University of London International Programmes
CO2209 Database systems
Coursework assignment 1 2015–2016

Important

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

FamilyName_SRN_COxxxxcw#.pdf (e.g. Zuckerberg_920000000_CO3323cw2.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).

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

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

The best way to approach this coursework assignment is to look over the whole thing first, and get an idea of what you will want to concentrate on as you read the *subject guide* and other materials such as your textbook. **Part B** covers some of the fundamental ideas of database theory and most of it can be done independently of **Part A**, so you might wish to do those parts of **B** right away. 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. Thus, 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, along with Coursework assignment 2 (which forms the other part of your assessment), will introduce you to the basic concepts of the most common data model (the relational model) around which most databases are constructed; will give you some practical experience in designing, implementing and using a database management system; and 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, you ought to be able to give a good account of yourself at a job interview which touches on the subject of databases, 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 get experience with real databases, 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 to keep educating yourself about developments in it long after you have finished this course. This coursework assignment aims to help you do this.

Suggested sources

This coursework assignment is 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**, I 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.

A note on Wikipedia

Wikipedia is the place where most people begin their online searches, and rightly so. This coursework assignment will direct you to Wikipedia articles frequently. 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.) Because Wikipedia articles can be written by anyone, and can have many authors, who are usually anonymous, you cannot treat Wikipedia the way you treat ordinary references. Remember that everything you read there was written by someone, and perhaps by someone whose knowledge was only partial, or who was overly partisan, and/or who had a material interest in convincing readers of a particular point of view. And a Wikipedia article can be changed at any time.

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 [https://en.wikipedia.org/wiki/Comparison_of_MySQL_database_engines] – at least, the article online in the summer of 2012 – you will see that it clearly has been written by someone who is an InnoDB enthusiast. It would be very dangerous to make a decision about the relative merits of these two alternative database engines based solely on that article, which is highly partisan.

Another example: the Wikipedia article on Data Integrity [https://en.wikipedia.org/wiki/Data_integrity] was written by someone with only a middling grasp of English. The contents are useful, but you would definitely not want to quote what is written here in a professional report because of its poor grammar.

Therefore, you should never rely **only** on Wikipedia as a source of information; when you use Wikipedia, 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. 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 in Wikipedia, or anywhere else online, ask about it via the course forum.

Coursework assignments and the examination

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 assignments, which consist essentially of having you pay close, systematic attention to the *subject guides* and make 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. But note, your revision should start in November or December at the latest, not in April.

ASSSIGNMENTS

This coursework assignment has two parts: **Section A** is 'specific' (to a particular system) and 'practical'; and **Section B** is 'general' and 'theoretical' (dealing with relational database theory in general and thus applicable to whatever system you encounter after this course).

SECTION A

Section A of this coursework assignment has five tasks: downloading and setting up the software for managing a database; and then implementing a 'toy' database with it; describing this database graphically; finding out how to get help from experienced database users; and running some queries on the database you have implemented.

A1 Setting up a DBMS

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

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

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

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

Note: most people have no trouble downloading and installing the MySQL package. But …problems can occur. If they do, and you cannot solve these quickly by yourself, you MUST come to the online forum for this course right away and get help there.

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 started by people concerned about MySQL's public status after it was bought by Oracle Corporation, which, they fear, may eventually let it wither on the vine. You can read about MariaDB here:
http://en.wikipedia.org/wiki/MariaDB

What to submit: write a short report describing any problems you had in downloading and installing MySQL. If you did not have any problems, just write a description of how you did it. Pretend that you are writing instructions for someone who is a novice at downloading and installing software. In both cases, indicate how much space the package takes on your hard disc.

If you already had MySQL installed on your computer, write a short (one page) summary of the contents of the MySQL Manual. You can do this by summarising (but not just copying) the most important items in the Table of Contents.

NOTE: re-read the three previous paragraphs above. A 'report' that says 'I downloaded and installed it without any problems' is not a report and will not gain any marks.

