DATABASE DEsign and development

National 5 Computing Science

Contents

[Introduction 3](#_Toc498235826)

[Analysis 6](#_Toc498235826)

[Design 10](#_Toc498235827)

[Data Protection Act 1998 10](#_Toc498235828)

[Entities 11](#_Toc498235829)

[Attributes 11](#_Toc498235830)

[Relationships 11](#_Toc498235831)

[Entity relationship diagrams 12](#_Toc498235832)

[Data dictionary 16](#_Toc498235833)

[Designing a solution to a query 20](#_Toc498235834)

[Example designs 23](#_Toc498235835)

[Implementation 25](#_Toc498235836)

[Referential integrity 25](#_Toc498235837)

[SQL. 26](#_Toc498235838)

[Testing 28](#_Toc498235839)

[Evaluation 28](#_Toc498235840)

## Introduction

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| pupilID | surname | firstname | class | subjectID | subject | level | day  |
| 1 | Davies | Paul | 4A2 | ML01 | French | Nat 4 | Monday |
| 2 | Jackson | Catherine | 5I2 | AD01 | Drawing | Nat 5 | Wednesday |
| 3 | Davies | Paul | 4A2 | EN01 | English | Nat 4 | Thursday |
| 4 | Jones | Susan | 4S1 | SC01 | Chemistry | Nat 5 | Friday |
| 5 | Davies | Paul | 4A2 | HU01 | Geography | Nat 4 | Tuesday |

Here is part of a database. A Flat Database file has one table. A file is made up of Records. A record is all the related information about one person or thing. Records contain fields. A field holds the individual pieces of data like surname or age as above.

This is unnecessary as some information is stored more than once (eg the same phone number). This also results in an increased file size for the database. Using flat file databases can lead to three very specific problems:

* Insert anomaly
* Delete anomaly
* Update anomaly

**Insert anomaly**

In the above example, it is not possible to add a new supported study to the database without also having to add a pupil who will attend at the same time. The table expects a pupil’s details and the details of a supported study to be stored together as one record.

At the moment, there is no way to add for example the Maths supported study without also having to add a pupil attending. This problem is known as an insert anomaly.

**Delete anomaly**

A delete anomaly is the opposite of an insert anomaly. When a delete anomaly occurs it means that you cannot delete data from the table without having to delete the entire record.

For example, if we want to remove Susan Jones from the table, we would also need to remove all data that is stored about the chemistry support study. This means we would lose data that we might not want to lose.

**Update anomaly**

Take a look at the table shown above again. If the Paul Davis surname is spelt incorrectly and should be changed to Davies it would need to be changed in three different records.

If the change only happened in one of the three records, then an update anomaly would have taken place.

In small tables it can be easy to spot update anomalies and make sure that changes are made everywhere. However, large flat file tables would often contain thousands of records, meaning that it is difficult to make changes to every record. Update anomalies lead to inaccuracy and inconsistency in a database.

**Linked database**

The way to avoid insert, delete and update anomalies is to design relational databases that use two or more linked tables to store data. In our example, it is possible to split the data into two different tables and to link them together using a field that is relevant to both tables.

The first table could store details on each Pupil in S4 to S6. This table stores their pupil ID, firstname, surname and class and subject ID. The pupil ID is the primary key and is a unique value - two pupils cannot share the same pupil ID.

**Pupil Table**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| pupilID | surname | firstname | class | subjectID |
| 1 | Davies | Paul | 4A2 | ML01 |
| 2 | Jackson | Catherine | 5I2 | AD01 |
| 3 | Davies | Paul | 4A2 | EN01 |
| 4 | Jones | Susan | 4S1 | SC01 |
| 5 | Davies | Paul | 4A2 | HU01 |

**Study Table**

|  |  |  |  |
| --- | --- | --- | --- |
| subjectID | subject | level | day  |
| ML01 | French | Nat 4 | Monday |
| AD01 | Drawing | Nat 5 | Wednesday |
| EN01 | English | Nat 4 | Thursday |
| SC01 | Chemistry | Nat 5 | Friday |
| HU01 | Geography | Nat 4 | Tuesday |

This table contains four fields, subjectID, subject, level, and day. subjectID is the primary key and is unique for each subject. The tables are linked together because the primary key in this table (subjectID ID) is also needed in the pupil table – so that you know which supported subject each pupil will take part in. In the pupil table, subjectID is known as a foreign key.

When you use linked tables in a relational database you reduce the likelihood of insert, update and delete anomalies. In this example, moving to linked tables means:

* A new subject can be added without having to add pupils alongside it.
* If a pupil was now deleted, such as Paul Davis, you would no longer lose the information held about the supported study subjects.
* If the Paul Davis’s name should be Davies has to be changed, it would only have to change once in the pupil table.

**Field Types**

When creating tables in a database, it is necessary to define the type of data that will be held in each field.

**Text field**

Text fields store words, numbers and other characters such as punctuation marks. A forename, surname and postcode are all examples of text fields.

**Number field**

Stores both whole numbers and decimal numbers. Within database packages, it is possible to define the number of values that can appear after a decimal place if storing decimal numbers

**Date field**

Dates can be stored using the date field type such as Date of birth, the date of exams are stored as date fields. Within database packages, it is possible to set the format for how a date appears on screen e.g. dd/mm/yyyy. In Some packages use one field type to store date/time

**Time field**

A time field stores a specific time, hh/mm/ss. Within database packages, it is often possible to set the format for how a time appears on screen. Some packages use one filed type to store date/time

**Boolean field**

A Boolean field stores either true or false, often showing on screen as Yes or No. A school could use this to see if a pupil is absent or not.

## Analysis

The development of a database has several phases. These include:

**1. Analysis**

**2. Design**

**3. Implementation**

**4. Testing**

**5. Evaluation**

**End user requirements**

During the **analysis phase**, it is necessary to determine **end user** and **functional requirements**.

**End user requirements** should clearly state **the tasks that users wish to perform**.

For example, if creating a database for Award trips in the school, it would be necessary to speak to the teachers who would make use of the database.

Developers could ask that each user group lists the tasks that they wish to be able to complete. In this example, two different end user groups exist, office staff and teaching staff.

