University of London International Programmes

Computing and Information Systems/Creative Computing

CO2209 Database Systems

Coursework assignment 1 2017–18

Your coursework assignment should be submitted as a single PDF file, using the following file-naming conventions:

YourName_SRN_COxxxxcw#.pdf (e.g. MarkZuckerberg_920000000_CO2209cw1.pdf)

- ***YourName** is your full name as it appears in your student record (check your student portal);*
- ***SRN** is your Student Reference Number, for example 920000000;*
- ***COXXXX** is the course number, for example CO2209; and*
- ***cw#** is either cw1 (coursework 1) or cw2 (coursework 2).*

If you're not sure how to do this, come to the course discussion board and ask.

It should take between 20 and 40 hours to complete, depending on how much you already know about the topics. There are some easy parts, and some which are more challenging.

Each part of the coursework assignment has been chosen to help you understand some key issue in the subject of databases. It should be undertaken with the subject guide, Volume I, at hand. There are four Appendices at the back to supplement the information in the subject guide.

The best way to approach this coursework assignment is to look over the whole thing first, and get an idea of what you will want to concentrate on as you read the subject guide and other materials such as your textbook. If you have never encountered relational database ideas before, the terms will be unfamiliar, and it will take some time for them to become part of your everyday working inventory of ideas – learning these definitions by heart might be a good strategy to start with, because this will help you gain a deeper conceptual understanding of them as you do the coursework assignment.

Background

If you look closely at almost any online enterprise, public or private, you will find a database system behind the public face. Therefore, the more knowledge and experience you have with database systems, the better your chances are of finding a good job. The aim of this coursework assignment is to help you gain some of that knowledge and experience.

This coursework assignment (and coursework assignment 2) will introduce you to the basic concepts of the most common data model (the relational model), around which most databases are constructed. It will give you some practical experience in designing, implementing and using a database management system, and it will acquaint you with some of the issues currently of concern in the database world. To put it another way, if you do this coursework assignment conscientiously, then you should to be able to talk confidently about databases in a job interview, as well as in the examination.

This course can only introduce you to the basics. To start to become a professional in the field, you will need to gain experience with a real database, which will be much more complex in every way than the simple examples we will look at here. You will also need to learn how to stay up to date in this area, and how to keep educating yourself about developments in it long after you have finished this course. This coursework assignment aims to help you understand how to do this.

Why it is important for you to complete this coursework assignment
The coursework assignment has two practical aims: first, to provide you with a 'rehearsal' for the examination. Second, more importantly, the coursework assignment reaches areas that are not covered in the subject guide. If you do the coursework assignment, you will be up-to-date with respect to recent developments in the database field. Several of the questions have been written with an eye to the questions you might be asked during a job interview, and they are designed to allow you to give a competent answer to them.

Suggested sources: this coursework assignment has been designed around the subject guide, but in addition to this, and the recommended reading, you will want to consult the wealth of information available via the internet. **Appendix I**, we have provided some links relating to MySQL, but you should not confine yourself to them. Becoming familiar with reliable sources of information about current database systems, and using them to keep your knowledge up to date, is part of becoming a database professional.

There are also discussions about dealing with the practical issues involved in database design in the other appendices to this coursework assignment. Be sure to look at the Appendices to become familiar with what they contain. You are strongly advised to consult them as you do these coursework assignments, as well as when you revise for the examination.

A note on Wikipedia: Wikipedia is the place where most people begin their online searches. This coursework assignment will frequently direct you to Wikipedia articles. Wikipedia articles often provide a good introduction to a topic (although occasionally they are over-technical and not useful for beginners). However, Wikipedia is not an unquestionable authority. (No authority is unquestionable, of course.)

You **cannot** treat Wikipedia the same way that you treat ordinary references because the articles can be written by anyone, can have many authors (who are usually anonymous) and the contents can be changed at any time. Remember that the author(s) of these articles may only have a partial knowledge of the subject, may be overly partisan and/or have a material interest in convincing readers of a particular point of view.

For example, the popular DBMS MySQL can be used with several different software systems (or 'engines') for physical layout in secondary storage and indexing; one is called MyISAM and another is called InnoDB. If you read the Wikipedia article [accessed during summer 2016] comparing the two, [https://en.wikipedia.org/wiki/Comparison_of_MySQL_database_engines] you will see that it clearly has been written by someone who is an InnoDB enthusiast. It would be very dangerous to make a decision about the relative merits of these two alternative database engines based solely on that article, which is highly biased.

Another example is the Wikipedia article on Data Integrity [https://en.wikipedia.org/wiki/Data_integrity]. The author of this article has only a middling grasp of English. The contents are useful, but you would definitely not want to quote from this article due to its poor grammar.

Therefore, you should never rely **only** on Wikipedia as a source of information. When you use Wikipedia, you should check the warnings at the top of the page to see if 'the neutrality of this article is disputed' or if there are any other listed concerns about it. It is good practice to consult the 'Talk' page for the article and see if there are disputes among the contributors. Use Wikipedia, if necessary, as a starting place and as a source of links to follow, but do not quote Wikipedia articles as authoritative sources. In fact, in a formal report that requires a list of references, you are better off not listing Wikipedia at all. If you're not sure about something regarding databases that you have found in Wikipedia, or anywhere else online, please ask about it via the course discussion board.

Coursework assignment and the examination
As mentioned earlier, the coursework assignments are also designed to help you prepare for the examination, so you will doubly gain from making the most of them. You will notice that both coursework assignments include some very simple initial assignments, which consist essentially of having you pay close, systematic attention to the subject guides and making notes about the most important parts. Copy these notes onto separate sheets of paper (and perhaps use them to make 'flash cards'), and you will then have a ready-made set of revision materials. However, please note, your revision should start in November or December at the latest, not in April.

Coursework assignment and real life
This coursework assignment has also been prepared to give you a solid footing should you face questions on practical database subjects during a job interview. You will be able to honestly say that you have had experience of implementing a database, albeit a 'toy' one, of making relational designs, and, upon completion of the coursework assignment 2, of using a genuine database. In addition, some of the questions help you to engage you with current developments in this fast-moving field.

IMPORTANT NOTE:
It is important that your submitted assignment is your own individual work and, for the most part, written in your own words. You must provide appropriate in-text citation for both paraphrase and quotation, with a detailed reference section at the end of your assignment. It is important that your submitted assignment is your own individual work and, for the most part, written in your own words. Copying, plagiarism and unaccredited and/or wholesale reproduction of material from books online sources, etc. is unacceptable, and will be penalised (see How to avoid plagiarism).

Coursework assignment 1
The coursework assignment this year has been designed to address some of the typical errors and confusion with respect to relational databases shown in previous years' coursework assignments. Sometimes even mature students who have worked with databases for many years have revealed gaps in their knowledge, especially about certain concepts of basic design.

Do not hesitate to use the course discussion board if you have questions – it's one of the great advantages of an online course that it's actually easier to get help than in many traditional courses, if you will take advantage of the facilities offered.

This coursework assignment has eight basic tasks: **(A)** downloading and setting up the software for managing a database, **(B)** becoming familiar with the manual, **(C)** drawing up a visual model of the properties and inter-relationships of the related groups of data we will incorporate in a database, **(D)** showing certain key facts about how the data within these groups is related, **(E)** drawing up an abstract 'schema' of our database using the previous two tools, then **(F)** implementing a 'toy' database, **(G)** seeing what should happen if we violate some of the rules governing what data can, and cannot, be in the database, **(H)** finally running some queries on the database itself.

If you complete these tasks, you should receive a good introduction to the fundamental ideas of database. Do not hesitate to use the course discussion board if you have questions. One of the great advantages of an online course, if you take advantage of the facilities offered, is that it is actually easier to get help than in many traditional courses. **However, please be careful not to post anything that will form part of your submission.** In addition to the help from the VLE tutor, you will find that many of your fellow students are experienced database users already and will be more than willing to discuss issues.