[2 hours, 5 marks]

A2 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, and 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, and have a basic familiarity with the following words, which apply to relational databases in general: **Table** (or **Relation**), **Column** (or **Attribute**), **Primary Key**, **Candidate Key, Foreign Key**, **Domain, Data Type**, and **Constraint.**

While doing this coursework assignment, you should consult pages 59–84 of the **Database systems** *subject guide*, Volume 1, which covers SQL.

The background

A government bureau, which manages a National Park and Wildlife Refuge, wants you to implement a database recording certain information.

It wants you to record information about its **Park Rangers,** who are assigned to look after specific areas in the park; about the **Park Areas** they look after; and about which **Park Ranger** is working in which **Park Area.** (In the language of Entities and Relationships, we have two Entity Types here – **Park Ranger** and **Park Area** – and a relationship – Looks After – between them.)

Every Area in the park is being sponsored by a private entity, called a **Sponsor** – these may be companies, religious groups, schools, or voluntary organisations. These **Sponsors** take part in voluntary 'work days' doing trail maintenance, replanting, rubbish collection, fund-raising, and other activities in support of the Park.

For each **Park Area**, the government bureau wants you to record the name of the private entity which sponsors it (that is, the **Sponsor**), the start date on which the sponsor began their sponsorship and the date of their next group visit to the park to take part in a 'work day'. They also want you to record its size in hectares, in the attribute **AREA**. **Park Areas** are identified by **GEOCODE**s. Each Park Area also has a unique **PNAME**, which is a kind of 'nickname' for the Park Area.

About each **Ranger**, it wants you to record their (unique) Badge Number (**BADGENUM**), their name, their date of birth, and the woodcraft skills which each ranger is a qualified expert in.

It also wants you to record which **Rangers** are looking after which **Park Area**. (A Ranger can exist without looking after a Park Area, and a Park Area may, temporarily, not have any Rangers assigned to it. A Ranger may look after more than one Park Area, and some Park Areas may have more than one Ranger assigned to look after them.)

If we were to draw a simple E/R diagram for two of the entity types for this database, it would look something like this.

```
 _____________                                       ____________
|              |                                    |             |
|  RANGER      |--0:N-----<looks-after>------0: - M  PARK AREA      |
| ____________ |                                    |___________ |
```

The meaning of this Entity/Relationship diagram is that a given RANGER can look after zero or more Park Areas – in other words, a RANGER can exist without being assigned to a Park Area – and that a Park Area can exist without a RANGER being assigned to it at the moment, and that several RANGERs can be assigned to the same Park Area. (The "0:N" and "0:M" in the diagram tells us the minimum and maximum – if there is an "N" it means there is no maximum number of 'entity-instances' that can take part in this particular relationship. (We use two letters, 'N' and 'M', so that there is no false impression given that these numbers must equal each other.) NOTE WELL: the relevant "minimum:maximum" for a particular entity type is on the other side of the relationship name; this is called 'look-across' notation.)

The relational schema has been designed for you. All you have to do is to implement it.
Conventions: Each relation (table) has a name, and under it, a list of attributes (columns). The attribute of attributes that make up the Primary Key are underscored. Thus the first relation is called **PARK AREA**, and has five attributes (**GEOCODE, PNAME, AREA, SPONSORNAME, STARTDATE** and **NEXTEVENT**); **GEOCODE** is the Primary Key – this means that there must not be more than one tuple (row) with the same **GEOCODE**. Both **GEOCODE** and **PNAME** are Candidate Keys – either could serve to identify a unique tuple (row). We have chosen **GEOCODE** for the Primary Key because it is fixed-width and thus will be easier to index, and will make *JOIN* queries faster. (If you are new to databases, a lot of these terms will not mean anything to you. Don't worry. We will learn as we go along.)

We do not show the Domain of each Attribute, because Domains, although part of relational theory, are not directly implemented in most database systems. Instead we will need to know the 'data type' of each attribute. Implement **GEOCODE** as VAR(4), **SPONSORNAME** as VARCHAR(32), and the two dates as DATE.