Office staff report that they must be able to:

* add a pupil to a trip
* search for pupil data
* delete pupil from trip

Teaching staff report that they must be able to:

* add medical notes
* output pupils with medical issue
* output details of pupil who have not paid for a trip
* output details of pupils who have been awarded a particular trip
* output details of all pupils who have been awarded a trip
* output details of all pupils who have not been awarded a trip

**Functional requirements**

**Functional requirements** should **relate to the tasks** **that the database system will perform**, usually in the form of searching and sorting (queries).

Functional requirements may also provide detail around the data that must be held in the database.

In the example of the award trip, the functional requirements could include the following:

A table should exist storing award details including fields for:

* awardID
* organiser
* place
* cost
* date
* time

The award data table should be linked to a pupils records table.

The pupil records table should include the following:

* pupil ID
* first name
* surname
* house
* year group
* age
* awarded
* award ID
* health issue
* medical details
* medication
* paid

Simple and complex queries should exist to allow for searching and sorting on both tables.

Sorts should be generated to quickly sort data in both tables.

Forms will need to exist to allow both user groups to add or amend records.

In reality the example above may involve a more complex database than this.

What is shown above is just an illustration to highlight how the end user and functional requirements may be determined.

Actual implementation would likely involve more than two tables to avoid anomalies, such as a medical table. However, at National 5 level, only two tables need to be implemented and understood.

**Categorising requirements**

There is no absolute standard in how to collate or categorise end user and functional requirements.

However, it can be helpful to take the view that:

* **end user requirements** can be **summarised for each user group**
* **functional requirements** take account of the **inputs, processes and outputs necessary** for the **database to function properly**

**End user requirements**

There are two groups of end-users in the award trip example, each with different needs:

* office staff who must be able to:
	+ add a pupil to a trip
	+ search for pupil data
	+ delete pupil from trip
* teaching staff who must be able to:
	+ add medical notes
	+ output pupils with medical issue
	+ output details of pupil who have not paid for a trip
	+ output details of pupils who have been awarded a particular trip
	+ output details of all pupils who have been awarded a trip
	+ output details of all pupils who have not been awarded a trip

**Functional requirements**

|  |  |  |
| --- | --- | --- |
| **Input (Forms)** | **Processes (Queries)** | **Outputs (Reports)** |
|  |  |  |
| **office group** |  |  |
| Sign up pupil | Search/sort by surname, house, yeargroup, pupilID, award ID | Pupil details |
| Edit pupil details | Amend trip | Appoint trip |
| Delete pupil | Remove pupil from trip |  |
|  |  |  |
| **teacher group** |  |  |
| Add medical notes | Search/sort by medical issue, award ID  | Medication, medical details |
| Generate pupils not paid | Search paid | Pupils detail not paid |
| Generate pupils award trip | Search place, awarded | Pupils detail awarded |
| Generate pupils award trip | Search awarded | Pupils detail awarded |
| Generate pupils not award trip | Search awarded | Pupils detail not awarded |

##

## Design

### Data Protection Act 1998

Companies and organisations like the government hold a lot of personal data about individuals. To protect this data and people’s rights when storing, transmitting and exporting it, the Data Protection Act was passed by Parliament to protect peoples’ privacy

In short the law states that:

* individuals will be asked if data may be held about them (prior consent)
* data should be accurate - if data held about a data subject is incorrect they can demand it is changed
* data should be used only for limited, specifically stated purposes
* the data user is obliged to keep the data safe, secure and up to date, and not to hold the data for longer than is necessary

**Data Subjects** are the individuals who have data kept about them. If a data subject wishes to inspect data held about themselves they can request to see their own personal data

**Exceptions to the Act**

If any data is held by the police, security forces or the Inland Revenue, then you are denied access to the data

**Data Users** are the employees that work for organisations/companies that have access to / make use of the information held about individuals

**Data Controllers** are the companies, organisations or people who store personal information about individuals. Data Controllers must be registered with the **Data Protection Commissioner**. Data Controllers must apply for permission to store data and state what data they want to store and for what purpose

**Data Protection Commissioner** is the person you contact if you wish to raise a complaint about the accuracy of data. They in turn contact the Data Protection Registrar who oversees the administration of the act

**The Data Protection Register** is an information system freely available to the public that holds information about who holds information about members of the public

### Entities

An **entity** is a single person, place, or thing about which data can be stored e.g. a car, Spain, flight 4A07

### Attributes

An entity will have attributes. In the example below, the car entity has four attributes: make, model, colour, registrationNumber

make: Vauxhall

model: Zafira

colour: grey

registrationNumber: S012 GHJ

The attributes have values that can be edited.

When the data is entered into a database, the attributes become the **fields** of a **record**. If we made a database of 1200 cars, we would have 1200 records, each with four fields

### Relationships

We are surrounded by entities that have relationships with each other. The relationships can be described as

* ONE to ONE
* ONE to MANY
* MANY to MANY

For National 5 we only look at ONE to MANY relationships

For example, ONE car has MANY repairs. You might think – but many cars have many repairs – but that is not what the relationship is saying. It means that one specific repair (for example, the brake pads on Vauxhall Zafira SO12 GHJ replaced on 5.3.17) was for ONE car but that ONE car might have MANY different (unique) repairs

has

CAR

REPAIR

For example, the relationship between mother and children is ONE to MANY. You might think – but many mothers have many children – but that is not what the relationship is saying. It means that one specific child (for example, Zoltan Albard born on 12.8.2001) is the child of ONE mother but that ONE mother might have MANY different (unique) children

has

CHILD

MOTHER

### Entity relationship diagrams

A flat file database is shown below – it has 9 fields. It would be better if the database was split into two linked tables with a relationship created between them otherwise modifying duplicated data can lead to errors. For example, if Miss Chess changed her name to Mrs Board, every entry for her would have to be amended which could easily lead to errors (loss of data integrity)