A. Downloading a DBMS
You can – and should – get started on this part of the coursework assignment immediately, even if you know nothing about databases, just in case you have some problems setting this system up.

Download and install the MySQL database package on your own computer.

You can download MySQL from here: http://dev.mysql.com/downloads/mysql
(Get the 5.7.9 version for whatever Operating System you have. If you already have an earlier version of MySQL, it is *not* necessary to download a later version.) You will have to create an Oracle Account if you do not already have one, but this only takes a few minutes.

Note: Depending on how fast your connection to the internet is, this download may take a long time. It's a large (320 MB) package.

Note: Most people have no trouble downloading and installing the MySQL package. However, problems can occur. If they do, and you cannot solve them quickly by yourself, you **must** connect with the course discussion board for this course straightaway and get help.

Further helpful links to sources of information and help with MySQL can be found in **Appendix I.**

If for some reason you cannot get a working version of MySQL installed on your computer, download and install **MariaDB** from http://mariadb.org/en/. This is a 'drop-in' equivalent of MySQL, and it was started by people concerned about MySQL's public status, after the

Oracle Corporation bought it. They fear that it may eventually be left to wither on the vine. You can read about MariaDB here: https://en.wikipedia.org/wiki/MariaDB

What to submit: write a short report describing your own experience of databases and/or database system software. If you have had no experience, describe any issues/problems you had downloading and installing MySQL. If you have had no experience of databases, and also had no problems downloading and installing MySQL (or you already had it on your computer), you should write a short report (no more than one page) summarising the contents of the MySQL Manual. You can do this by listing (but not copying) the most important items in the Table of Contents.

[2 hours. 5 marks]

B. Learning about MySQL

Subject guide reference: pages 34-55 of *Database Systems*, *Volume 1.*

Look at the tutorial on using MySQL, which you can find at: **https://dev.mysql.com/doc/refman/5.7/en/tutorial.html** (Note: the tutorial refers to tables, rows and columns, which in this context mean relations, tuples and attributes.)

Read through this part of the MySQL manual and answer the following questions.

B(1) What is the URL of the section entitled **‘B.5.2 Common Errors When Using MySQL Programs’**?

B(2) The tutorial tells us that after a query that you enter is executed, the MySQL server ‘*shows how many rows were returned and how long the query took to execute…*’ Is the server performance time thus presented precise, and if not, why not?

B(3) What is the query that would have MySQL tell you your user name, its version number, and the current date?

B(4)(a) What is the meaning of each of the following MySQL prompts?

Prompt	**Meaning**
mysql>	
->	
‘>	
‘>	
`>	
/*>	

B(4)(b) Suppose you enter a query, but nothing happens. You see that the MySQL prompt now looks like this: -> What have you forgotten to do? What must you add for the query to complete?

B(5) **[Section 3.3]** What query will tell us what databases currently exist?

B(6) Suppose we know that a database called ‘SchoolData’ currently exists. How do we access it so that we can modify it or query it?

B(7) What is the command to create a database called ‘SchoolData’ if one does not exist?

B(8) Suppose you create a database called ‘SCHOOLDATA’, and later try to access it by typing USE Schooldata. Does it make a difference **(a)** under Windows, and **(b)** under UNIX?

B(9) Once you are using a particular database, how can you see what tables it consists of?

B(10) If you know an existing table’s name – let’s say it’s called STUDENTDATA – how can you get MySQL to show you its structure (the name of each of its columns, their datatype, whether this column can have null values, whether it’s part of the Primary Key, its default value...)?

B(11) Suppose you want to create a table called STUDENTDATA. You want it to have columns called SNum, FName, LName, BirthDate, and Sex. You want SNum to hold integers, FName and Lname to hold strings of characters, each of which can be up to 36 characters long, Birthdate to be a date, and Sex to be a single character, either ‘M’ or ‘F’. What command will let you create such a table?

B(12) Suppose you now wish to put data into the table you have created, and you already have the data in a .txt file called Studentdata.txt – one row of data per line.

(a) What command can copy the data from the .txt file into your table?
(b) How should each data item on a line in the .txt file be separated from the next one?
(c) If you do not know a certain value, and want to insert the value NULL in its place, how should this be represented in the .txt file?

B(13) Suppose you want to add a row of data directly to Studentdata… how would you add: 314159 Kapoor Sahib 2002-02-14 M?

B(14) Suppose, after creating the relation Studentdata, you decided to add another column (attribute), called ‘MI’ of type Char[1] between the columns ‘FName’ and ‘LName’. What command would let you do this?
(See: https://dev.mysql.com/doc/refman/5.7/en/alter-table.html).

B(15) Suppose you wanted to change Sahib’s BirthDate to ‘2002-02-15’. What statement would allow you to do this?

B(16) Suppose you wanted to see all of the columns of all of the rows of STUDENTDATA. What command would allow you to do this?

B(17) Suppose you wanted to see only the first names of all the students. What command would allow you to do this?

B(18) Suppose you wanted to see all the first names of all the female students only. What command would allow you to do this?

B(19) Suppose you wanted to see all the Firstnames of all the female students who were born before January 1, 1999. What command would allow you to do this?

B(20) Suppose you wanted to see the first names of all the female students who were born before January 1, 1999, sorted on their birthdates with the youngest first. What command would allow you to do this?

B. What to submit: Answers to questions **B(1)–B(20).** ***Be sure to number your answers.***

[4 hours. 10 marks]

C. Creating a visual model of related data

In this section of the coursework assignment we will take an imaginary business which wants to create a database, and draw up a visual model of how its data is related, called an 'Entity-Relationship Diagram'.

> **Subject guide reference:** pages 103-123 of the subject guide, *Database Systems*, *Volume 1*, and the section on 'Types and Instances' in **Appendix II** of this coursework assignment. Note that in ordinary language, as used in this coursework assignment, the distinction between 'type' and 'instance of a type' is not always reflected in language used. You must understand the distinction, where it is important, from context.

The background

A manufacturer of SatNav devices gets its **electronic parts** from a number of different **suppliers**, and assembles various **models** of SatNav device from them. These SatNav devices are then sold to high street **distributors**. The manufacturer will hold information on **suppliers** even if they are not currently supplying any **electronic parts**.

In order not to become dependent on any one supplier, the manufacturer has arranged it so that any given electronic part-type is supplied by at least two separate suppliers.

A SatNav **model-type,** is identified by a (unique) 'Model Number' (MNUM). Each model is made up of several different types of **electronic part-types,** each of which is identified by a 'Part Code' (PCODE). Each electronic part-type can be used in the assembly of several different models of SatNav models. A SatNav **model** can exist which is not being distributed by anyone at the moment. A SatNav model *must* be made of one or more **electronic parts**, but an electronic part-type could exist which is not currently being used in making any of our SatNav models.

A given SatNav device **distributor**, identified by a (unique) 'Distributor ID' (D-ID) can receive and re-sell many different models of SatNav device. No SatNav device distributor has a monopoly on re-selling any given SatNav device type. A distributor must be distributing at least one model of SatNav device in order to be considered a distributor. We do not allow a distributor to distribute a particular model until we are sure that they can install and service it.

Each **electronic part-type** has a name, not necessary unique, and a weight. Each **supplier** is identified by a (unique) Supplier Number (SNO), and has a name and country. Each **model** of a SatNav device has a name, and a wholesale base price. Each **distributor** of the devices has a name.

A **supplier** supplies us with a fixed number of electronic part types per month – which we will identify as a QTY (for 'quantity'). A **distributor** of the manufacturer's SatNav devices places **orders** for them: an **order** is identified by a unique Order Number (ORDNO), which has a date, and which consists of one or more Order Lines. An Order Line consists of a Model Number and a quantity.

Draw an Entity/Relationship Diagram showing the relationships among **Suppliers**, **Parts**, Models and **Distributors**, as recorded in your database. Show Entity Types, their Attributes, and their Relationships with each other. Be sure to indicate Cardinality and Participation Constraints. Your diagram should have a *key* so that anyone looking at it will know how you have indicated Cardinality and Participation.

