University of London International Programmes
CO2209 Database systems
Coursework assignment 1 2016–2017

General information

Important

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

FamilyName_SRN_COxxxxcw#.pdf (e.g. Zuckerberg_920000000_CO2209cw1.pdf)

- **FamilyName** is your family name (also known as last name or surname) 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 CO1108, and
- **cw#** is either cw1 (coursework 1) or cw2 (coursework 2).

IMPORTANT: your coursework assignment should be submitted as a single PDF file, using the file-naming conventions that you've been given. If you're not sure how to do this, please come to the online forum for this course and ask.

It should take between 20 and 40-hours to complete this coursework assignment; however, this will depend on how much you already know about the topics. There are some easy parts and some that are more challenging.

Each part of the coursework has been chosen to help you understand a key issue in the subject of databases; it should be undertaken with the subject guide, Volume I, at hand. There are six Appendices at the end of this coursework assignment 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. Parts **A, B** and **E** require downloading and using a database system; however, the other questions can be started straightaway. If you have never encountered relational database ideas before then the terms will be unfamiliar to you. 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 are your chances 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 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

And therefore that you attempt each question in it. 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 guides, but in addition to these and your textbook, you will want to consult the wealth of information available via the internet. In **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.

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, [http://www.en.wikipedia.org/wiki/Comparison_of_MySQL_database_enginesyou 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 forum.

Coursework 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 and real life

This coursework 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 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.

Coursework assignment 1

The first part of this coursework assignment has five tasks:

1. Downloading and setting up the software for managing a database.
2. Implementing a 'toy' database with MySQL.
3. Describing this database graphically.
4. Finding out how to get help from experienced database users.
5. Running some queries on the database you have implemented.

A. Setting up a DBMS

You can – and **should** – get started on this part of the coursework 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 don't 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 online Forum 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: http://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.

[Time: 2 hours. Marks: 5]

B. Creating a (toy) database with MySQL

The purpose of doing this exercise is to give anyone who has never used a database before, 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. They are also much more complicated in terms of their structure – with more and larger relations, and more complicated relationships being captured by those relations.

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

While doing this coursework assignment, you should consult pages 71–102 of the subject guide, *Database systems*, Volume 1, which covers SQL.

The background
A company that sells vitamins and medicinal herbs wants you to implement a database describing its products. Each distinct type of vitamin or herb it sells has a unique PRODNUM, a NAME, a single QUANTITY (how many pills or capsules in a container) and may – but does not always – have a particular employee who possesses specialist knowledge of that product who has been designated the 'go to' person for information about it. An employee may be a specialist in more than one product, but a product will have only one employee-specialist listed for it.

It also wants to record which customer has ordered how much of which product, and the date of the order. A customer never places more than one order for the same product on the same date.

This miniature database will have only two tables, linked by a single common attribute. The purpose of doing this exercise is to give anyone who has never used a database before, some experience in setting up and querying one. Everything has been made as simple as possible – real databases are a bit more complicated!

Relation Name:	**PRODUCT**	
Attributes:	**PRODNUM**	Primary Key. A number, ranging from 0 to 1 million.
	NAME	A string of characters, from 4 to 36 characters long. Note that more than one product may have the same NAME.
	FORM	**How the product is packaged: as a pill, a capsule, loose, etc.**
	QUANTITY	A number, ranging from 1 to 500. This number is how many units of the product are in a single container.
	EMPNUM	A number, ranging from 0 to 1,000,000. This field may

be empty, in which case we put the keyword NULL there.)

PRODNUM	NAME	FORM	QUANTITY	EMPNUM
2357	B12 – 50 mcg	capsule	100	112358
1113	B12 – 15 mcg	pill	60	132134
1719	D3 – 20 mcg	pill	240	132134
2323	Glucosamine – 500 mg	pill	100	558914
2931	Devil's Claw 1oz	loose	NULL	558914
3737	D3 – 400 mcg	pill	240	NULL
4143	Wheatgrass extract	capsule	24	558914
4753	Dextromethorphan – 15 ml	capsule	12	314159
9193	Quinidine Sulphate	capsule	28	NULL
2358	C – 500 mcg	pill	240	198887

Relation Name:	**ORDERS: Primary Key will be CUSTNUM + PRODNUM + DATE**
CUSTNUM	**A whole number, ranging from 0 to 999999**
PRODNUM	**A whole number, ranging from 0 to 10000; this is a Foreign Key which references PRODNUM in PRODUCT**
DATE	**A date. Must be > 01-01-1993**
QTY	**A whole number, must be greater than 0 …. Never above 10000**

CUSTNUM	PRODNUM	DATE	QTY
136101	2357	21-03-2016	05
136101	2357	12-10-2016	01
136101	3737	12-10-2016	10
212836	3737	16-09-2015	06
455566	4143	09-02-2016	10
182764	2357	21-03-2015	12
125216	2323	21-03-2016	02
182764	2357	12-05-2016	10
455566	4143	12-05-2016	10
136101	2357	25-11-2016	05
136101	9193	25-11-2016	05

What to submit: You should submit a copy of the SQL statements you used to create your database, along with a list of the tables that you created.

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 (in this example, I have placed it on my D: disc, but it 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.

[5 hours, 10 marks]

C. Entity/Relationship diagrams

Draw an Entity/Relationship diagram showing the relationships among Customers, Products and Employees, as recorded in your database. You need only show Entity Types and their relationships, not attributes. 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.

[1 hour, 5 marks]

D. Getting help

Find an online forum which deals with MySQL (either MySQL alone, or with other database systems as well). On that forum, find a post that asks a question or poses a problem about using MySQL, which the poster needs to solve. Describe the poster's question, and give one or more suggestions by others as to how the problem could be solved.

(**Appendix I** in this coursework assignment has suggestions for possible forums, or you can find others.)