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| ***activityID*** | ***title*** | ***ageRange*** | ***venue*** | ***teacherID*** | ***Name*** | ***teachingRoom*** | ***subject*** | ***fullTime*** |
| *ASC2745* | *Revision* | *11-18* | *Room 101* | *T87* | *Miss Chess* | *214* | *Maths* | *false* |
| *Susp1024* | *Water sports* | *14-18* | *Porty beach* | *T56* | *Mr Splash* | *303* | *PE* | *true* |
| *Susp1013* | *Adventure* | *11-15* | *Crazy Ways* | *T14* | *Ms Yeehah* | *504* | *Geography* | *true* |
| *Susp1067* | *Make a mural* | *11-18* | *PHS* | *T39* | *Mrs Colourist* | *612* | *Art* | *false* |
| *Susp1018* | *Cycling* | *16-18* | *Glen Tress* | *T44* | *Mr Spokes* | *405* | *CDT* | *true* |
| *ASC2543* | *Board games* | *11-14* | *Room 102* | *T87* | *Miss Chess* | *214* | *Maths* | *false* |
| *ASC2716* | *Cycling* | *14-17* | *PHS* | *T44* | *Mr Spokes* | *405* | *CDT* | *true* |
| *ASC3487* | *Adventure* | *15-18* | *PHS* | *T56* | *Mr Splash* | *303* | *PE* | *true* |

If the data shown in the flat file database above was held in two tables as a relational database, the **entity relationship** would be shown as follows:

ONE teacher offers MANY activities

offers

TEACHER

ACTIVITY

N.b.

 the word describing the relationship is written into the diagram i.e. “offers”

 the “crow’s foot” shows the “many” aspect

The full entity relationship diagram (ERD) would be:



**NOTE** It is useful to remember:

 “The **primary key** from the **ONE table** **goes into** the **MANY table** as the **foreign key**”

Here’s another example:



### Data dictionary

A data dictionary should include the following:

#### Entity names

For the school activities example, the entity names are Activity and Teacher

#### Attribute names

The attribute names for the school activities example are:

*activityID*, *title*, a*geRange*, v*enue*, *teacherID*, *name*, *teachingRoom*, *subject*, *fullTime*

#### Key - Primary or foreign key

***Primary key:*** this is the attribute that **uniquely identifies each entity** e.g. accountNumber, ID, carRegNumber, etc.

In the school activities example, the primary keys are activityID for the Activity table and teacherID for the Teacher table

***Foreign key:*** this is used to link one table to another table in a database i.e. set up relationships between tables. N.b. the foreign key in one of the tables will be the primary key in another table

Helpful hint: “the primary key from the ONE table is used as the foreign key in the MANY table”

#### Attribute (field) types

##### Text

e.g. EH9 1EF, 07652 915845, Jane

(N.b. a telephone number is text – it has a lead zero, may contain a space and would not be used in calculations)

##### Number

e.g. 95, 0.645 Also £32.95 (formatted as currency)

##### Date

e.g. 15/04/2017 (formatted as dd/mm/yyyy)

##### Time

 e.g. 23:30, 06:00 (formatted as hh:mm)

##### Boolean

true/false

#### Attribute size

For example, a text attribute might be maximum 20 characters in length

#### Validation

Validation is the process of checking that data entered into a system is of the **correct type and structure**

1. Presence check

 checks that data has been entered/data field not left blank

1. Restricted choice

 the user presented with a list of options to choose from e.g. from a drop-down menu

1. Field length check

 specifies the size of a field e.g. number of characters allowed in a field

1. Range check

 specifies the range of values a field can hold e.g. range 0 to 100 inclusive

NB

Restricted choice-limits its choice to the given options and removes human error when inputting.

Altogether, the data dictionary might look like this:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Entity** | **Attribute name** | **Key** | **Type** | **Size** | **Unique** | **Required** | **Validation** | **Sample data** |
| ***Activity*** | ***activityID*** | *PK* | *Text*  | *8* | *Yes* | *Yes* | *Presence check* | *ASC2745* |
| ***title*** |  | *Text*  | *20* |  | *Yes* |  | *Revision* |
| ***ageRange*** |  | *Text* | *5* |  |  |  | *11-18* |
| ***venue*** |  | *Text*  | *15* |  | *Yes* |  | *Room 101* |
| ***teacherID*** | *FK* | *Lookup* |  | *No* | *Yes* | *Lookup from TEACHER* | *T87* |
|  |  |  |  |  |  |  |  |  |
| ***Teacher*** | ***teacherID*** | *PK* | *Text*  | *4* | *Yes* | *Yes* | *Presence check* | *T87* |
| ***name*** |  | *Text*  | *40* |  |  | *Restricted choice* | *Miss Chess* |
| ***teachingRoom*** |  | *Text*  | *3* |  | *Yes* | *Field length check* | *214* |
| ***subject*** |  | *Text*  | *20* |  |  |  | *Maths* |
| ***fullTime*** |  | *Boolean* |  |  |  |  | *false* |

Data dictionary

Data dictionaries contain the following metadata.

* name of each entity
* name of all attributes associated with each entity
* type of data that will be held by each attribute
* size of each attribute (taking account of the overall file size to be held in RAM or backing storage)
* indication of attributes that will be used as primary or foreign keys during implementation
* validation rules that are to be applied to attributes



Here’s another example:

N.b. The foreign key is not unique in the MANY table. For example, there may be many orders with the same customerID on them

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Entity** | **Attribute name** | **PK/FK** | **Type (size)** | **Unique** | **Required** | **Validation** | **Format** | **Sample data** |
| ***Customer*** | ***customerID*** | *PK* | *Text (8)* | *Yes* | *Yes* | *Presence check* |  | *C01456AR* |
| ***title*** |  | *Text (3)* |  |  | *Restricted choice* |  | *Mrs* |
| ***name*** |  | *Text (30)* |  | *Yes* |  |  | *Amy Bryce* |
| ***address*** |  | *Text (150)* |  | *Yes* |  |  | *16 Main St, Craigfar, CK9 1RF* |
| ***age*** |  | *Number* |  |  | *Range check* *(16 to 100)* | *000* | *37* |
| ***Order*** | ***orderID*** | *PK* | *Text (4)* | *Yes* | *Yes* | *Presence check* |  | *T87* |
| ***orderDate*** |  | *Date* |  |  | *Restricted choice* | *dd/mm/yy* | *12/05/17* |
| ***totalToPay*** |  | *Number* |  | *Yes* | *Field length check* | *£0000.00* | *£36.95* |
| ***customerID*** | *FK* | *Lookup* | *No* | *Yes* | *Lookup from CUSTOMER* |  | *C01456AR* |