Make your own choices for the data types of the other relations. You do not have to implement 'foreign keys' in this coursework assignment. Except for making the appropriate attribute or attributes for the Primary Key, you do not have to implement any other constraints in these relations.

Create a database in SQL consisting of the tables (and their data) shown below.

(See the **Database systems** *subject guide, Volume 1*, pages 64 and 65, and/or the relevant pages of the MySQL manual dealing with the *CREATE* command.)

In the tables below, the Primary Key is underlined. All codes (for Park Areas and Ranger Badge Numbers, are exactly four characters long. **PNAME**s can be up to 16 characters long. **SPONSORNAME**s can be up to 32 characters long. In the relation **PARKAREA**, both **GEOCODE**s and **PNAME**s are unique. (A **PNAME** cannot be used for more than one Park Area.) The Primary Key for each table has been underlined.

PARKAREA

<u>GEOCODE</u>	PNAME	AREA	SPONSORNAME	STARTDATE	NEXTEVENT
K003	South Meadow	45.6	Newton School	2011-12-01	2015-12-30
K002	Bear Caves	67.0	Ibn Khaldun Club	2009-05-15	2016-01-29
K010	Bird Lake	12.3	Rotary Club	2013-11-21	2016-04-24
K009	Old Forest	95.8	Newton School	2015-03-01	2016-04-24
K011	Rocky Hill	23.3	Chang's Autos	2012-12-01	2016-02-11
K005	Piney Woods	65.5	Friends of Nature	2012-03-01	2016-03-05

RANGER-MASTER

<u>BADGENUM</u>	SURNAME	FIRSTNAME	DOB
B007	Singh	Simon	1991-11-27
B006	Hardy	Lucy	1982-03-12
B012	Sen	May	1989-09-01
B023	Shariff	Athar	1984-08-23
B020	Rabin	Rebecca	1986-07-09
B045	Kutuzov	Vasilli	1983-10-25

RANGER-EXPERT

<u>BADGENUM</u>	<u>WOODCRAFT SKILL</u>
B007	First Aid
B007	Trailcraft
B007	Birdcalls
B007	Veterinary
B006	First Aid
B006	Veterinary
B006	Firemaking
B023	Trailcraft
B023	Wild Food Finding
B045	Veterinary

RANGER-PARK AREA

<u>BADGENUM</u>	<u>GEOCODE</u>	DATE-ASSIGNED
B007	K003	2011-12-06
B007	K010	2010-01-23
B020	K003	2004-11-22
B023	K003	2013-04-15
B023	K005	2012-04-04
B012	K010	2012-12-21
B012	K005	2014-03-01
B012	K002	2015-10-20

What to submit: submit a copy of the SQL statements you used to create your database. Also submit a listing of the tables you create.

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.

[5 hours, 10 marks]

A3 Entity/Relationship diagrams

(1)	Copy the E/R diagram above, and add Entity Types for Sponsor and Woodcraft Skill.
(2)	Redraw the diagram, assuming that a Ranger can only look after one Area, and that an Area can have only one Sponsor.

[1/2 hour, 2 marks]

A4 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 one or more suggestions by others as to how the problem could be dealt with.