Note that there is not a single unique 'correct' way of drawing up such a diagram. In particular, you may choose to show that Distributors' orders Models as a 'pure' relationship between Model and Distributor, or you may wish to create 'intermediate' Entity-Types called Order and **Order-Line**. As long as we can use your E/R diagram to design our Relations later on, it will serve its purpose.

Note that Distributors have two distinct relationships to Models: they are *authorised to distribute* certain Models; and they *place orders* for models.

C. What to submit: An E/R Diagram conforming to the requirements above.

[4 hours. 10 marks]

D. Determining Functional Dependencies

Subject guide reference: Pages 129-131 of subject guide, *Database Systems*, *Volume 1*.

The ‘dependencies’ in our data tell us what data will appear in the same table. If there are direct ‘dependencies’ between data items [See Appendix II], these data items will appear in the same table.

We need to understand the ‘Functional Dependencies’ in our data in order to design our tables.

For this section of the coursework assignment you will need to draw up a set of **Functional Dependency Diagrams**, showing the ‘functional dependencies’ among the attributes of the entity types (and relationships) you have identified in the previous diagram. Your diagrams should be non-trivial and left-irreducible in the sense described in the subject guide.

Draw your FD diagrams using this model: Assume a Student Registration Number uniquely identifies a student, and that a student can have only one Surname. Then we would show the functional dependency from Student Registration Number to Surname this way.

Student Registration Number $\rightarrow$ Surname

We would read this as, ‘For a given Student Registration Number, there can be only one Surname.’ (Note what this **does not** say: that for a given Surname, there can be only one Student. Many people get this wrong – do not be one of them!)

Assume that a Student can take a Course, which is identified by a Course-Code, only once, and gets only one grade for it. Then we would show the Functional Dependency between Student Registration Number, Course-Code, and Grade, this way.

Your submission should look like this!

Student Registration Number + Course-Code $\rightarrow$ Grade

We would read this as, ‘For a given combination of Student Registration Number and Course-Code, there can be only one Grade.’

So…suppose a student whose Student Registration Number was 218281828 has taken three courses, CS101, CS104, and CS110, and has received 72, 89, and 72 in these courses, respectively, and a student with Student Registration Number 314159265 who has taken CS101 has received 68 in it, as has another Student, 11235813. They have had their information recorded like this:

218281828	CS101	72
218281828	CS104	89
218281828	CS110	72
314159265	CS101	68
011235813	CS101	68

But if we saw the following entry:

218281828	CS104	76

we would know, according to the functional dependency
Student Registration Number + Course-Code $\rightarrow$ Grade
that something was wrong, because for a particular combination of Student Registration

Number and Course-Code there can be only 1 grade, not 2.

Note that any single Student Registration number can be associated with several Course-Codes, and *vice-versa*. And the same for Registration Numbers and Grades; and for Grades and Course-Codes.

Note also that this constraint is not something ‘inherent’ in the data. If the institution giving these courses allowed re-sits, we would no longer have a Functional Dependency: a given Student and a given Course could be associated with more than one grade. But in this case, we assume that the policy is: one grade only.

Do **not** show trivial Functional Dependencies, such as Surname → Surname, or ones which can be reduced to simpler forms, such as:

Surname + Student Registration Number + Course-Code → Grade

(In the case above, we can eliminate ‘Surname’.)

Assume that only system-given codes and numbers, like SNO, MNUM, PCODE are unique – do **not** assume that natural names are unique, even if they happen to be in this particular database instance.

D. What to submit: A set of (non-trivial, left-irreducible) Functional Dependency Diagrams, in the same format as the example given, for the data you described in your E/R diagram in **Part C.**

Give your data items the titles indicated in their descriptions above: SNO, PCODE, D-ID, MNUM, DATE, QTY, NAME, ORDNO, COUNTRY, WEIGHT, PRICE.

[4 hours. 10 marks]

E. Creating a Relational Schema

Subject guide reference: Pages 119-126 of subject guide, *Database Systems*, *Volume 1*.

To complete the remainder of the coursework assignment, you will need to know how a relational database is structured, and have a basic familiarity with the following words, which apply to relational databases in general: **Table** (or **Relation**), **Column** (or **Attribute**), **Primary Key**, **Candidate Key, Foreign Key**, **Domain, Data Type**, and **Constraint.**

Prepare a **Relational Schema** for the data you have described in the E/R Diagram and Functional Dependency Diagrams (Parts **C** and **D**.). A Relational Schema is just a formal description of the relations that will make up our database. Identify, for each relation, the Primary Key you have chosen, and any Alternate Keys (assuming there is more than one Candidate Key); also, for relations showing how Entity-types are related to each other, identify any Foreign Keys and show the Base ['Master'] Relation of which they are the Primary Key. You do **not** have to indicate the datatype (or 'domain') of each attribute.

Note: if you understand normalization, you can create a normalized set of tables. However, there is no penalty if your tables are not fully normalized (in this coursework assignment).

Entity-Type ('Master') Relations
You will need a table for each Entity-type which records information about that type only (that is, it does not record any information about its relationships with other entity-types.) In some circumstances, you might need more than one such table – for instance if each instance of an Entity-type like Supplier had multiple copies of some attribute, such as several telephone numbers – but that will not be the case here. (If it were the case, you would have to have a 'supplementary' second relation, with two attributes, consisting of the supplier number, and a telephone number – the two attributes taken together would make up the Primary Key.)

Call the table that has information only about Suppliers (not about what they supply, just their names and locations) SUPPLIER-MASTER. Call the table with information only about Part-types (their Part numbers, names and weights), PART-MASTER. Call the table with information only about Models (their Model numbers, names and wholesale prices), MODEL-MASTER. Call the table with names only about our Distributors (we're only recording their IDs and names), DISTRIBUTOR-MASTER.

Relationship Relations
You'll also need tables showing how Entity-Types are related to each other. These will often contain attributes about that relationship.

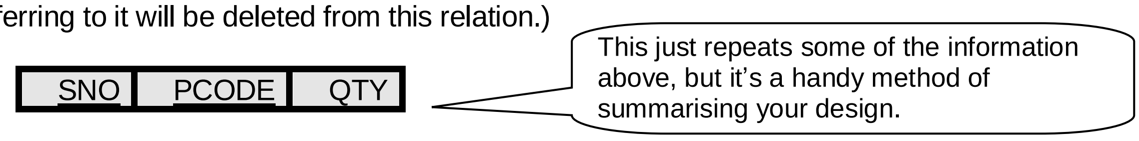
You will have a relation showing which Supplier supplies us with which Part-type, and how many per month they supply. Call this relation SUPPLIER-PART. You will have a relation showing which Part-types are used in which Models; call it PART-MODEL.

You will also have a relation showing which Distributors are authorised to distribute which of our models – this information is independent of whether or not a given distributor has actually ordered any of those models. Call this relation MODEL-DISTRIBUTOR (We do not allow a Distributor to distribute a Model until we are sure they can install and service the model.) We also need a relation to show which Distributors have actually ordered which Models, on what dates, and what their Order Numbers for each order were. Order Numbers are unique: no two distinct orders, even from the same Distributor, will have the same Order-Number. (An Order has a unique date, and one or more Model Numbers and quantities of that model being ordered). You can show this information using just one relation, in which case call it DISTRIBUTOR-ORDERS. Or you can show it using two relations, one showing a unique Order-Number, its date, and which Distributor is associated with that number, and the other showing the Order-Number, the Model ordered, and its quantity. If you do it the second way, call the second relation ORDERS. If this is unclear, look now at the next section, **F**. For this coursework assignment, either method is correct.

Your schema will be correct if we can use it to find all of the information that we want to record, without erroneous or missing data, or erroneous or missing associations among the data.

Here is a schema for the relation which shows which suppliers supply which parts and in which quantity (per month). Use this as a model for your other relations.