### Designing a solution to a query

#### Multiple tables

Suppose the flat file database shown here …

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| ***activityID*** | ***title*** | ***ageRange*** | ***venue*** | ***teacherID*** | ***Name*** | ***teachingRoom*** | ***subject*** | ***fullTime*** |
| *ASC2745* | *Revision* | *11-18* | *Room 101* | *T87* | *Miss Chess* | *214* | *Maths* | *false* |
| *Susp1024* | *Water sports* | *14-18* | *Porty beach* | *T56* | *Mr Splash* | *303* | *PE* | *true* |
| *Susp1013* | *Adventure* | *11-15* | *Crazy Ways* | *T14* | *Ms Yeehah* | *504* | *Geography* | *true* |
| *Susp1067* | *Make a mural* | *11-18* | *PHS* | *T39* | *Mrs Colourist* | *612* | *Art* | *false* |
| *Susp1018* | *Cycling* | *16-18* | *Glen Tress* | *T44* | *Mr Spokes* | *405* | *CDT* | *true* |
| *ASC2543* | *Board games* | *11-14* | *Room 102* | *T87* | *Miss Chess* | *214* | *Maths* | *false* |
| *ASC2716* | *Cycling* | *14-17* | *PHS* | *T44* | *Mr Spokes* | *405* | *CDT* | *true* |
| *ASC3487* | *Adventure* | *15-18* | *PHS* | *T56* | *Mr Splash* | *303* | *PE* | *true* |

… has been implemented as a **relational database** with **two tables** of **records** as shown here…

activityID: ASC2745

title: Revision

age Range: 11-18

venue: Room 101

teacher ID: T87

activityID: Susp1024

title: Water sports

age Range: 14-18

venue: Porty beach

teacher ID: T56

activityID: Susp1013

title: Adventure

age Range: 11-15

venue: Crazy Ways

teacher ID: T14

activityID: Susp1067

title: Make a mural

age Range: 11-18

venue: PHS

teacher ID: T39

activityID: Susp1018

title: Cycling

age Range: 16-18

venue: Glen Tress

teacher ID: T44

activityID: ASC2543

title: Board games

age Range: 11-14

venue: Room 102

teacher ID: T87

activityID: ASC2716

title: Cycling

age Range: 14-17

venue: PHS

teacher ID: T44

activityID: ASC3487

title: Adventure

age Range: 15-18

venue: PHS

teacherID: T56

teacherID: T18

name: Mr Donne

teaching Room: 100

subject: English

full Time: Yes

teacherID: T87

name: Miss Chess

teaching Room: 214

subject: Maths

full Time: No

teacherID: T62

name: Mr Kildare

teaching Room: 101

subject: History

full Time: Yes

teacherID: T56

name: Mr Splash

teaching Room: 303

subject: PE

full Time: Yes

teacherID: T14

name: Ms Yeehah

teaching Room: 504

subject: Geography

full Time: Yes

teacherID: T39

name: Mrs Colourist

teaching Room: 612

subject: Art

full Time: No

teacherID: T75

name: Miss Overby

teaching Room: 518

subject: HE

full Time: No

teacherID: T28

name: Mr Spokes

teaching Room: 405

subject: CDT

full Time: Yes

teacherID: T44

name: Mr Spokes

teachingRoom: 405

subject: CDT

fullTime: true

The head teacher wants to create a **report** that shows the names (in **alphabetical order**) of all the **full time** teachers who offer **adventure** activities. Which teachers would the report show?

The report would show *Mr Splash and Ms Yeehah*

There is a programming language called **SQL** that is used to query, insert, update and modify data in a database. SQL stands for **Structured Query Language**.

First step: **design a solution to the SQL query**. Here is an example:

|  |  |
| --- | --- |
| **Field(s)** | name, fullTime, title |
| **Table(s)** | Teacher, Activity |
| **Search criteria** | fullTime = true AND title = “Adventure” |
| **Sort order** | name ASC |

#### Fields and tables

When a database is created using software the entity sets become the tables, the attributes become the fields and each entity becomes a record in a table

|  |  |
| --- | --- |
| **Design stage** | **Implementation stage** |
| Entity set | Table |
| Attribute | Field |
| Entity | Record |

#### Search criteria

Simple search (find)- searching on one field only

e.g. search for records with field (forename = “James”)

Complex search - searching on two or more fields

e.g. (surname = “McRury”) AND (purchased = true)

#### Sort order

Sort in alphabetical, numerical, chronological (time) order (ascending/descending)



This table has been sorted on **one** field

It has been sorted on the field **forename** in alphabetical ascending order (A to Z). You can see that there are two Gregors, three Jacks and two Mhairis but, within those groups, no other field has been sorted – the Gregor and Mhairi surnames are in order but the Jack surnames aren’t and similarly with their ages

Sorting on more than one field: sorts on a first field, then sorts within the groups on the second field



This table has been sorted on two fields (attributes) – first on forename in alphabetical ascending order and then on age in numerical ascending order

How to tell: if you look at the Gregors, Jacks and Mhairis, their surnames are sometimes ascending, sometimes descending but their ages (in their little groups) are all increasing

### Example designs

Data dictionary



SQL query design examples

1.

2.

## Implementation

Advantages of a relational database over a flat-file database:

In the **flat file** database shown previously, if Miss Chess changed her name to Mrs Chequer, then someone would have to change every entry for Miss Chess. They are likely to make mistakes and, especially in a large database, this would quickly lead to the data becoming **inaccurate** and **unreliable** (a loss of data integrity)

In the **relational** database, there would be just one record for Miss Chess in the Teacher table. There would be two activities linked to her record by having her teacherID on each activity record and, if her name needed to be amended (e.g. she became Mrs Chequer), then only the record about her would need to be amended and all the activities linked to her would automatically see her new name

So the advantage of a **relational database with linked tables** is that it **avoids modification errors for data that is duplicated** that would occur if the data was held in a flat database. Links can be made from data in one table to data that already exists in another table to provide different pathways through the data. The implementation should ensure that the tables are linked together in the most efficient and useful way so that any necessary reports can be generated

### Referential integrity

Integrity means trustworthiness

A database has **referential** integrity if the table relationships are consistent. In other words, any foreign key field must agree with the primary key that is referenced by the foreign key. If a foreign key has a value that doesn’t exist as a primary key value in the linked table then the database will produce errors resulting from a lack of referential integrity

For example, in the schools activities database, if one of the activity records was trying to reference a teacher record with a teacherID that didn’t actually exist in the Teacher table then there would be a lack of referential integrity

###

**Enforce Referential Integrity**

If you enforce referential integrity, the following rules apply:

You cannot enter a value in the foreign key field of a related table if that value doesn't exist in the primary key field of the primary table — doing so creates orphan records.