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

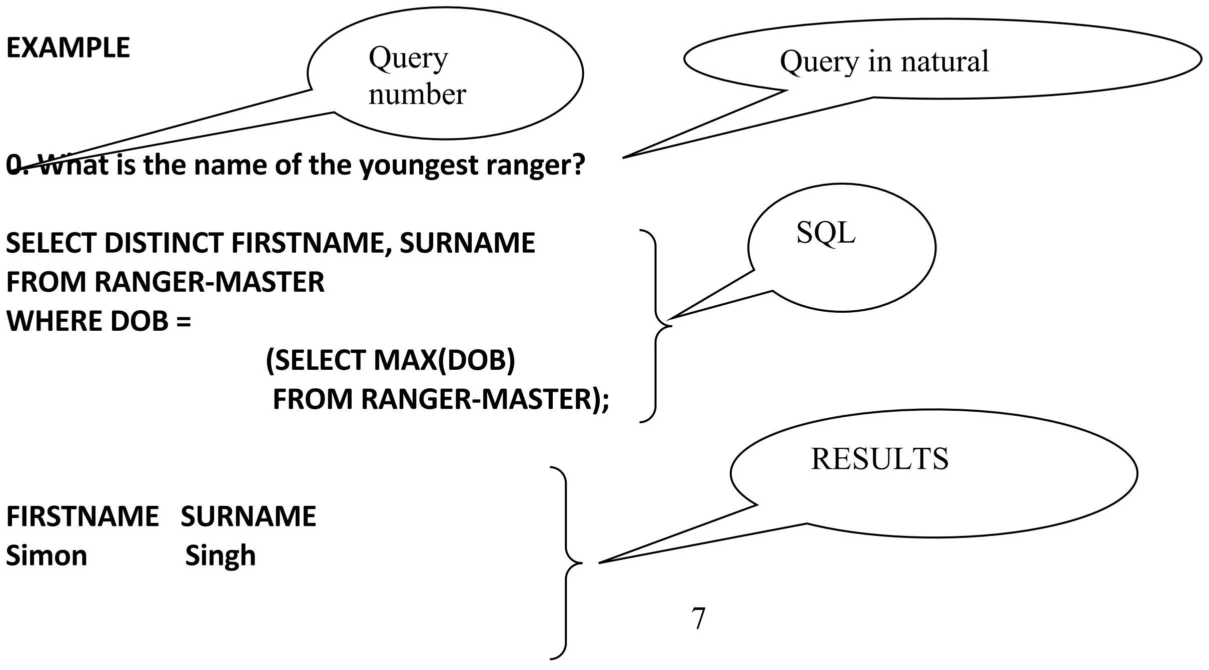
[1 hour, 3 marks]

A5 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 which results. Again, use the **TEE** command to record your work. Please do **not** use screen shots, as they are often hard to read. Also: you will be using this coursework assignment to revise for the examination, and this format will make it easier for you to revise the SQL.

EXAMPLE



0. What is the name of the youngest ranger?

```
SELECT DISTINCT FIRSTNAME, SURNAME
FROM RANGER-MASTER
WHERE DOB =
                    (SELECT MAX(DOB)
                     FROM RANGER-MASTER);
```

```
FIRSTNAME   SURNAME
Simon        Singh
```

(01) List the first names and surnames of all our RANGERs**.**
(02) List the first names and surnames of all our RANGERs, ordered on their surnames, in ascending order.
(03) What are the Badge Numbers of RANGERs who were born before 1987?
(04) What are the Badge number(s) of the oldest RANGER(s)? (Note that there can be a 'tie', so that there is no single 'oldest' RANGER.)
(05) How many RANGERs are there? (Note: do not just count them by hand! We want the SQL that will yield this answer, and that will still be valid if we run it again next year if we hire new Rangers; and the answer will NOT be a list of RANGERs, but a single number.)
(06) What is the name of the Park Area which has been sponsored the longest?
(07) What are the Badge Numbers of RANGERs who look after a Park Area sponsored by Newton School?
(08) What are the names of RANGERs who look after a Park Area sponsored by the Rotary Club or Chang's Autos?
(09) What are the names of Park Areas that Ranger Lucy Hardy looks after?
(10) What are the names of the Sponsors whose Park Areas Lucy Hardy looks after?
(11) What are the names (first and surname) of RANGERs who are experts in Trailcraft?
(12) What are the BADGE numbers of RANGERs who look after two or more Park Areas?
(13) What are the BADGE numbers of the RANGER(s) who have worked on the most Park Areas?
(14) What is the average number of Park Areas our RANGERs looks after? (Note: not each Ranger, but the group as a whole.)
(15) What are the Badge Numbers of those RANGERs who are experts in both Veterinary **and** First Aid?
(16) What are the Badge Numbers of those RANGERs who are experts in Veterinary **or** First Aid?
(17) What are the Badge Numbers of those RANGERs who are not experts in Veterinary?
(18) What are the Badge Numbers of those RANGERs, if any, who have **not** been assigned to a Park Area and who know Veterinary?
(19) We want to add a new RANGER, whose number will be E787, whose name is Edward Teller, and who was born on the 8th of August, 1989, to the database. What command can we use to do that?
(20) What is wrong with the following query?
SELECT SURNAME
***FROM* RANGER-MASTER**
***WHERE* DOB = NULL;**
(21) How many hectares in total is the park?
(22) What groups, if any, have their next working-day in the park on the same date?
(23) What are the names of the Rangers with no skills?
(24) What skill(s) [there may be a tie] are most commonly held?
(25) If the Rotary Club ceased its sponsorship of Bird Lake, what command would we use to change the database to reflect this fact?

