

University of London
Computing and Information Systems/Creative Computing
CO2209 Database systems
Coursework assignment 1 2018-19

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 on 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).

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 a key issue on the subject of databases. It should be undertaken with the subject guide, Vol. 1, to hand. There are four appendices at the back to supplement the information in the subject guide.

The best way to approach this coursework 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. Parts **A**, **B**, and **E** require downloading and using a database system, but the other parts can be started even before you do that. 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 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 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's important for you to complete this coursework

The coursework assignment has two practical aims. Firstly, to provide you with a 'rehearsal' for the examination. Secondly, and 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, Vol. 1, especially Chapters 3 and 4. However, in addition to these and your textbook, you will want to consult the wealth of information available on the internet. In **Appendix I**, we have provided some links relating to MySQL to start with, 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 on practical issues involved in database design in the other appendices to this coursework assignment. Be sure to familiarise yourself with the content of the appendices. 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 would treat ordinary references because Wikipedia articles can be written by anyone, can have multiple authors (who are usually anonymous), and the content 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 comparing the two (at least, the article online in the autumn of 2018), [https://en.wikipedia.org/wiki/Comparison_of_MySQL_database_engines], you will see that it has clearly 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.

Therefore, you should never rely **only** on Wikipedia as a source of information. When you do 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 to 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 are not sure about something regarding databases that you have found on 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, a sort of two-for-the-price-of-one deal. You will notice that both coursework assignments include some very simple initial tasks, which consist essentially of having you pay close, systematic attention to the subject guide and the MySQL manual, 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 have a ready-made set of revision materials. However, please note, your revision should start in December at the latest, not in April.

Coursework 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 coursework assignment 2, of using a genuine database. In addition, some of the questions in coursework assignment 2 will 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. 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 and examinations. 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, in that it's actually easier to get help than in many traditional courses, if you take advantage of the facilities offered.

This coursework assignment is comprised of five parts (A-E): **A** downloading and setting up the software for managing a database, **B** becoming familiar with the manual, **C** implementing a 'toy' database, **D** seeing what should happen if we violate some of the rules governing what data can, and cannot, be in the database, and **E** running some queries on the database itself.

If you complete all five parts, you should receive a good introduction to the fundamental ideas of database. Do not hesitate to use the course discussion board if you run into problems or have questions at any point. 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 come to discuss issues. **However, please be careful not to post anything that will form part of your submission.**

Part 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 any 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 8.0.13 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 to the course discussion board for this course straightaway to ask for 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. It was started by people concerned about MySQL's public status after it was bought by Oracle Corporation. They fear that it may eventually be left to flounder. You can read about MariaDB here: <http://en.wikipedia.org/wiki/MariaDB>

Note: Once you have downloaded the install package and have installed your database software, it may not be obvious how to run the package. Again, post to the course discussion board for help if this is the case.

Part A - 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 already had it on your computer), write a short report (no more than one page) summarizing the contents of the MySQL Manual. You can do this by listing and very briefly describing (but not just copying) the most important items in the Table of Contents.

[4 hours. 5 marks]

Part 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/8.0/en/tutorial.html>.

Note that 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:

1. What is the URL of the section entitled '**B.6.2 Common Errors When Using MySQL Programs**'?
2. What is the query that would have MySQL tell you your user name, its version number, and the current date?
3. 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?

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

Prompt	Meaning
mysql>	
->	
'>	
">	
`>	
/*>	

(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?

5. [Section 3.3] What query will tell us what databases currently exist?
6. What is the command to create a database called 'CHEMCO' if one does not exist?
7. Suppose we know that a database called 'CHEMCO' currently exists. How do we access it so that we can modify it or query it?
8. Suppose you create a database called 'CHEMCO', and later try to access it by typing USE ChemCO;

Does it make a difference (a) under Windows, and (b) under UNIX?

9. Once you are using a particular database, how can you see what tables it consists of?
10. If you know an existing table's name – let's say it's called ORDERS – 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, etc.)?

Part B - What to submit: Answers to questions 1—10. *Be sure to number your answers. Please start the answers to this question on a new page.*

[2 hours. 10 marks]

Part C: Implementing our database

Implement a database, consisting of the tables presented below, in MySQL and populate it with the data presented after the schema (a 'schema' is just a listing of the name of each table and the attribute names and data types of its columns, along with any constraints).

Subject guide reference: Pages 61, and 71-102 – especially page 78 – of the subject guide, *Database systems, Volume 1*, and the MySQL Manual: <https://dev.mysql.com/doc/refman/8.0/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 doing this exercise is to give anyone who has never created 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, 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.