You cannot delete a record from a primary table if matching records exist in a related table. For example, you cannot delete an award record from the award table if there are pupils assigned to that award in the Pupil table. You can, however, choose to **delete** a primary record *and* all related records in one operation by selecting the **Cascade Delete Related Records** check box.

You cannot change a primary key value in the primary table if doing so would create orphan records. For example, you cannot change an awardID in the Award table if there are records assigned to that award in the Pupil table. You can, however, choose to **update** a primary record *and* all related records in one operation by selecting the **Cascade Update Related Fields check box**.

We will learn about deletions and updating later using **SQL**.

### SQL

There is a programming language called **SQL** that is used to query, insert, update and modify data in a database. It stands for **Structured Query Language**. The following examples have been copied from W3Schools.com

#### SELECT

FROM

 For example: SELECT customerName, city

 FROM Customers;

 customerName, city are **fields** in the **table** called Customers

WHERE

 For example: SELECT customerName

 FROM Customers

 WHERE country = 'Mexico ';

 customerName and country are **fields** in the **table** called Customers

AND, OR, <, >, =

 For example:

SELECT \* // \* means all the records

FROM Customers
WHERE country='Germany' AND city='Berlin';

SELECT \*

FROM Customers
WHERE city='Berlin' OR city='München';

SELECT name

FROM Students

WHERE age >= 14 AND age <= 18;

ORDER BY (max of two fields)

 For example:

SELECT \*

FROM Customers
ORDER BY country DESC;

SELECT \*

FROM Customers
ORDER BY country ASC, customerName DESC;

#### INSERT

 For example:

INSERT INTO Customers (customerName, city, country)
VALUES ('Cardinal', 'Stavanger', 'Norway');

#### UPDATE

 For example:

UPDATE Customers
SET contactName = 'Alfred Schmidt', city= 'Frankfurt'
WHERE customerID = 1;

#### DELETE

 For example:

DELETE FROM Customers
WHERE customerName='Alfreds Futterkiste';

#### INNER JOIN (Equi-join between tables)

 For example:

SELECT Orders.orderID, Customers.customerName
FROM Orders
INNER JOIN Customers ON Orders.customerID = Customers.customerID;

 For example:

SELECT Teacher.name

FROM Teacher

INNER JOIN Activity //this makes the link between the two tables

ON Teacher.teacherID = Activity.teacherID

WHERE (Teacher.fullTime = true) AND (Activity.title = “Adventure”);

## Testing

This would involve testing the SQL operations to ensure that they work correctly and produce the correct output

A test table could be drawn up showing the test data, the expected output and the actual output in the form of screenshots. This test table could be referred to in support of your evaluation

## Evaluation Stage

Finally, the complete database is evaluated in terms of whether it is **fit for purpose**. This means making a judgement as to whether it meets its **specification** by delivering its **functional requirements** (see Analysis). You would also judge whether it meets the **end-user requirements**. You would consider your test results and draw conclusions from them.

Evaluate the solution in terms of:

1. Fitness for purpose
2. Accuracy of output

**Fitness for purpose**

There are different SQL commands. Depending on the SQL operation depends on how you asses ‘fitness for purpose’.

**From:** identify the entity(ies) - table(s).

**Where:** the correct search criteria has been used.

**Order By:** the records have been displayed in the correct order.

**Insert:** the correct record has been inserted with the correct fields.

**Update:** the correct record has been updated and the correct filed(s) has changed accordingly.

**Delete:** the correct record has been deleted.

**Equi-join:** which fields should have the same value to link the two tables.

**Accuracy of output**

Accuracy of output is the correct fields have been included in the display. All the expected fields are displayed only. There are no fields included which should not be displayed.

**Select:** the correct fields have been displayed.

Are all the fields included in the display which were expected to be displayed – this is **accurate output**.

Are there any fields included which should not have been displayed – this is **inaccurate output**.

**Fitness for Purpose**

**Fitness for purpose – you must explain an appropriate response from the list below:**

**Identify correct table(s) - From command**

Has the correct table been identified to obtain the fields?

**Searching – Where command**

Has the correct search criteria being used - what should it be if incorrect.

**Sorting – Order By command**

Have the field been sorted correctly in ascending or descending order on the correct fields?

**Adding a new record - Insert command**

Has the correct record has been inserted with the correct fields.

**Amending -Update command**

Has the correct record has been updated and the correct fields changed and secondly has any other fields been amended which should not have change?

**Delete command**

Has the correct record has been deleted. Have any other records been deleted which should not have been deleted

**Equi-join command**

Have the correct fields been used so that the same value in the two table are linked?

**You must state:**

The script allowed these operations to carry out and the **output was accurate** (see test results) so my database is **fit for purpose** in that sense.

**OR**

The script did not allowed these operations to carry out and the **output was inaccurate** (see test results) so my database is not **fit for purpose** in that sense.

**Accuracy of output**

**Display particular fields from the table(s)**

**Select command**

Have the correct fields been displayed.

Are all the fields included in the display which were expected to be displayed – this is **accurate output**.

Are there any fields included which should not have been displayed or are there any fields that should have been displayed but have not been displayed – this is **inaccurate output**.

**Example 1– Selection and Order By (Search and Sort) command**

A list of all award trips which cost less than £10 displaying fields organiser, place and time. The list should be sorted in descending order of place.