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; and
(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? Because it will help you when you use this coursework assignment for revision.)
Be sure to number your answers, using the numbers shown above. Remember: do not submit screenshots for this question.

[15 hours, 50 marks]

SECTION B

B1 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.
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 BADGENUM and PNAME as identifiers for rangers and park-areas as described above, how would we show the following are possible:

A Ranger can look after only one ParkArea. A ParkArea can be looked after by only one Ranger.
A Ranger can look after only one ParkArea. A ParkArea can be looked after by more than one Ranger.
A Ranger can look after more than one ParkArea. A ParkArea can be looked after by only one Ranger.
A Ranger can look after more than one ParkArea. A ParkArea can be looked after by more than one Ranger.

[1/2 hour, 4 marks]

B2 Relational concepts

Write brief definitions to show what we mean by the following terms, as they apply to relational databases. **For each definition, refer to the relevant feature(s) or data values of the database you implemented in Section A to illustrate it**. (In other words, give an example from your database. No marks will be given for answers that do not do this.)

(a) Attribute
(b) Tuple
(c) Degree
(d) Cardinality
(e) Candidate key
(f) Primary key
(g) Alternate key
(h) Foreign key
(i) Functional dependency
(j) Determinant
(k) Compound (or composite) key
(l) NULL value

[See **Database systems** *subject guide*, *Volume 1*, page 31, and pages 49–52]

[1 hour, 6 marks]

B3 Relational design

In your work in Part A, you were given the following relation.

PARKAREA

GEOCODE	PNAME	AREA	SPONSORNAME	STARTDATE	NEXTEVENT
K003	South Meadow	45.6	Newton School	2011-12-01	2015-12-30
K002	Bear Caves	67.0	Ibn Khaldun Club	2009-05-15	2016-01-29
K010	Bird Lake	12.3	Rotary Club	2013-11-21	2016-04-24
K009	Old Forest	95.8	Newton School	2015-03-01	2016-04-24
K011	Rocky Hill	23.3	Chang's Autos	2012-12-01	2016-02-11
K005	Piney Woods	65.5	Friends of Nature	2012-03-01	2016-03-05

Suppose the Park Administration decided that it wanted to allow more than one Sponsor to sponsor a given area.

(1) Would they need to redesign the database? If so, propose a new relational schema that can allow more than one sponsor to sponsor an area. (Note that this may require the creation of new relations.)

(2) Suppose they also decided not to have more than one sponsoring group working in a given area on the same date. How could a re-design of the database allow this?

(HINT: think about the purpose of the Primary Key.)

What to submit: relational schemas that will meet the new requirements for (a), and then for (b). Your schemas need not show any relations that do not have to be changed.

[4 hours, 10 marks]

B4 'Big Data'

Write a brief (two-page) answer to the question on 'Big Data' and whether traditional relational databases can adequately meet its challenge.

Your essay should address such questions as: What is meant by 'Big Data'? How are 'Big Data' situations different from those that characterised traditional Very Large Databases? What alternatives to the relational model and SQL now exist to address the 'Big Data' situation? (You can find information that will allow you to answer this question on the internet.)