SUPPLIER-PART
Attributes: SNO, PCODE, QTY
Primary Key: SNO +PCODE
Alternate Keys: None
Foreign Keys: SNO from **SUPPLIER-MASTER**; PCODE from **PART-MASTER**
SNO: ON DELETE CASCADE ; PCODE: ON DELETE CASCADE
Meaning: The Supplier identified by SNO supplies us on a monthly basis with the Part Type identified by PCODE, in the quantity indicated by QTY. (The 'ON DELETE CASCADE' statement means that if a supplier, or a part type, is deleted from its master relation, tuples referring to it will be deleted from this relation.)



This just repeats some of the information above, but it's a handy method of summarising your design.

Note: indicate which attributes make up the Primary Key in your actual diagram by underlining them (or putting an asterisk by them – both methods are used in practice, but choose one!)

E. What to submit: A set of descriptions of the relations that will capture the data shown in your E/R diagram, modeled on the example above. Note: each Entity-Type will have a 'master relation', with its attributes making up the attributes (columns) of the relation; and there will also be a relation for each relationship among Entity-Types.

Give the attributes (columns) of your relations the titles indicated in the descriptions here: SNO, PCODE, D-ID, MNUM, DATE, QTY, NAME, ORDNO, COUNTRY, WEIGHT, PRICE.

[4 hours. 10 marks]

F. Implementing our database

Implement your schema in MySQL and populate it with the data presented here.

Subject guide reference: Pages 61, and 71-102 – especially page 78 – of subject guide, *Database Systems*, *Volume 1*; and the MySQL Manual: https://dev.mysql.com/doc/refman/5.7/en/create-table-foreign-keys.html – do not worry about the references to 'indexes'. Just see how to implement the 'foreign key constraints' via the ON DELETE CASCADE statement.

The purpose of this exercise is to provide some experience in setting up and querying one. Everything has been made as simple as possible – real databases are a bit more complicated! They are both much larger, physically, with almost all of their data being held on slow secondary storage when they are being accessed, and also much more complicated in terms of their structure – with more and larger relations, and more complicated relationships being captured by those relations. There are also 'physical' considerations, most importantly indexing, which we will ignore here.

IMPORTANT: Note that SQL can automatically record your commands to it, and the results of them, if you are working from the command line (which I strongly suggest that you do, to start with).

If you give SQL the command
TEE <path-and-filename>; it will output your commands and their results to a file, as in the following example:

SQL> **TEE** D: OutputLog.txt ; – whatever shows on the screen is also copied to the file OutputLog.txt which I have placed on my D: disc in this example, but which can be located anywhere you like.
SQL> **NOTEE ;** turns it off.

You can do a final listing of a whole table by using the *SELECT * FROM <tablename>* command. (See **B(16)**.)

Supplier Numbers, Model Numbers, Distributor Codes, and Part Codes are all of type CHAR(5). Look at the ORDNO (Order Number) and work out for yourself what type it should be. Quantities and Weights are of type INT. Dates are of type DATE. NAMEs, and COUNTRIES are of type VARCHAR(24). Prices are of type DECIMAL (10,2).

Implement your database in MySQL. When you have created its structure (the relation and its attributes), you can add the following data using either the 'INSERT' statement, or you can LOAD the data from a text file, as described in **Part B** of this coursework assignment.

We currently have six Suppliers: Bindar Pte, whose Supplier Number (SNO) is S0001 -- located in India; Thaksin Products, whose Supplier Number is S0002, also located in India; San-Lin Company, whose Supplier Number is S0003, located in Singapore; KaiShek Enterprises, S0004, located in Taiwan; Obedian Ltd, S0005, located in Malta; and Vitalsky Corporation, S0006, located in the Russian Federation.

We want to record information – Part number, name and weight – on 12 different Part types:

Part P0001 is a magnetic microswitch and weighs five grams.
Part P0002 is a power supply, and weighs 200 grams.
Part P0003 is a microwave receiver, and weighs 100 grams.

Part P0004 is a battery fixture, and weighs 75 grams.
Part P0005 is a control panel, and weighs 30 grams.
Part P0006 is a power supply, and weighs 225 grams.
Part P0007 is a capacitive tuner and weighs 50 grams.
Part P0008 is a buffer assembly, and weighs 20 grams.
Part P0009 is a power supply, and weighs 250 grams.
Part P0010 is a transformer and weighs 75 grams.
Part P0011 is a qubit resolver but we do not know its weight yet. [See Section 3.3.4.6 of the subject guide, 'Working with NULL Values', to see what to do here.]
Part P0012 is a heat sink and weighs 75 grams.

Model M0001 of our SatNav devices is called 'The Palinurus'; its wholesale price is $59.95. M0002 is called 'The Tenzing Norgay', and has a wholesale price of $75.00. M0003 is 'The Sacagawea' and has a wholesale price of $100.00.

We have three distributors. Distributor D0001 is named 'Quality Auto Accessories'. D0002 is 'Car Warehouse'. D0003 is 'Paul's Parts'.

Supplier S0001 supplies us with Electronic Parts P0001, P0003, P0005, and P0006, (5 each per month); Supplier S0002 supplies us with Electronic parts P0002 (4 per month), P0003 (2 per month), and P0004 (5 per month), and Supplier S0003 supplies us with Electronic parts P0001 and P0004 (4 and 6 per month respectively). Supplier S0004 is not currently supplying us with anything. S0005 supplies us with P0005 and P0006 (4 per month of each). S0006 supplies us with 1 each of P0006, P0007 and P0001 each month.

We use Electronic parts P0001 and P0002 to make SatNav device Model M0001; Model M0002 is made from P0002 and P0003 and P0004; and Model M0003 is made from P0001, P0005, and P0006. Model M0004, an experimental model, is made from Parts P0006, P0007, P0008, and P0011.

Distributor D0001 is authorized to distribute all of our models which are currently on the market (M0001 through M0003); D0002 can distribute M0001 and M0002 only (at the moment), and D0003 can distribute models M0001 and M0003.

Distributor D0001 ordered six Model M0001s on 2nd January 2017, and in the same order he ordered ten Models M0003. The OrderNumber for this order was 20170101. He ordered five more M0001s on the 9th January. The OrderNumber for this order was 20170102. Distributor D0003 ordered five each of Models M0001 and M0003 on the 9th January of this year – the OrderNumber for this order was 20170103. D0002 ordered five M0001s and two M0002s on 22nd December, 2016. The OrderNumber for this order was 20161232.

What to submit: (1) a copy of the SQL statements you used to create your database. (2) a listing of the tables you create. Use the TEE and NOTEE commands.

[4 hours. 10 marks]

G. Trying to modify the database while violating constraints

'Constraints' are restrictions on the data we enter into a database in order to try to prevent it from recording erroneous/impossible data. We call this trying to maintain the 'integrity' of the database, and three kinds of integrity are usually mentioned. Be sure you know how the phrases 'Entity Integrity', 'Attribute Integrity', and 'Referential Integrity' are used. (We normally use the CHECK statement to enforce 'Attribute Integrity', but MySQL does not implement it. If you are a sophisticated user who is already familiar with MySQL and you want to have CHECK functionality in your database, this link shows you ways to do it: https://mysqlserverteam.com/new-and-old-ways-to-emulate-check-constraints-domain/. However, it is not required in this coursework assignment.)

Note: if you are not sure of the answer to any of these questions, try them on your database and see what results you get.

What should happen, in the ideal case, if:

(1) You attempted to add a new supplier to the master supplier relation, but you put the value 'NULL' into the Supplier Number field?

(2) You attempted to add a tuple showing that Supplier S0007 supplies 5 of Part P0002 to us each month?

(3) You attempted to add a tuple showing that Distributor D0002 ordered 5 of model M0003?

(4) You redefined the key of the master supplier relation to be a composite key consisting of both Supplier Number **and** Supplier Name, and then added a new supplier with Supplier Number S0001, and Supplier Name 'SuperDyne, Ltd'? How many suppliers do we now have with the Supplier Number S0001? [If you try this, then after seeing the result, remember to delete the tuple you just added.]