A chemical supply company wants you to implement a database describing its products and customers and which customer has ordered which product. Each distinct chemical it sells has a unique CHEMNO, a DESCRIPTION (a brief name), a single DLEVEL which specifies its 'Danger Level' and should have – but does not always – the Employee Number (ENO) for a particular employee who possesses specialist knowledge of that chemical and has been designated the 'go to' person for information about 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. Chemicals which have not been ordered can exist in the database and customers can exist who have not yet ordered any chemical.

The schema has been created already. For this part of the coursework, you need to create a database and populate with the following tables and data.

NOTE: if you make a mistake in adding the data to these tables, it may mean that you get a wrong answer to one or more of the queries that you will be putting to these data later, and lose marks. For instance, if you listed a chemical's DLEVEL as 8 when it should have been 3, one of your queries will, even if the SQL is correct, generate a wrong answer. So double check your data after you have entered it.

RELATION: CHEMICALS
PRIMARY KEY: CHEMNO
ATTRIBUTES

CHEMNO Primary Key. A number, ranging from 0 to 1 000 000. This number identifies a particular kind of chemical.

DESCRIPTION A string of characters, from 4 to 36 characters long, which is a short description of the product. Note that more than one product may have the same description. (A given chemical may exist in more than one form, for example, it may have both a hydrated and non-hydrated form.)

DLEVEL A number, ranging from 1 to 10, indicating how hazardous this chemical is. The higher the number, the more hazardous.

ENO A number, ranging from 100000 to 999999. These are the employee numbers (ENOs) of an employee who is a specialist in the safe handling of this particular chemical and who is responsible for its safe storage and shipping. This field may be empty, in which case we put the keyword NULL there.)

CHEMNO	DESCRIPTION		DLEVEL	ENO
8543	Chromium Permanganate	5	736698	
8762	Mercury Chlorate	6	342734	
8971	Sodium Metal	8	342734	
9230	Calcium Chloride	2	354234	
9231	Calcium Chloride	2	354234	
9377	Sodium Nitrate	5	NULL	
9498	Chromium Cyanide	8	354234	
9959	Hydrogen Sulphide	3	198887	

RELATION: ORDERS
PRIMARY KEY: CUSTNUM + CHEMNO + DATE

This relation records which customer ordered which chemical, and when the order was received.

ATTRIBUTES

CUSTNUM	A whole number, ranging from 0 to 999999; Foreign key: references CUSTNUM in relation CUSTOMERS
CHEMNO	A whole number, ranging from 0 to 1 000 000; Foreign key: references CHEMNO in CHEMICALS
DATE	A date
QTY	A whole number, must be greater than 0; never above 10000

CUSTNUM	CHEMNO	DATE	QTY
765489	8543	23-03-2016	85
765489	8543	14-10-2017	85
765489	9377	14-10-2017	40
469873	9377	12-10-2017	70
434590	9498	09-02-2016	35
687744	8543	21-03-2016	40
765489	9230	21-03-2016	50
687744	8543	13-05-2016	45

RELATION: CUSTOMERS

PRIMARY KEY: CUSTNUM

This relation records details about each customer.

ATTRIBUTES

CUSTNUM	A whole number, ranging from 0 to 999999
NAME	A string of characters, from 4 to 36 characters long: the customer's business name.
LOCATION	A string of characters, from 4 to 36 characters long. The customer's location. [We will assume that there is only one location per customer.]

CUSTNUM	NAME	LOCATION
387665	AmChem	New York
469873	PestAway	Nairobi
434590	GreenGrow	Paris
687744	ConGene	New York
765489	Tata	New Delhi
987746	FanGuaiDo	Shanghai

Part C - 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. Please start this answer on a new page.

[8 hours. 20 marks]

Part D: Trying to modify the database while violating constraints

Subject guide reference: pages 63 onwards of the subject guide, *Database Systems, Volume 1*. You may also benefit from reading the Wikipedia article on Database Integrity. Be sure you know the meanings of 'Entity integrity', 'Attribute integrity' and 'Referential integrity'.

'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.)

If you are not quite sure you know the answers to some or all of the following questions, you can just try them out on the database you created in the previous question.