**Query Design and Testing.**

**Design**

Query Design: Return a list of all trips which cost less than £10.

|  |  |
| --- | --- |
| Field(s) | organiser, place, time |
| Table(s) | Award |
| Criteria | cost < 10 |
| Sort Order | place DESC |

**Solution 1 - not fit for purpose and not accurate output**

**Implementation**

**SELECT Award.organise, Award.place,** **Award.cost**

**FROM Award**

**WHERE cost<= 10**

**ORDER BY Award.place ASC;**

**Testing**

**Predicted result**

The predicted result shows what the developer expects the query to return.

| **organiser** | **place** | **time** |
| --- | --- | --- |
| Mr Rae | Pizza and Move | 09:30 |
| Mrs Stokes | M&D’s | 09:30 |

**Actual Result**

The actual result of the query is then recorded. It can be compared to the expected result to make sure that the query has executed as expected.

| **organiser** | **place** | **cost** |
| --- | --- | --- |
| Mrs Stokes | M&D’s | £5.00 |
| Mr Rae | Pizza and Move | £2.00 |
| Miss Simpson | Soar into Braehead | £10.00 |

**Assessment on result** – actual output does not match expected output for the following reasons:

**Fitness for purpose**

**Searching – Where command**

The correct search criteria has not been used – It should be cost less than 10 and not less than and equals to as stated in the SQL script.

**Sorting – Order By command**

The field has not been sorted correctly into descending order of place. It was sort in ascending order of place and this is wrong.

The SQL script did not allow the operation to be carry out correctly and the requirements in the analysis stage have not been met therefore the SQL script is not **fit for purpose**.

**Accuracy of output**

The cost field should not have been included in the output. The time field should have been included in the output. Therefore this is **inaccurate output**.

**Solution 2 - fit for purpose and accurate output**

**Implementation**

**SELECT Award.organise, Award.place,** **Award.time**

**FROM Award**

**WHERE cost< 10**

**ORDER BY Award.place ASC;**

**Testing**

**Predicted result**

The predicted result shows what the developer expects the query to return.

| **organiser** | **place** | **time** |
| --- | --- | --- |
| Mr Rae | Pizza and Move | 09:30 |
| Mrs Stokes | M&D’s | 09:30 |

**Actual Result**

The actual result of the query is then recorded. It can be compared to the expected result to make sure that the query has executed as expected.

| **organiser** | **place** | **time** |
| --- | --- | --- |
| Mr Rae | Pizza and Move | 09:30 |
| Mrs Stokes | M&D’s | 09:30 |

**Assessment on result** – actual output does match expected output for the following reasons:

**Fitness for purpose**

**Searching – Where command**

The correct search criteria has been used – It has been sated in the SQL script cost less than 10.

**Sorting – Order By command**

The field has been sorted correctly into descending order of place.

The SQL script did allow the operation to be carry out correctly and the requirements in the analysis stage have been met therefore the SQL script is **fit for purpose**.

**Accuracy of output**

The correct fields have been included in the output. The fields displayed from the Award table were organise, place, time**.** Therefore this is **accurate output**.

**Example 2– Insert (add a record) command**

A new player is required to be added. playerID is 16, playerName is Mathew Henry, height is 1.89, weight is 78, is age 22 and he is a free agent. His agent is Out Win.

**Query Design and Testing**

**Design**

Query Design: Insert the player new record for Mathew Henry, height is 1.89, weight is 78, is age 22, free agent true and agent is Out Win.

|  |  |
| --- | --- |
| Field(s) | playerID, playerName, height, weight, age, freeAgent, agentID |
| Table(s) | Player |
| Criteria | Insert Player playerID = 16, playerName = Mathew Henry, height = 1.89, weight = 78, age = 22, freeAgent= True, agentID = OW466 |

**Solution 1 – not fit for purpose and not accurate output**

**Implementation**

**INSERT INTO Player (playerID, playerName, agentID)**

**VALUES (16, “Mathew Henry”, PF822);**

**Testing**

**Predicted result**

The predicted result shows what the developer expects the query to return.

| **playerID** | **playerName** | **height** | **weight** | **age** | **freeAgent** | **agentID** |
| --- | --- | --- | --- | --- | --- | --- |
| 16 | Mathew Henry | 1.89 | 78 | 22 | √ | OW466 |

**Actual Result**

The actual result of the query is then recorded. It can be compared to the expected result to make sure that the query has executed as expected.

| **playerID** | **playerName** | **agentID** |
| --- | --- | --- |
| 16 | Mathew Henry | PF822 |

**Assessment on result – actual output does not match expected output for the following reasons:**

**Fitness for purpose**

**Adding a record – Insert command**

The correct record has been added using the SQL script however the correct Insert criteria has not been used. It should be also include the fields height, weight, age, freeAgent with the values 1.89,78, 22, True and not only playerID, playerName and agentID as stated in the SQL script.

The SQL script did not allow the operation to be carry out correctly and the requirements in the analysis stage have not been met therefore the SQL script is not **fit for purpose**.

**Accuracy of output**

All the fields where required to be added to the record and should have been displayed in the output. The missing values are 1.89, 78, 22, True for height, weight, age and freeAgent consecutively. Therefore this is **inaccurate output**.

**Solution 2 – fit for purpose and accurate output**

**Implementation**

**INSERT INTO Player (playerID, playerName,** **height, weight, age, freeAgent agentID)**

**VALUES (16, “Mathew Henry”, 1.89,78, 22, PF822);**

**Testing**

**Predicted result**

The predicted result shows what the developer expects the query to return.

| **playerID** | **playerName** | **height** | **weight** | **age** | **freeAgent** | **agentID** |
| --- | --- | --- | --- | --- | --- | --- |
| 16 | Mathew Henry | 1.89 | 78 | 22 | √ | OW466 |

**Actual Result**

The actual result of the query is then recorded. It can be compared to the expected result to make sure that the query has executed as expected.

| **playerID** | **playerName** | **height** | **weight** | **age** | **freeAgent** | **agentID** |
| --- | --- | --- | --- | --- | --- | --- |
| 16 | Mathew Henry | 1.89 | 78 | 22 | √ | OW466 |

**Assessment on result – actual output does match expected output for the following reasons:**

**Fitness for purpose**

**Adding a record – Insert command**

The correct Insert criteria has been used – the SQL script used has added the correct record.