(5) You deleted the tuple for Supplier S0001 in the master supplier relation? (Note: If nothing happened, besides the tuple being deleted, does this leave the database in an anomalous state?) [If you try this, then after seeing the result, remember to insert the tuple again.]

What to submit: Answers to questions (1) to (5). Number your answers.

[2 hours. 5 marks]

H. Querying the database

Show the SQL needed to answer the following queries, and also show the results you get when you execute each query.

Your answer should include the question number, the natural language version of the query, the SQL, and the output which results. Again, use the **TEE** command to record your work. Please do **not** use screen shots, as they are often hard to read. Also: you will be using this coursework assignment to revise for the examination, and this format will make it easier for you to revise the SQL.

EXAMPLE – how your submissions should look

Assume you have put the order information in a table called 'ORDERS', with the data of the order in a column called 'DATE'.

Query in natural language

Query number

0. What is the distributor number of the distributor who placed the earliest order?

```
SELECT DISTINCT  D-ID
FROM ORDERS
WHERE DATE  =
      (SELECT MIN(DATE)
      FROM ORDERS);
```

SQL –Don't put your whole query on one line –use several lines, and indent as I have done.

```
D-ID
 D0002
```

RESULTS (Your actual output will have some primitive text-graphics around it.)

(1) List the names and supplier numbers of all our suppliers.
(2) List the names and supplier numbers of all suppliers who are currently supplying us with parts.
(3) List the names and supplier numbers of all suppliers who are currently supplying us with Part P0002.
(4) List the part numbers of the parts Model M0003 is made out of.
(5) List Suppliers who supply any part Model M0003 is made out of.
(6) List the names of the distributors who distribute all of our currently-distributed models. (Note that we may have models which are not being distributed.)
(7) List the names of all suppliers who supply a part used in any model distributed by distributor D0001.
(8) List the Part Number of the part we receive the most of each month.
(9) List the Model Numbers of any of our models which made from Parts which no Supplier currently supplies, if any.
(10) List the supplier numbers of suppliers who supply us with at least two different types of part.
(11) List the Part Numbers of Part types which share the same name.
(12) List the Part Numbers of any Part for which we do not know the Weight.
(13) List the Part Number and Names of the Part(s) with the greatest Weight. (There may be a tie.)

(14) List the Total Value of all Orders placed by Distributors in January 2017. (This will be a single number.)

(15) List the number – count, i.e. how many – M0001's were ordered in January of 2017.

(16) List the Part Number of the Part which was most frequently ordered in January 2017.

(17) List the Distributor Name of any Distributor who is authorised to distribute a Model, but has never ordered it.

(18) List the Distributor Name and Total Value all Orders placed by each Distributor in January 2017, sorted from the lowest value to the highest. [There should be one row for each Distributor who placed orders in January 2017.]

(19) List the Supplier Numbers and Names of all Suppliers who do not currently supply us with any Parts.

(20) List the Supplier Numbers and Names of all Suppliers who supply us with Parts, but not with Part P0001.

What to submit:

(1) The natural language version of the query (just a copy of the original question.

(2) Your *SQL* **query** for each question, and

(3) the **results** of running that query.

[10 hours. 40 marks]

Both **(2)** and **(3)** can be easily recorded using the TEE command, with output to a text file which can then be incorporated into your PDF file submission.
Do not submit separate SQL files.

Be sure to number your answers, using the numbers shown above. Remember: *do NOT submit screenshots, or separate SQL files, for this question.*

[GRAND TOTAL: 100 marks]

[END OF COURSEWORK ASSIGNMENT 1]

Appendix I: About MySQL

MySQL is a major DBMS, originally open-source but now owned by Oracle Corporation. It is used in enterprises all over the world.

Here is an excerpt from a job advertisement (August 2016). Note the database systems with which they want their prospective employee to be familiar:

> '*The [name omitted] database team works closely with developers, operations and client groups to provide a 'full stack' perspective on providing highly available data services at scale. We believe a polyglot approach to databases is the best way to learn and to solve today's challenging data problems. Like Postgres? Prefer MySQL? Maybe you fancy NoSQL-style data stores? Everyone has their favourites, but understanding database fundamentals, and how the properties of different database systems interact with applications and operating systems is a key to our success.'*

Their requirements are*:*

- '3-4+ years working with two or more database systems including Postgres, MySQL, Oracle, or MSSQL.
- 4+ years working in Unix/Linux environments, particularly with web facing systems.
- Proficiency *in* tuning database processes and queries, both physically and logically.
- A solid understanding of database replication, including Master-Slave, Master-Master, and Distributed setups.'

You can download MySQL from here: http://dev.mysql.com/downloads/mysql
(Get the 5.6 version for whatever Operating System you have. If you already have an earlier version of MySQL, it is not necessary to download a later version.)

You can read about MySQL here: https://en.wikipedia.org/wiki/MySQL

A handy list of MySQL commands can be found here: https://en.wikibooks.org/wiki/MySQL/CheatSheet

A list of administrator commands for MySQL can be found here: http://refcardz.dzone.com/refcardz/essential-mysql
(These are not necessary for this coursework assignment, but they may prove useful if you wish to go further with MySQL.)

(**TIP**: the Dzone site has many other free downloadable 'ref cards' for other computing topics which you may find useful on other courses, and/or for your computing knowledge in general. You could print the relevant 'cards' out, using a colour printer, and post them some place where you will see them every day.)

Here are online forums for discussing issues relating to MySQL or getting help with it:

http://lists.mysql.com/ – This site has links to several city-specific user groups, and to user group mailing lists in languages other than English.
http://mariadb.org/en/ – This is a 'drop-in' equivalent of MySQL. It was started by people concerned about MySQL's public status, after Oracle Corporation bought it. They fear it will eventually be left to wither on the vine. Oracle has already started to charge high prices for extensions to the core product (which remains free). You can read about MariaDB here: https://en.wikipedia.org/wiki/MariaDB.

And here: http://www.devshed.com/c/a/mysql/oracle-unveils-mysql-5-6/

You can see comparisons of some of the most common database systems here: https://en.wikipedia.org/wiki/Comparison_of_relational_database_management_systems

Appendix II: Notes on data

'Type' versus 'Instance'

Natural language often does not distinguish between a physical instance of something, a single example of it, on the one hand, and its 'type' on the other. Thus, the description of the data items in the coursework assignment sometimes refers to 'Parts', and sometimes to 'Part-types'. In this context, 'Parts' means the same thing as 'Part-types'. We record information about a particular type of Part – its name and weight – but not about each 'instance' of a part. (In other situations – say, if we were building automobile engines – we might well record information about each single item – the serial number stamped on it, perhaps, or when it was manufactured and where.) Another way to think about the distinction: the company will have on hand only a small number of 'Part-types', but possibly many thousands of 'Parts'.

A good way to understand the distinction is to think about the different uses of the word 'dog' in the following two sentences. 'The dog is a common pet in many parts of the world.' 'The dog is barking loudly.'

The word 'dependency'

We use the word 'dependency' to indicate that one data item has a specific relation to another. For instance, we say that there is a dependency between a person (or, more precisely, a data item identifying a person) and that person's birthdate.

Note that a person can have only one birthdate, but that a particular date will be the birthdate of many persons. In the first case the dependency is 'one-valued' from Person to Birthdate, and 'many-valued' from Date to Person. We call 'one-valued' dependencies 'Functional Dependencies' (FDs) and the other kind, 'Multi-valued dependencies' (MVDs).

Note that when we talk of Functional or Multi-valued dependencies, they are not 'between' items, but from one item (or set of items – see next paragraph) but **from** one **to** the other, and vice versa. So, between any two 'things' there are two dependencies, one in each direction.