1. Suppose we have added a new chemical to our stock. It's called 'diphenyl hydride' and it has a danger level of 3. Employee 465755 will be in charge of ensuring its safe storage. However, we have not yet assigned it a unique CHEMNO, so we decided to record that field as NULL. What happens when you try to add it to the relation CHEMICALS?
2. Suppose we make a mistake, and assign to the new chemical the CHEMNO 9498. What happens when you try to add it to the relation CHEMICALS?
3. Suppose the database you created is being stored on a hard disc, which becomes badly fragmented. This means that large files are no longer able to be written down sequentially on the disc, but are stored in segments all over the disc, in sectors which may be 'far apart' from each other, resulting in longer retrieval time. If the disc is defragmented, which involves consolidating files so that each one is stored as a sequence of sectors, will we need to change any of the queries in the next section (Part E)? (You can run the 'DEFRAG' option on your own hard disc and see what happens.)
4. Suppose we add a new attribute (column) to the CUSTOMER file – say, a telephone number between NAME and LOCATION. Will we need to change any of the queries you will write in the next section, or will they all continue to work?
5. Suppose we delete the attribute (column) LOCATION in the CUSTOMER relation. Will we need to change any of the queries you will write in the next section, or will they all continue to work?

Part D – What to submit: Answers to questions 1 to 5. Number your answers. Please start the answers to this question on a new page.

[2 hours. 15 marks]

Part E: Querying the database

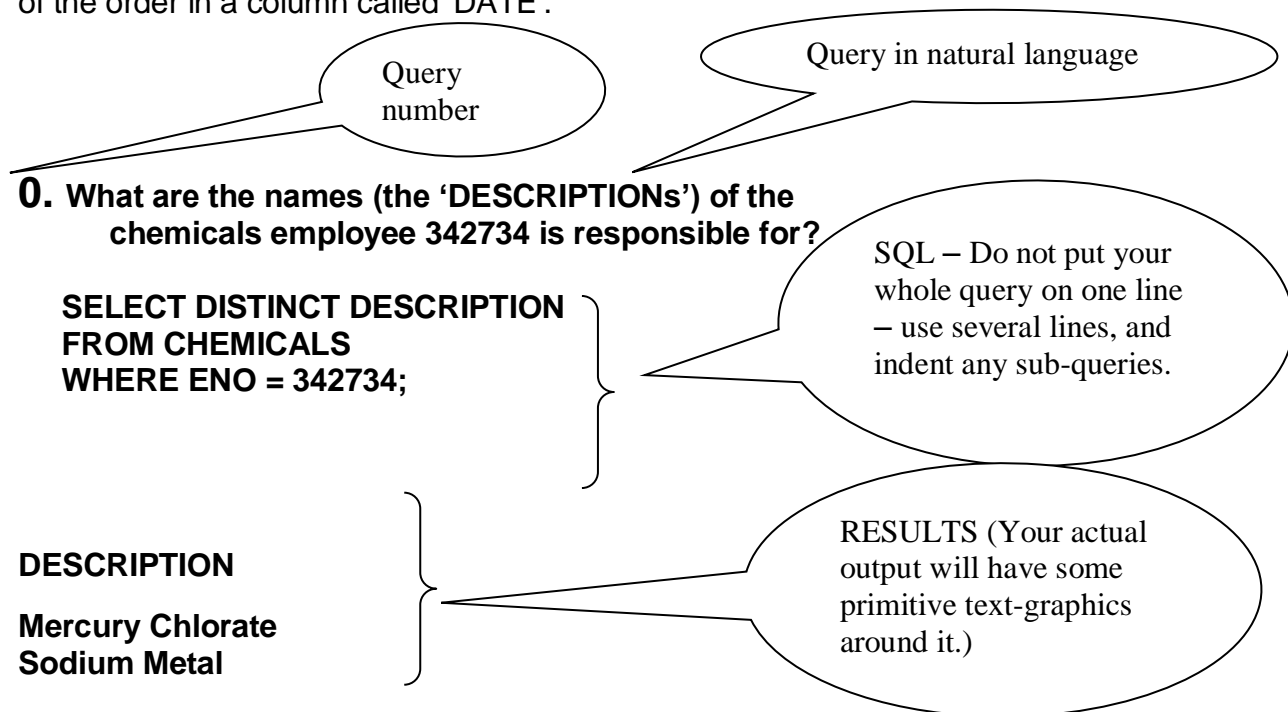
Show the SQL needed to answer the following queries, and also show the results you get when you execute each query.

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 screenshots, as they are often hard to read. Also: you

will be using this coursework 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'.



