of key terms

Key term	Definition
attribute	an individual fact, detail or characteristic of an entity (also commonly called a field)
composite primary key	a primary key for a record that consists of the values in more than one field in that record
database	a persistent store of related information
entity	a thing, person, object or relationship about which data can be collected
field	a single piece of information about an entity (see attribute). A field has a *name* and a *type*, for example the field holding a student's first name would be called *FirstName* and be of type *text* or *string*
flat file database	a database containing a single table
foreign key	a field in a table which contains the value of a *primary key* in another table: since the primary key is unique this means that the foreign key identifies, or links to, just one record in the other table
inconsistency	if data about an entity is duplicated within a database then it is possible that the data is inconsistent, for example the names *Stuart James* and *James Stuart* in the example
primary key	a field within a table that contains a unique value for each record in that table A primary key may be composite, that is made by combining the values of more than one existing field in each record
redundancy	the duplication of data about an entity within a database
record	a collection of fields/attributes about a particular entity in a table
relational database	a database with multiple tables linked using primary and foreign keys
table	a collection of records. The name of a table is usually that of the entity whose details are being stored in that table, for example *Student*, *Test*, *StudentTest*

3.7.2 Structured query language (SQL)

SQL

When working with a relational database, tasks that will need doing include:

- *creating* data by inserting (adding) one or more records/rows to a table
- *retrieving* data by finding one or more records/rows that meet some required condition, for example students with test scores of more than 75 on tests that have a maximum of 100
- *updating* (*changing*) one or more records/rows, for example adding 2 to all test scores for students who took a particular test
- *deleting* one or more records/rows, for example all the test scores for a particular student

All these tasks need to be done quickly and efficiently, and SQL (Structured Query Language) is the programming language that is most commonly used to manipulate information within a relational database.

Retrieval of data from a database - the `SELECT` statement or query

The **`SELECT`** statement is the simplest of SQL statements and is used to retrieve information within a database. It is often known as a *query*.

A **`SELECT`** statement comes in four main parts or clauses:

Clause	Description
SELECT	the information to retrieve from the database
FROM	the table or tables containing the information
WHERE	the conditions or criteria you need any records to satisfy to be included in the retrieved information
ORDER BY	how the data is sorted when it is retrieved.

Example:

```
SELECT
    FirstName, LastName
FROM
    Student
WHERE
    StudentID > 3
ORDER BY
    LastName ASC, FirstName DESC
```

This example will return only the `FirstName` and `LastName` of all the students from the Student table who have a `StudentID` that is greater than 3. The results will be shown ordered in ascending order of `LastName` and then, if two or more students have the same

`LastName` by descending order of `FirstName`, so `James Stuart` would be displayed before `Arthur Stuart`.

Multiple or cross table queries

When the data required comes from more than one table there are two ways it can be retrieved. The original method uses and extends the `WHERE` clause but the more modern method is to use a `JOIN` clause.

For the purposes of the examination, it will not matter which method students use in their answers.

Using the `WHERE` clause (original method)

When combining data from multiple tables the same `SELECT` statement is used, but conditions are added to the `WHERE` clause to make sure the linking between tables is carried out correctly using the primary and foreign keys:

```
SELECT
    Student.FirstName, Student.LastName, Test.Subject,
    StudentTest.TestScore,
    Test.MaximumMark
FROM
    Student, Test, StudentTest
WHERE
    Student.StudentID = StudentTest.StudentID AND
    StudentTest.TestID = Test.TestID
```

In the `WHERE` clause the names of any linking fields which are the same in both the *one* and the *many* sides of the relationship **must** be preceded by the name of the table containing them, so `Student.StudentID` and `StudentTest.StudentID` rather than `StudentID = StudentID`.

Using a `JOIN` (modern method)

```
SELECT
   Student.FirstName, Student.LastName, Test.Subject,
   StudentTest.TestScore, Test.MaximumMark
FROM
  Student
  INNER JOIN
     (StudentTest
      INNER JOIN
        Test
      ON
        StudentTest.TestID = Test.TestID)
  ON
     Student.StudentID = StudentTest.StudentID
```

The shaded `INNER JOIN` between `StudentTest` and `Test` is done first (because of the brackets, just as in arithmetic) and that resulting combination of data is then `INNER JOIN`ed with `Student`.

Again, where the linking fields in the `INNER JOIN` have the same name they **must** be preceded by the table name, so `StudentTest.TestID = Test.TestID` and, as in the original method, `Student.StudentID` and `StudentTest.StudentID`.