Many dependencies are just between two data items, as in the previous paragraph, from person to birthdate. But there can be dependencies among **more** than two data items: for example, among a **person** and a course and the final **grade** that person receives in that course. In the Person and Course and Final Grade example, the Final Grade depends on **both** Person **and** Course.

Dependencies can also be 'transitive': A person has one country of birth, and that country of birth has a capital city. This is not a three-part dependency, but two two-part Functional Dependencies with a common data item. The dependency of Person and Capital City is not a direct one (in the case where we know their country of birth) but is indirect – via the 'transitive' relationship among Person, Country and Capital City. If we know a person's country of birth, and if we know that country's capital city, we can derive – by looking up – that person's capital city. In the second example, Capital City depends only on Country, and Country depends only on Person. (Note that we are ignoring pesky 'real-world' problems, such as countries which change their names, or are split into two countries, or amalgamated into another country, or which have more than one capital city over time, or in which the capital city changes its name, and so on.)

We can also have more-than-two (sometimes called 'n-ary') relationships where there are no Functional Dependencies at all. Suppose Students can take Courses, but for each Course, several different Textbooks are recommended out of a larger pool of possible suitable textbooks, and the student chooses two or three. Some of these textbooks are used in several different courses. Now, if we want to know which students are using which textbooks in which courses, we have to record that information explicitly in a three-part dependency – but all dependencies are multi-valued. Knowing any one of the values for Student, Course or Textbook does not allow me to predict the other two. Nor does knowing any pair of values allow me to predict the other one. I must explicitly record all three.

Now...in daily life these are just possibly-interesting academic distinctions. We only pay attention to them in database design because the relational model, for good reasons, requires us to pay particular attention to Functional Dependencies. We have to get better at spotting when there is a Functional Dependency among data items, because this allows us to design efficient, robust tables.

Data and Reality
For the most part we use the word 'data' to identify 'things' in the real world. These data are represented to us by symbols. (Modern databases can also store sounds and images, and enormous blocks of text, but although these could be called 'data' in a certain sense, here we will restrict the term 'data' to mean 'relatively short strings of symbols'.)

The things that these data can refer to are exactly as numerous as the things language itself can refer to: people, places, smells, colours, quantities of money, rates of inflation, dates, what-have-you. If you can name it, then its name can be entered as data in a database.

Of course, all the problems to which language is subject – ambiguity, incorrectness, vagueness – can also apply to data, which is just a 'frozen language'. For instance, suppose we have an attribute called 'COLOUR' in a relation, and we want to record the data value 'blue' in some of the tuples. Whether we can do this or not will depend on what language the data is being recorded in. Some languages have no word that corresponds to 'blue' (Ancient Greek); others have no word for blue in general, but two words for what in other cultures are perceived as two different **shades** of blue (Russian), etc.

These issues are outside the scope of database theory, except for two aspects of the problem:

(1) Data reliability. The data we enter into the database may be incorrect, through human or other error, dishonesty, etc. This is why key features of modern database management systems are the various mechanisms to try to ensure data integrity. We call these mechanisms 'constraints'.

To try to ensure data integrity, we try to define the domains ('data types') of each attribute as narrowly as possible: for example, we don't permit salaries of zero or less, we don't have employees born in the 19th century, and so on. We call this 'attribute integrity'. The rules about Candidate and Foreign Keys (no nulls in Candidate Keys, Foreign Keys must refer to existing values in other relations) try to ensure 'entity integrity' and 'referential integrity' respectively.

Important MySQL note: The way we try to ensure 'attribute integrity' (no DATEs earlier than 1900/01/01, for example) is through the CHECK command. However, although MySQL will parse this command – that is, it will not throw an error if you use it in a CREATE <tablename> statement, it will not implement it either. There are workarounds using the TRIGGER statement, but you are not responsible for knowing them for this course (just that they exist). We can also have 'stored procedures' to do more elaborate checking on data: for example, to implement a 'check digit' procedure for data where we have incorporated this method of trying to ensure data integrity.

(2) Data representation. There is often more than one way to represent something in the real world.

Precision: When we measure, we have to decide how precise our measurements will be. For example, the length of a board can be represented with various degrees of precision:

50 cm, 50.1 cm, 50.07 cm, 50.068 cm. Assume that these measurements are all correct – each one has been made with the increasingly finer measuring device. Note that ‘accuracy’ and ‘precision’ are not the same thing. These concepts are ‘orthogonal’ to one another: we can be very precise, but inaccurate (if we said that the board in the previous sentence was 45.14297 cm long), accurate but not very precise (if we said the board in the previous example was between 40 and 60 cm long), etc.

Datatype: Often there is more than one datatype that can be used for an attribute. For example, if an attribute will only hold the numbers 0 to 9, we can choose between several datatypes that will do the job. As a rule, we want to choose the datatype that will take up the least space in memory (thus SMALLINT rather than INT or DECIMAL, or CHAR rather than VARCHAR). We also want it to be compatible with the operations we intend to use it in (will we compute with it?; in which case we should use a numeric datatype; or just display it?; in which case we should use a character datatype).

Format: The trickiest of all. Often, very common items – Passport Numbers, Employee Numbers, Telephone Numbers, Addresses – do **not** have a common format, especially when we expand the scope of our data collection beyond a single company or country. As ‘globalisation’ continues, these problems are slowly being overcome through standardisation, but they are still a serious problem, especially for the relational model which assumes, or at least does better with, a uniform format for data of the same semantic sort, i.e. data which has the same ‘meaning’ or use. To take an example, when two companies undergo a merger, and each of them had a different format for their employee numbers – one using pure integers, for example, while the other uses a character string – there will be a problem in merging their respective employee databases.

Similarly, trying to design a database that can hold addresses is another example, especially when these addresses can be in many different countries, each of which has different conventions for their postal codes, geographic subdivisions, etc. Often, we have to bodge the solution by redefining the common format as just a collection of ‘string’ types, as when ADDRESSes become ‘ADDRESS-LINE-1’, ‘ADDRESS-LINE-2’, ‘ADDRESS-LINE-3’, which obliterates useful information about streets, buildings, geographic units [provinces, states, nations within federations, postal codes, etc.   If my database held only geographic addresses within the USA, for instance, I could have an attribute called STATE, and search for all addresses where STATE = ‘Texas’. But an international address database will require a much more elaborate design for its addresses.

Appendix III: Names and keys

See the subject guide, *Database Systems Volume 1*, pages 59–63.

Names are words which identify things in the real world. For the purposes of this discussion, let's call them by a somewhat broader term, 'identifiers'.

Look at the table called **SUPPLIER-MASTER** in **Part E.** You may have wondered why we needed both SNOs *and* **NAME**s to identify a particular Supplier. Why not just use the **NAME** alone, especially since it's probably how the Suppliers refer to themselves, and also how our employees refer to them?

The answer is, we often give things 'artificial names' (sometimes called 'surrogates') to identify them in database work because their 'natural names' are inadequate for our purpose. More than one person can have the same name, a person (or company) can change its name (through marriage, or merger), chemicals can be spelled with slight variations ('sulfur' versus 'sulphur'), the names of cities can have several spelling variants or even be different ('Leningrad' versus 'St Petersburg'), and so on. This process of creating systematic, rational substitutes – formal identifiers – for 'natural' names in fact precedes the development of computerised systems by many decades.

You probably have several formal identifiers that are – or should be – associated with you alone: your passport number, military service ID, student number, *etc*. If you have a car, there will be an engine number unique to it. Your smartphone will have a unique ID that will remain the same even if you change phone numbers. Book titles have 'ISBN's (International Standard Book Numbers), airports have character codes.

When we choose 'artificial names' for things that we want to uniquely identify, we need to take several things into account:

Growth: We want to be sure that we will not 'outgrow' the identifier's datatype, by eventually having more items than the datatype can represent. (If your 'artificial name' is a four-digit number, you can only identify 10,000 things, assuming every four-digit number, from 0000 to 9999 is valid.)