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.**

1. List full details (all attributes) of all customers who are located in New York.
2. List the CUSTNUMs of customers who placed orders on 12-10-2017.
3. List the DESCRIPTIONs of all chemicals with a DLEVEL > 4.
4. How many different chemicals are in the CHEMICALS table?
5. List all chemicals, sorted by DESCRIPTION. (Your answer should be one column only.)
6. List the CUSTNUMs of all customers who have ordered chemical 8543.
7. List the CUSTNUMs of all customers who have ordered any chemical with a DLEVEL > 4.
8. List all CHEMNOs of chemicals with 'SODIUM' in their DESCRIPTION. (Hint: read about LIKE, %)
9. List the total quantity of all chemicals ordered after 01-01-2016. (Hint: do not confuse COUNT and SUM)
10. What are the CHEMNOs of chemicals whose DLEVELs are greater than that of Sodium Nitrate? [Note: your query should still be valid even if, later, we change the DLEVEL of Sodium Nitrate.]
11. What is the DLEVEL of the most hazardous chemical(s) we stock? (The greater the hazard, the greater the DLEVEL).
12. What is/are the DESCRIPTION(s) of the chemical(s) we stock with the most hazardous DLEVEL?

13. List the CUSTNUMs and total quantities of all chemicals ordered by each customer. [Hint: read about GROUP BY]
14. List the CUSTNUMs and total quantities of all chemicals ordered by each customer in 2016.
15. List the CUSTNUMs and total quantities of all chemicals ordered by each customer where the total is greater than 75. (Hint: read about GROUP BY and HAVING.)
16. List the NAMES of the customers who have ordered chemical 8543.
17. List the CUSTNUMs and NAMES of the customers who have ordered any chemical with a DLEVEL greater than 6.
18. List the CHEMNOs and DESCRIPTIONs of any chemicals which do not have an employee assigned to assure their safe storage.
19. List the CUSTNUMs and NAMES of any customer who has ordered chemical 9377.
20. List the CUSTNUMs and NAMES of any customer who has ordered any chemical other than chemical 9377. [Note: your answer should include customers who have also ordered 9377, and those who have not, so long as they have ordered something else.]
21. List the CUSTNUMs and NAMES of any customer who has only ordered chemical 9377. (In other words, they have not ordered any other chemical.)
22. List the CUSTNUMs and NAMES of any customer who has never ordered chemical 9377.
23. List the date of the oldest order(s).
24. List the number (count) of the distinct chemicals we stock.
25. List the number (count) of the distinct chemicals which have been ordered. (For example, if every order had been for the same chemical, that count would be 1.)

Part E - What to submit:

- (1) The natural language version of the query (just a copy of the original question. questions above)
- (2) Your *SQL query* for each question, and
- (3) The *results* of running that query.

Please start your answer on a new page. Remember: *do NOT submit screenshots, or separate SQL files, for this question.*

[20 hours. 50 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. Being familiar with it will be an asset on your CV.

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 databases systems including Postgres, MySQL, Oracle, or MSSQL.*
- 4+ years working in Unix/Linux environments, particularly with web facing systems.*
- Proficient (sic) 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 8.0.13 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 for this coursework. I have found it easier to download the 'Community' version, rather than the 'Web' version. Note that the downloaded file is about 320 MB.)

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 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 somewhere that 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 also to user group mailing lists in languages other than English.)

<http://mariadb.org/en/> – a ‘drop-in’ equivalent of MySQL started by people concerned about MySQL's public status after it was bought by Oracle Corporation. You can read about MariaDB here: <http://en.wikipedia.org/wiki/MariaDB> and here: <http://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://www.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 a 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) **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 '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, *etc.*)

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 us to predict the other two. Nor does knowing any pair of values allow us to predict the other one. We 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 as numerous as the things language itself can refer to: people, places, smells, colours, quantities of money, rates of inflation, dates, *etc.* 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, mutability – can also apply to data, which is just '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 it 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 a key feature 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 do not permit salaries of zero or less, we do not have employees born in the 19th-century, *etc.* 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' (for example, no DATES earlier than 1900/01/01) 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. There are workarounds to this serious limitation 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 measurement are all correct – each one has been made with an 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, we would be precise but not very accurate; if we said the board in the previous example was between 40 and 60 cm long, we would be accurate but not very precise.

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) and which is 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 – for example, one using pure integers, 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 our database held only geographic addresses within the USA, for instance, we 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 **CHEMICALS** in **Part C**. You may have wondered why we needed both **CHEMNOs** and **DESCRIPTIONs** to identify a particular chemical. Why not just use the **DESCRIPTION** alone, especially since it's probably how the people who will be buying the chemicals 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 variation of a chemical 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' vs 'St Petersburg'), *etc.* 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 which 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, I and 1, *etc.*), to avoid data entry/transcription errors, and assuming our language uses the Roman alphabet. Look up the Wikipedia article on the Canadian Postal Code system for an example of a thoughtful design which takes into account likely human errors in writing down postal codes.