The SQL script did allow the operation to be carry out correctly and the requirements in the analysis stage have been met therefore the SQL script is **fit for purpose**.

**Accuracy of output**

All the fields where required to be added to the record and have been displayed in the output. Therefore this is **accurate output**.

**Example 3 – Update (amend) command**

A player Danny Young with playerID 3 wishes to be known as Daniel Young and is only 26 and not 28 as recorded in the database. Amend the database accordingly.

**Query Design and Testing**

**Design**

Query Design: Amend the record with playerID 3 for Danny Young to Daniel Young aged 26.

|  |  |
| --- | --- |
| Field(s) | playerName |
| Table(s) | Player |
| Criteria | Update player Danny Young with playerID =3 to playerName=”Daniel Young”, age=26 |

**Solution 1 - not fit for purpose and inaccurate output**

**Implementation**

**UPDATE Player**

**SET playerName=”Daniel Young”, age = 26**

**WHERE name= “Danny Young;**

**Testing**

**Predicted result**

The predicted result shows what the developer expects the query to return.

| **playerID** | **playerName** | **height** | **weight** | **age** | **freeAgent** | **agentID** |
| --- | --- | --- | --- | --- | --- | --- |
| 3 | Daniel Young | 1.85 | 62 | 26 | √ | BF511 |

**Actual Result**

The actual result of the query is then recorded. It can be compared to the expected result to make sure that the query has executed as expected.

| **playerID** | **playerName** | **height** | **weight** | **age** | **freeAgent** | **agentID** |
| --- | --- | --- | --- | --- | --- | --- |
| 3 | Daniel Young | 1.85 | 62 | 26 | √ | BF511 |
| 17 | Daniel Young | 1.85 | 64 | 26 |  | JC322 |

**Assessment on result** – actual output does not match expected output for the following reasons:

**Fitness for purpose**

**Update – amend command**

The correct update criteria has not been used – It should identify the correct Danny Young to be updated as it has update two Danny Young’s in the database. The player ID should have been used to uniquely identify the player record requiring to be update, **“WHERE playerID=3”** should have been as stated in the SQL script.

The SQL script did not allow the operation to be carry out correctly and the requirements in the analysis stage have not been met therefore the SQL script is **not** **fit for purpose**.

**Accuracy of output**

The correct fields player name and age have been update in the output however another player’s record has been changed which should have not been updated. Therefore this is **inaccurate output**.

**Solution 2 - fit for purpose and accurate output**

**Query Design and Testing**

**Design**

Query Design: Amend the record for Danny Young to Daniel Young aged 26 where playerID is 3

|  |  |
| --- | --- |
| Field(s) | playerName |
| Table(s) | Player |
| Criteria | Update player with playerID =3 to playerName=”Daniel Young”, age=26 |

**Implementation**

**UPDATE Player**

**SET playerName=”Daniel Young”, age = 26**

**WHERE playerID=3**

**Testing**

**Predicted result**

The predicted result shows what the developer expects the query to return.

| **playerID** | **playerName** | **height** | **weight** | **age** | **freeAgent** | **agentID** |
| --- | --- | --- | --- | --- | --- | --- |
| 3 | Daniel Young | 1.85 | 62 | 26 | √ | BF511 |

**Actual Result**

The predicted result shows what the developer expects the query to return.

| **playerID** | **playerName** | **height** | **weight** | **age** | **freeAgent** | **agentID** |
| --- | --- | --- | --- | --- | --- | --- |
| 3 | Daniel Young | 1.85 | 62 | 26 | √ | BF511 |

**Assessment on result** – actual output does not match expected output for the following reasons:

**Fitness for purpose**

**Update – amend command**

The correct update criteria has been used – It has identify the correct Danny Young to be updated as the player ID unique identify has been stated in the SQL script along with the player name, **“WHERE playerID=3 AND name= “Danny Young”**.

The SQL script did allow the operation to be carry out correctly and the requirements in the analysis stage have been met therefore the SQL script is **fit for purpose**.

**Accuracy of output**

The correct fields player name and age have been update in this record. Therefore this is **accurate output**.

**Example 4 – Delete command**

Leo Felicevich with playerID 9 has decide to retire from football his record has to be deleted.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **playerID *(primary Key)*** | **name** | **height**  | **weight** | **Nationality** | **age** | **freeAgent** | **agentID** |
| 1 | Aaron Walker | 1.93 | 76 | British | 23 |  | BF511 |
| 2 | Wallace Smith | 1.75 | 83 | British | 21 |  | JC322 |
| 3 | Danny Young | 1.85 | 62 | British | 28 |  | BF511 |
| 4 | Adam Young | 1.69 | 92 | British | 22 |  | JC322 |
| 5 | Alexis Vidal | 1.8 | 68 | Italian | 31 |  | FS447 |
| 6 | Leo Felicevich | 1.75 | 78 | Italian | 26 |  | BF511 |
| 7 | Nicholas Bravo | 1.78 | 80 | German | 25 |  | FS447 |
| 8 | Manuel Volland | 1.74 | 86 | German | 28 |  | PF822 |
| 9 | Leo Felicevich | 1.88 | 78 | Italian | 34 |  | FS447 |

**Query Design and Testing.**

**Design**

Query Design: Delete the record for Leo Felicevich with playerID.

|  |  |
| --- | --- |
| Table(s) | Agent |
| Criteria | Delete record playerID=9 |

**Solution 1 - not fit for purpose and not accurate output**

**Implementation**

**Delete From Agent**

**WHERE playerName = “**Leo Felicevich”

**Testing**

**Predicted result**

The predicted result shows what the developer expects the query to return.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **playerID *(primary Key)*** | **name** | **height**  | **weight** | **Nationality** | **age** | **freeAgent** | **agentID** |
| 1 | Aaron Walker | 1.93 | 76 | British | 23 |  | BF511 |
| 2 | Wallace Smith | 1.75 | 83 | British | 21 |  | JC322 |
| 3 | Danny Young | 1.85 | 62 | British | 28 |  | BF511 |
| 4 | Adam Young | 1.69 | 92 | British | 22 |  | JC322 |
| 5 | Alexis Vidal | 1.8 | 68 | Italian | 31 |  | FS447 |
| 6 | Leo Felicevich | 1.75 | 78 | Italian | 26 |  | BF511 |
| 7 | Nicholas Bravo | 1.78 | 80 | German | 25 |  | FS447 |
| 8 | Manuel Volland | 1.74 | 86 | German | 28 |  | PF822 |