Human factors: We also want to take human weaknesses into account. For instance, if there is any chance of ambiguity, we will want to avoid using symbols that are easily confused with each other, such as 0 and O, 5 and S, l and 1, etc. to avoid data entry/transcription errors), assuming our language uses the Roman alphabet.

Error detection: We might want to embed error-detection into our names via a 'Check Digit' (as is done with credit card numbers and ISBN numbers that identify book titles).

Stability: Even unique artificial identifiers might not be stable, with respect to the thing they are supposed to identify. In some countries, when your passport expires, your new passport will have a new number. So, using 'passport number' to identify someone may not be a good idea. An artificial identifier generated by your system will be guaranteed to be stable, if you avoid the practice discussed in the next paragraph.

Embedded information: Should a Primary Key that we generate simply be an identifier, or should it also carry information 'inside' itself? For instance, suppose we want to design an Employee ID number that will uniquely identify each employee, and we are also aware that in future we will want to know how long an employee has worked for us. We *could* incorporate the date that the employee was hired into their Employee ID number, with an extra two digits if more than one employee was hired that day. Therefore, '98061200' and '98061201' might be the employee numbers of two employees hired on June 12th 1998. (The alternative would be to have a separate attribute in the employee master file, recording the date hired. This will require retrieving that attribute along with the employee number when we do a query.)

However, by doing this, we have opened ourselves up to several potential problems: what if we hire more than 100 employees on a certain day? What if an employee quits, and is then rehired?

The moral is to think twice before making an attribute serve both as an identifier, and as an information-bearer. Unless there are strong considerations to indicate otherwise, an attribute that is designed to uniquely identify an instance of an entity type should not do double-duty as an information-bearing attribute also, if there is any chance that the 'same' instance of an entity-type could see the embedded information in its identifier become obsolete, or perhaps not be available in some cases in the future. (If you make an employee number include the employee's birthdate, for purposes of calculating later pension payments perhaps, what happens if you hire an employee who learns that their original birthdate was mistakenly-recorded and that they are actually two years older (or younger) than they originally thought? If their birthdate is recorded as a separate attribute, this can simply be changed – but if it is part of their unique identifier, you will have a problem.

Computing considerations:

Efficiency of processing. The datatype and format of an identifier can affect how efficiently it is processed by a computer. If an identifier is going to be a Primary Key, this is especially important, since database systems often index Primary Keys automatically, and our queries will often use these indexes in searches. Thus – for reasons of creating an easily searchable index – we should ideally make our Primary Keys fixed-width (*i.e.* Not VARCHAR, or, at least that is the conventional wisdom), and we should choose, where possible, integer datatypes over character, and fixed-width character over varying width character. (Note that not everyone agrees with this.)

Note: This rule is generally **not** followed in the toy databases used in teaching materials and coursework assignments, since we want to make it easy for readers to distinguish out-of-context data. For example, we might use 'E123' as an employee identifier and 'P0001 23' as a project identifier (that is, we might incorporate characters into the identifier so that the datatype has to be a character type) rather than using digits only. This would allow us to use a numeric type, so that it is immediately obvious to the reader what kind of identifiers they are. However, in designing a real database, efficiency is the major consideration.

Case sensitivity: Another consideration is whether data is going to be moved between systems (either between different database management systems, or between Operating Systems (*e.g.* UNIX-based systems), or processed by certain programming languages, (*e.g.* C++, Java, Python). If this is the case, be aware that some systems are case-sensitive and others (*e.g.* Windows) are not; however, it is often possible to reverse this if you need to. As a quick rule of thumb, case-insensitive is best (so that 'PNUM' and 'pnum' and 'PnUm' are treated as the same thing – in case-sensitive systems, they are three different things). Therefore, be aware that if you run into subtle problems in transferring the database data from one system to another, differences in case-sensitivity may be the cause.

Primary keys: when an identifier is going to be used as a Primary Key (or part of a composite Primary Key), it is doubly-important to get it right. Remember that a Primary Key is a Candidate Key that we have chosen to play the role of the Primary Key. (Most of the time, there is just one Candidate Key, so we do not even have to choose.) Remember also that a Candidate Key can be made up of more than one attribute. (This is **not** the same as having two or more Candidate Keys.) A Candidate Key, made up of one attribute, or more than one, uniquely identifies a tuple. **In other words, in a relation, there cannot be more than one tuple with the same Candidate Key.**

The 'autoincrement' temptation: every database management system, including MySQL, has a method which allows someone creating a database to have the system automatically insert a sequence of numeric values into a column. So, you can design a table so that, say, the first column has a 1 in the first tuple, a 2 in the second tuple, and so on. Note that this makes this attribute a Candidate Key, since no two tuples will have the same value here. For people who do not really understand Key design, it's a temptation to do this, and just declare this attribute to be the Primary Key.

However, if the relation has a 'natural' Primary Key, then it is a mistake to do this. Consider one of the Master Relations in this coursework assignment – say, the Part-Master relation. We could add an attribute (column) to this relation and make it an 'auto-increment' one, and declare that it is the Primary Key. However, then we could have two different tuples, each with the same PCODE, but with different attributes – a disaster when it comes to assembling SatNav devices. Also, most queries involving relations will involve reference to 'natural' attributes.

Sometimes, AUTOINCREMENT can be useful, such as if a tuple has no 'natural' key, and we would just assign an arbitrary identifier to it anyway. We might have wanted to treat Order Numbers this way, or employee numbers. The autoincrement function would guarantee that we did not give two different orders the same number, and it would get around the problem mentioned in the last paragraph of the previous section headed 'Embedded Information'. (Our current Order Numbers seem to have embed the Order Date, which could be mistakenly recorded.)

Another use for putting an AUTOINCREMENT attribute in a relation is to allow us to order our tuples in the order in which they were created, should we think this might be useful information in the future. Note, however, that it would be possible to have a 'Date-created' attribute in a relation that would do the same thing, and provide more information.

The important lesson here is: do not just casually use AUTOINCREMENT because you cannot work out what attributes or combination of attributes constitutes a 'natural' Primary Key.

Note that SQL does **not** automatically enforce this rule in its **queries***:* you **must** use **SELECT DISTINCT** and not just **SELECT** if you want your resulting tables to be relations – that is, to have no duplicate values in 'Primary Key' column(s) (and you almost always **do** want this). It **does** enforce the rule in the original relations, but **not** in the relations that result from queries.

IMPORTANT!!!

Appendix IV: 'Normalizing' relations

The goal of relational design is to end up with a set of 'normalized' relations. (Please note: although we sometimes leave some relations in less than completely normalized form to improve performance, for your coursework assignments and in the examination you should assume that all your relations are fully normalized, unless told otherwise.)

You should consult the subject guide, *Database Systems Volume 1,* pages 132–145 where there is a thorough exposition of normalization.

There are three things to be aware of as you read the subject guide, or other sources of information on normalization:

NORMALIZATION

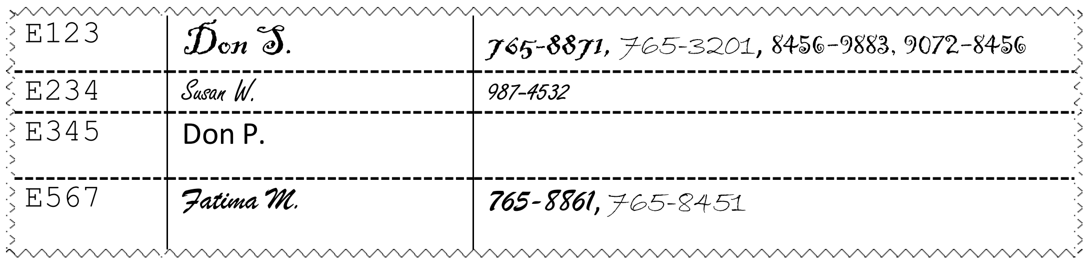
The word 'normalization' can be confusing; unfortunately, we are stuck with this word. Beginners are sometimes confused by the word because they have encountered it at some other point in their studies. The word 'normalization' is used in statistics, mathematics, physics, sociology, neurology, in various areas of technology and also in computing. Even **within** each of these fields, including computing, it can be used in several different, completely unrelated ways. So, remember that we are talking about '**database** normalization', which has nothing to do with any other use of the word, except in the very general sense of making something more usable. In the database context, as the subject guide explains, it simply means taking one relation, and splitting it up into two or more relations, which are equivalent, in terms of the information they hold, to the first relation, but which have desirable properties the first, single, relation did not have.