Error detection: We might want to embed error-detection into our artificial names via a 'Check Digit' (as is done with credit card numbers and ISBN numbers that identify book titles). If someone makes a typical mistake when reproducing these numbers, such as omitting a digit, transposing digits, or even typing a wrong digit in place of a correct one, this error can be detected.

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 also are aware that in the 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. In the same way, 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 later 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.

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. Therefore, 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 should choose, where possible, integer data types over character, and fixed-width character over varying width character. (It is important to 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 'P1 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 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 looking at. However, in designing a real database, if efficiency is a major consideration, we may sacrifice 'usability' in favour of processing speed.

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. for example UNIX-based systems), or processed by certain programming languages (for example C++, Java, Python). Be aware that some systems are case-sensitive and others (for example Windows) are not, but that 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 different 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.**

In addition, sometimes students will say, 'This relation has no Primary Key'. However, since by definition a relation can have no duplicated tuples, then even if no Primary Key is declared, the whole tuple serves as a Primary Key. However, the database software **must** implement this definition. In MySQL, relations that are derived as the result of a query can have duplicate tuples, and are thus not really relations, unless you use the key word DISTINCT after SELECT.

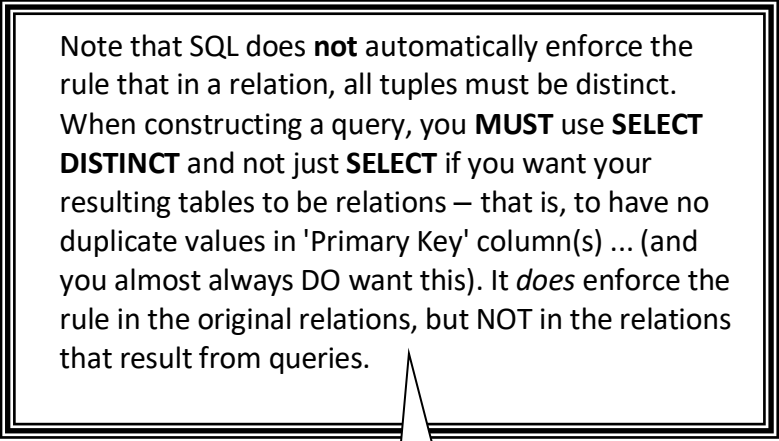
The 'autoincrement' temptation: Every database management system, including MySQL, has a method that allows someone creating a database to have the system automatically insert a sequence of numeric values into a column. As such, you can design a table so that, say, the first column has a 1 in the first tuple, a 2 in the second tuple, etc. 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 is 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 usually a mistake to do this. Consider one of the Master Relations in this coursework assignment – say, the ORDERS 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. But 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, but 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 section headed 'Embedded information'.

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, as well as giving us more information than a sequential number would.

The important lesson here is to not 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 the rule that in a relation, all tuples must be distinct. When constructing a query, 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 should be 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:

The word 'Normalization' can be confusing: Unfortunately we are stuck with this word. Beginners are sometimes confused by it, 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 that 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 Normal 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

<u>EMPNUM</u>	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 is a very bad idea to do so. It will make your searches, insertions and deletions, on **PHONE-NUMBERS** very complicated, as you will 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 will not be able to do a *COUNT* on the attribute, or a *JOIN*.

If that argument does not 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 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 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. First, it will be awkward to add a fifth phone number; it is awkward to query. And second, 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 land-line 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

<u>EMPNUM</u>	<u>NAME</u>	<u>PHONE</u>
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 has a fatal weakness when it comes to doing 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 unnecessarily complicated – is to have **two** relations:

Design 4

EMPLOYEES

<u>EMPNUM</u>	<u>NAME</u>
E123	Don S.
E234	Susan W.
E345	Don P.
E567	Fatima M.

EMPLOYEE-PHONE

<u>EMPNUM</u>	<u>PHONE</u>
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 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 numbers and names, including those of employees without phones, and the second to hold the information about employees with phones.

Each tuple in the two relations 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 phones does not appear in the second relation. Because an employee can have but one name, we can store the name in the 'existence list'. 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.

Note: many students think that the possible 'anomalies' associated with unnormalized relations are just 'errors'. This is not the case. The term 'anomaly' has a distinct meaning, and there are three kinds. Be sure you know them for the examination.

[END OF APPENDICES]