**Actual Result**

The actual result of the query is then recorded. It can be compared to the expected result to make sure that the query has executed as expected.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **playerID *(primary Key)*** | **name** | **height**  | **weight** | **Nationality** | **age** | **freeAgent** | **agentID** |
| 1 | Aaron Walker | 1.93 | 76 | British | 23 |  | BF511 |
| 2 | Wallace Smith | 1.75 | 83 | British | 21 |  | JC322 |
| 3 | Danny Young | 1.85 | 62 | British | 28 |  | BF511 |
| 4 | Adam Young | 1.69 | 92 | British | 22 |  | JC322 |
| 5 | Alexis Vidal | 1.8 | 68 | Italian | 31 |  | FS447 |
| 7 | Nicholas Bravo | 1.78 | 80 | German | 25 |  | FS447 |
| 8 | Manuel Volland | 1.74 | 86 | German | 28 |  | PF822 |

**Assessment on result** – actual output does not match expected output for the following reasons:

**Fitness for purpose**

**Delete command**

The correct search criteria has not been used – The SQL script It should have deleted only one record Leo Felicevich with playerID 9 and not two records.

The SQL script did not allow the operation to be carry out correctly and the requirements in the analysis stage have not been met therefore the SQL script is not **fit for purpose**.

**Accuracy of output**

The player table has two records deleted and only one record playerID 9, Leo Felicevich should no longer be in the table. Therefore this is **inaccurate output**.

**Solution 2 - fit for purpose and accurate output**

**Implementation**

**Delete From Agent**

**WHERE playerID =** 9

**Testing**

**Predicted result**

The predicted result shows what the developer expects the query to return.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **playerID *(primary Key)*** | **name** | **height**  | **weight** | **Nationality** | **age** | **freeAgent** | **agentID** |
| 1 | Aaron Walker | 1.93 | 76 | British | 23 |  | BF511 |
| 2 | Wallace Smith | 1.75 | 83 | British | 21 |  | JC322 |
| 3 | Danny Young | 1.85 | 62 | British | 28 |  | BF511 |
| 4 | Adam Young | 1.69 | 92 | British | 22 |  | JC322 |
| 5 | Alexis Vidal | 1.8 | 68 | Italian | 31 |  | FS447 |
| 6 | Leo Felicevich | 1.75 | 78 | Italian | 26 |  | BF511 |
| 7 | Nicholas Bravo | 1.78 | 80 | German | 25 |  | FS447 |
| 8 | Manuel Volland | 1.74 | 86 | German | 28 |  | PF822 |

**Actual Result**

The actual result of the query is then recorded. It can be compared to the expected result to make sure that the query has executed as expected.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **playerID *(primary Key)*** | **name** | **height**  | **weight** | **Nationality** | **age** | **freeAgent** | **agentID** |
| 1 | Aaron Walker | 1.93 | 76 | British | 23 |  | BF511 |
| 2 | Wallace Smith | 1.75 | 83 | British | 21 |  | JC322 |
| 3 | Danny Young | 1.85 | 62 | British | 28 |  | BF511 |
| 4 | Adam Young | 1.69 | 92 | British | 22 |  | JC322 |
| 5 | Alexis Vidal | 1.8 | 68 | Italian | 31 |  | FS447 |
| 6 | Leo Felicevich | 1.75 | 78 | Italian | 26 |  | BF511 |
| 7 | Nicholas Bravo | 1.78 | 80 | German | 25 |  | FS447 |
| 8 | Manuel Volland | 1.74 | 86 | German | 28 |  | PF822 |

**Assessment on result** – actual output does match expected output for the following reasons:

**Fitness for purpose**

**Delete command**

The correct search criteria has been used – It has been sated in the SQL script playerID 9 to be deleted.

The SQL script did allow the operation to be carry out correctly and the requirements in the analysis stage have been met therefore the SQL script is **fit for purpose**.

**Accuracy of output**

The record has been deleted - playerID 9 has been be deleted Therefore this is **accurate output**.

**Example 5 Equi-Join between tables**

**Now** display all records from the Player table and join it with the Agent table, joining the records where the attribute name agentID of the Agent table matches the attribute name agentID on the Player table:

**Solution 1 – not fit for purpose and not accurate output**

**Implementation**

SELECT \*

FROM Player

INNER JOIN Agent ON Agent.agentID = Player.agentID;

We must specify field names with dot notation such as Agent.agentID means the agentID field of the Agent table.



**Assessment on result** – actual output does not match expected output for the following reasons:

**Fitness for purpose**

**Equi-Join between tables**

Select \* has been used which means select all. Consequently agentID has been displayed from the player table and again from the agent table due to agentID being the foreign key. The solution would be to select each of the fields from the tables and not select the foreign key.

The SQL script did not allow the operation to be carry out correctly and the requirements in the analysis stage have not been met therefore the SQL script is **not** **fit for purpose**.

**Accuracy of output**

The playerID field has been displayed twice, from the player table and from the agent table and should only be displayed once. Therefore this is **inaccurate output**.

**Solution 2 –fit for purpose and accurate output**

**Implementation**

SELECT playerID, name, height, weight, nationality, age, freeAgent,agentID.agent, typeOfAgent, name, address, phoneNumber

FROM Player

INNER JOIN Agent ON Agent.agentID = Player.agentID;

**Fitness for purpose**

**Equi-Join between tables**

The Equi-join has correctly joined and extracted the fields from the two tables.

The SQL script did allow the operation to be carry out correctly and the requirements in the analysis stage have not been met therefore the SQL script is **fit for purpose**.

**Accuracy of output**

The correct fields have been displayed. Therefore this is **accurate output**.

**Finding errors**

In some solution to the examples, the predicted result and actual result do not match.

If the database is not fit for purpose and/or inaccurate output, it will be necessary to revisit previous phases of the development process – iterative process. In this case go back to previous stage such as the design and implementation phases to try to identify the error and correct it. Then run the corrected SQL script and check if now **actual matches expected results**.