The 'Normal forms': The 'normal forms' are like a set of concentric circles. Normalization is the process of going from the outer circle to the innermost circle. The outermost circle is First Normal Form. When you take a step toward the centre, you end up in the second circle, which is like Second Normal Form. If you are in Second Normal Form you are also in First Normal Form. And so on.

Higher Normal forms: The Fourth and Fifth Normal Forms are sometimes called the 'higher' normal forms. Note that by the time you have normalized a set of relations to Boyce-Codd Normal Form (BCNF, sometimes called 'three-and-half Formal Form'), you almost certainly have a set of relations that are also in Fourth and Fifth Normal Form. You only have to take special steps to get to these higher normal forms in certain very rare situations.

'REPEATING GROUPS'

In First Normal Form we want to eliminate 'repeating groups'. Consider phone numbers. Suppose we want to record the phone numbers of employees. (An employee can have no phone, one, or more than one telephone number.) If we were recording this information on a piece of paper, we might well do it this way:



E123	Don S.	765-8871, 765-3201, 8456-9883, 9072-8456
E234	Susan W.	987-4532
E345	Don P.	
E567	Fatima M.	765-8861, 765-8451

Now it is actually possible to do this in a relational database (ugly, even criminal, but possible)

All we have to do is to declare the attribute **PHONE-NUMBERS** of type 'string' (VARCHAR), making the maximum size of the string as large as possible (VARCHAR(<whatever the maximum your DBMS allows>)), and we can place all the phone numbers into one attribute in each tuple. In other words, we can make our relation look like the paper and pencil example above.

Design 1a

EMPLOYEE-PHONE

EMPNUM	NAME	PHONE-NUMBERS
E123	Don S.	765-8871, 765-3201, 8456-9883, 9072-8456
E234	Susan W.	987-4532
E345	Don P.	
E567	Fatima M.	765-8861, 765-8451

However, it's a very bad idea to do so!

It will make your searches, insertions and deletions on **PHONE-NUMBERS** very complicated, as you'll have to use the (slow) *LIKE* and sub-string facilities of SQL. It will make your printouts ugly. You will have trouble exporting your data to other systems, should you need to. You won't be able to do a *COUNT* on the attribute, or a *JOIN*.

If that argument doesn't convince you, consider this: Why have separate attributes at all? We could have a relation like this, with just one attribute instead of two:

Design 1b

EMPLOYEE-PHONE

EMPNUM-AND-PHONE-NUMBERS
E123, Don S., 765-8871, 765-3201, 8456-9883, 9072-8456
E234, Susan W.,987-4532
E345, Don P,
E567, Fatima M., 765-8861, 765-8451

For that matter, why have separate tuples? If the database system will allow us to make our strings long enough, we could have just one attribute and just one tuple, filled with one giant monster string of characters:

Design 1c

EMPLOYEE-PHONE

EMPNUMS-AND-NAME-PHONE-NUMBERS-AND-EVERYTHING
E123, Don S., 765-8871, 765-3201, 8456-9883, 9072-8456, E234, Susan W.,987-4532, E345, Don P., E567, Fatima M.,765-8861, 765-8451

For that matter, why have separate relations? But enough of this.

Here's a better solution, but still not a good one:

Design 2

EMPLOYEE-PHONE

EMPNUM	NAME	PHONE-1	PHONE-2	PHONE-3	PHONE-4
E123	Don S.	765-8871	765-3201	8456-9883	9072-8456
E234	Susan W.	987-4532	NULL	NULL	NULL
E345	Don P.	NULL	NULL	NULL	NULL
E567	Fatima M.	765-8861	765-8451	NULL	NULL

This design is in First Normal Form in the technical sense – each attribute of each tuple has a single 'atomic' value – or one we have chosen to think of as atomic – in it (although not necessarily an 'atomic value', in the strict sense, as we have seen/will see). However, there are two kinds of objections to this design: it will be awkward to add a sixth phone number, and it is awkward to query. Also, from a 'theoretical' point of view, it makes distinctions among phone numbers which do not reflect real life distinctions. What is the difference between a 'PHONE-1' and a 'PHONE-2'? (If these were different types of phone – home versus work, or landline versus mobile, that would be different. But the columns here are arbitrary.)

An alternative design that does make phone numbers easy to query in SQL (which is what is wrong with Designs 1 and 2), is the following.

Design 3

EMPLOYEE-PHONE

EMPNUM	NAME	PHONE
E123	Don S.	765-8871
E123	Don S.	765-3201
E123	Don S.	8456-9883
E123	Don S.	9072-8456
E234	Susan W.	987-4532
E567	Fatima M.	765-8861
E567	Fatima M.	765-8451

This design, however, not only duplicates data (the employees' names), but it has a fatal weakness when it comes to implementing insertions, deletions, and modifications on it. See page 107 of the subject guide, *Database systems, Volume 1* for an explanation of the problem with a relation designed like this. Please note, however, that when it comes to **displaying** our Phone List, we might well want to create a 'View' that looks like this, or even one that looks like Design 1A. This is easy to do in SQL. However, here we are talking about the underlying 'logical' relations that we will store the data in.

The best solution – although to beginners, this may look perverse and unnecessarily complicated – is to have **two** relations:

Design 4

EMPLOYEES

EMPNUM	NAME
E123	Don S.
E234	Susan W.
E345	Don P.
E567	Fatima M.

EMPLOYEE-PHONE

EMPNUM	PHONE
E123	765-8871
E123	765-3201
E123	8456-9883
E123	9072-8456
E234	987-4532
E567	765-8861
E567	765-8451

Beginners often raise the following objection: 'Are we not duplicating data here? In the first two designs, a particular employee number appeared just once. In this design, it can appear many times.'

This objection may have had some value 40 years ago, when the maximum size of direct-access secondary data storage was measured in kilobytes. It is of minor importance today. (In any case, the NULLs of Design 2 take up space of their own. See the concept of a 'sparse matrix' if you want to know more.) More importantly, this sort of 'duplication' allows us to design efficient database systems that are simple and easy to understand (once you get used to the relational idea).

Note that we do need two relations: one to hold our employee names and numbers, including those employees without phones, and the second to hold the information about employees with phones.

Each tuple in the two relations mentioned above records a single 'association fact'. The relation EMPLOYEES is both an 'existence list'... it tells us who our employees are – and it also tells us their names. The relation EMPLOYEE-PHONE tells us the valid associations of each employee with a phone number. An employee with no phone does not appear in the second relation. Since an employee can only have one name, we can store the name in the 'existence list'. However, because an employee can have more than one phone (i.e. there can be more than one 'phone association fact' for a single employee), we need a second relation to store them.

(Note: if we were **absolutely sure** that two employees will **never ever** share a single phone number – we could have made the PHONE alone in EMPLOYEE-PHONE the Primary Key. But this is a very poor assumption to make.)

These relations are easy to query, and a specific SQL query will remain valid no matter how many phones an employee has. An employee with eight phones will just take up eight rows in the second table. Whereas in Design 2, we would have to add four more columns to the whole relation, so that there were eight phone columns, which is a major undertaking. (Most likely, the whole database would have to be expanded on the disc.)

Be sure you understand why Design 4 is by far the best way to handle multi-valued attributes that are associated with ‘base relation’ Primary Keys. Because this is **not** the way we would design a paper-and-pencil ‘database’, it is one of the most common misconceptions that beginners have when they start using computer-based relational databases.