Both versions of this example will return the `FirstName` and `LastName` of all of the students in the `Student` table **who have taken a test** (any students added to the `Student` table **without** having taken a test will not be retrieved) with the subject of the test, the score they got in the test and the maximum mark they could have got.

All such results will be shown (there is no WHERE clause to restrict which are retrieved) and the results will not be sorted (there is no ORDER BY clause).

Ordering information

When retrieving data from a database, it is possible to specify the order in which the data is shown. This is done using the **ORDER BY** clause.

The **ORDER BY** clause has two options, namely `ASC` and `DESC` (if you have **ORDER BY** but don't give either `ASC` or `DESC` then `ASC` is used):

Ordering data in ascending order	Ordering data in descending order
ORDER BY Lastname ASC	ORDER BY Lastname DESC

You can also order data by multiple fields:

```
SELECT
    FirstName, LastName, DateOfBirth
FROM
    Student
ORDER BY
    DateOfBirth DESC, LastName
```

In the above example, data is first organized by `DateOfBirth` (youngest first[7]) and then by `LastName` if two or more students share the same birthday.

As with linking information, field names in the ORDER BY clause that appear in more than one table in the query **must** be preceded by the table name. In this example (which has only one

[7] This can seem confusing. The most recent (largest) date of birth shown is the one for the youngest student, the least recent (smallest) is for the oldest student.

table) it is not needed (although it would do no harm to write `Student.DateOfBirth DESC, Student.LastName`).

Creating data in a database

In order to add information into a table within a database using SQL, a `INSERT` statement is used.

The SQL `INSERT` statement has the following structure:

```
INSERT INTO
    table(field1, field2, field3, …, fieldn)
VALUES
    (value1, value2, value3, …, valuen)
```

The number of fields given **must** equal the number of values given.

Example

```
INSERT INTO
    Student(StudentID, LastName, FirstName, DateOfBirth)
VALUES
    (15, 'Smith', 'Bob', #2008/07/12#)
```

Quotation marks (either single or double) are used when entering string or text data, and dates are surrounded by `#`, single or double quotation marks (in Access) or single quotation marks in other databases[8].

If you are inserting a value for **every** field in a record, as in the example above, then you can shorten the `INSERT` statement (by leaving out the list of fields) to:

```
INSERT INTO
    Student
VALUES
    (15, 'Smith', 'Bob', #2008/07/12#)
```

In this case the order of the fields in the `VALUES` clause **must** agree with the order of the fields in the table, so:

```
INSERT INTO
    Student
VALUES
    ('Smith', 'Bob', #2008/07/12#, 15)
```

would not work because it would be trying to put text (`'Smith'`) in the `StudentID` field, a date in the `FirstName` field and a number in the `DateOfBirth` field[9].

[8] When entering dates always use the `YYYY/MM/DD` format, for example `'2026/12/07'` as there is then no confusion between 7th December and 12th August.

[9] While some database might convert some of these the resulting data would be incorrect.

Deleting data from a database

In order to delete information from a database using SQL, a `DELETE` statement is used.

The SQL `DELETE` statement has the following structure:

```
DELETE FROM
    Table
WHERE
    Condition
```

Example

```
DELETE FROM
    Student
WHERE
    StudentID = 15
```

If a record with a `StudentID` of `15` existed then this record (and no other, because `StudentID`s are unique) would be deleted. If there were no record with that `StudentID` then nothing would happen, so running the above query a second time will not delete any records.

WARNING: using `DELETE` or a real database can be dangerous. Omitting the `WHERE` clause or having a condition that is true for every record, means that **every** record in a table will be deleted. Make sure that the conditions are right by first trying them in a `SELECT` statement before changing it to a `DELETE` statement.

Updating (changing) data in a database

In order to update or change information within a database using SQL, an `UPDATE` statement is used.

The SQL `UPDATE` statement has the following structure:

```
UPDATE
    Table
SET
    field1 = value1, field2 = value2, …
WHERE
    Condition
```

Example

```
UPDATE
    Student
SET
    FirstName = 'Robert', LastName = 'Smithers'
WHERE
    StudentID = 15
```

Because the condition in the `WHERE` clause looks for a specific value of `StudentID` (the **primary key**) only one record (at most) will be updated.

You can also use an `UPDATE` statement to update multiple records at once by using a condition in the `WHERE` clause that matches more than one record. For example:

```
UPDATE
    StudentTest
SET
    TestScore = TestScore + 3
WHERE
    TestID = 3
```

would update all the scores for students who had taken the test with `TestID` ***3*** by adding *3* to the original score[10].

[10] If this is run by mistake it can always be undone by changing the `+3` to `−3` and running it again.