[1 hour, 5 marks]

E. SQL queries

When your database has been created, create SQL queries to answer the questions below:

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

EXAMPLE Query Query in natural

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

SQL

```
SELECT DISTINCT  CUSTNUM
FROM ORDER
WHERE DATE  =
                    (SELECT MIN(DATE)
                     FROM ORDER);
```

RESULTS

```
CUSTNUM
212836
```

(01) What is the name of the product whose PRODNUM is 2357?

(02) How many customers have ordered the product whose PRODNUM is 2357?

(03) What is the average quantity for orders for the product whose PRODNUM is 2357?

(04) What are the PRODNUMs of products that do not have an employee assigned to them?

(05) What are CUSTNUMs of Customers who have ordered a product that is in capsule form?

(06) When was the most recent order, from the customer whose CUSTNUM is 182764?

(07) What are the PRODNUMs of products that have not been ordered by anyone? [HINT: Imagine we have a set of all products, and we remove from it each product that HAS been ordered at least once. What do we have left?]

(08) What are CUSTNUMs of customers who have ordered both the product whose PRODNUM is 3737 and the product whose PRODNUM is 9193?

(09) What is the CUSTNUM of the customer who has ordered the greatest quantity of Products (there may be a tie)?

(10) What is the CUSTNUM of the customer who has ordered the greatest quantity of Products in 2015(there may be a tie)?

What to submit: **(1)** The natural language version of the query (just a copy of the questions above).
(2) Your *SQL* **query** for each question.
(3) The **results** of running that query.

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.** (Why should you repeat the natural language version of the query? Repetition will help you when you use this coursework assignment as part of your revision.)

Be sure to number your answers, using the numbers shown above. Remember: *do not submit screenshots for this question. Queries (01) to (05) are worth 4 marks each, and (06) to (10) are worth 5 marks each.*

[8 hours, 45 marks]

F. Functional dependencies

We show a 'functional dependency' between A and B (meaning, for each value of A, there is at most one B – for example, a person and their mother) through a single-headed arrow, like this. A -> B. This says nothing about B's relationship to A.

Sometimes 'multivalued' dependencies (for each value of A, there can be more than one value of B, for example, a person, and their siblings) are shown like this A ->> B.

Using this notation, and considering the attributes PRODNUM and EMPNUM as identifiers for Products and Employees as described above, how would we show the following possible relationships? (Note: three of these are NOT the actual relationships shown in the previous question.

- An Employee can look after only one Product. A Product can be looked after by only one Employee.
- An Employee can look after only one Product. A Product can be looked after by more than one Employee.
- An Employee can look after more than one Product. A Product can be looked after by only one Employee.
- An Employee can look after more than one Product. A Product can be looked after by more than one Employee.

[1 hour, 4 marks]

G. Relational concepts

Write a brief answer to show what we mean by the following terms, as they apply to relational databases. These terms are often confused, so it's important - especially for the examination - that you know the actual meanings of these terms.

(a) Attributes in Entity Diagrams and Attributes in Relations. (They are not the same!)
(b) The relation between 'Candidate Keys' and 'Primary Keys'. (Consider the relationship between 'triangles' and 'right-angled triangles.')
(c) What 'A determines B' actually means. Don't use the word 'determines' in your answer.

[See subject guide, Volume 1, pp.37–63]

[1 hour, 6 marks]

H. Relational design

In your work in Part **B**, you were given two relations. One held information about Products, and the other held information about the relationship between Customers and Products.

In the first relation, about Products, you were told that advice about a given Product was the responsibility of at least one Employee. Suppose the company decided to change this policy, and let more than one employee be designated as responsible for advice about a product. For example, dextromethorphan might become the responsibility of the employee whose employee number is 198887, while also remaining the responsibility of the employee whose employee number is 314159.

Design a new relational schema that can hold this information.

(A 'relational schema' is just a basic description of a relation: its name, and the names and types of its attributes (columns)). The relations given in Part (b) are preceded by their schemas.

What to submit: Relational schemas that will meet the new requirements. Your schemas need not show any relations that do not have to be changed. You don't have to include sample data.

[4 hours, 10 marks]

I. 'Big Data'

Write a brief (no more than two-page) essay describing 'NoSQL' database systems. You need not go into elaborate detail, but your essay should allow someone who knows about relational databases, but not about 'NoSQL' databases, to get a basic understanding of (1) what they are, and (2) why they have come into existence, when the relational model ruled effectively unchallenged for twenty-five years. In other words, what sort of data do they typically deal with, for which relational databases were not a natural solution**? Cut-and-paste submissions will receive no credit.**

[3 hours, 10 marks]

[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: http://www.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: http://en.wikipedia.org/wiki/MariaDB.

And here: www.devshed.com/c/a/mysql/oracle-unveils-mysql-5-6/ - more-448.

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

Appendix II: Notes on data

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 '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. The rules about Candidate and Foreign Keys try to ensure 'entity integrity' and 'referential integrity'.

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. 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).

Appendix III: Names and keys

See the subject guide, Volume 1, pp.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 **Product** in **Part B**. You may have wondered why we needed both **PRODNUM**s *and* **NAME**s to identify a particular product. Why not just use the **NAME** alone, especially since it's probably how the customers and employees themselves refer to the product?

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 their name (through marriage, or merger), vitamins 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.

Computer 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 'P123' 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 be may 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 don’t 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. 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 (and you usually **do** want this).

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, Volume 1, pp.132–45, 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 present 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 is a better solution, but it is 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.

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
E345	Don P.	NULL
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, Volume 1, for an explanation of the problem with a relation designed like this.

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: 'Aren't we 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 doesn't 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.

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.)

[END OF APPENDICES FOR COURSEWORK ASSIGNMENT 1]