Cut-and-paste submissions will receive no credit.

[4 hours, 10 marks]

[TOTAL 100 marks]

Appendix I: About MySQL

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

Here is an excerpt from a job advertisement (August 2012). 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 favorites, 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 5.6 version for whatever Operating System you have. If you already have an earlier version of MySQL, it is not necessary to download a later version.)

You can read about MySQL here: [https://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 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 do worse than print the relevant 'cards' out, using a colour printer, and posting them somewhere 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 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, which some people fear will eventually let it wither on the vine. It 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: [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

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 'frozen language'. For instance, suppose we have an attribute called 'COLOUR' in a relation, and want to record the data value 'blue' in some of the tuples. But 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 is 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: 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: I can be very precise, but inaccurate (if I said that the board in the previous sentence was 45.14297 cm long), accurate but not very precise (if I 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, I can choose between several datatypes that will do the job. As a rule, I 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 I intend to use it in (will I compute with it, in which case I should use a numeric datatype, or just display it, in which case I should use a character datatype?).

Appendix III: Names and keys

See the **Database systems** *subject guide*, pages 50–52.

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 **ParkArea** in **Part A**. You may have wondered why we needed both **GEOCODE**s *and* **PNAME**s to identify a particular area in the park. Why not just use the **PNAME** alone, especially since it's probably how the project members themselves refer to the project?

The answer is, we often give things 'artificial names' (sometimes called 'surrogates') to identify them in database work because their 'natural names' are inadequate for our purpose. More than one person can have the same name, a person (or company) can change its name (through marriage, or merger), chemicals can be spelled with slight variations ('sulfur' vs 'sulphur'), the names of cities can have several spelling variants or even be different ('Leningrad' vs '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 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 which 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 which 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. So, '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.)

But…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: think twice before making an attribute serve both as an identifier, and as an information-bearer. Unless there are strong considerations otherwise, an attribute which is designed to uniquely identify an instance of an entity type should not do double-duty as an information-bearing attribute also.

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 (ie. Not VARCHAR, or, at least that's the conventional wisdom), and should choose, where possible, integer data types 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 digits only, which would allow us to use a numeric type, so that it is immediately obvious to the reader what kind of identifiers they are. But in designing a real database, efficiency is the major consideration.

Case sensitivity: Another consideration: if 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), be aware that some systems are case-sensitive and others (e.g. Windows) are not, but 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) . And 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 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 almost always DO this).

Appendix IV: 'Normalizing' relations

The goal of relational design is to end up with a set of 'normalized' relations. (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, Volume 1,* pages 107–124, where there is an excellent 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: we are stuck with this word, unfortunately. 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're in Second Normal Form you are also in First Normal Form. And so on.

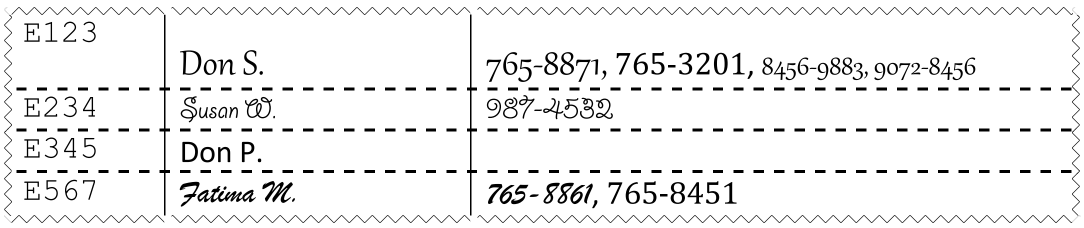
Higher Normal forms: the Fourth and Fifth Normal Forms are sometimes called the 'higher' normal forms. Note that by the time you have normalized a set of relations to Boyce-Codd Normal Form (BCNF, sometimes called 'three-and-half Formal Form'), you almost certainly have a set of relations which are also 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 I want to record the phone numbers of employees. (An employee can have no phone, one, or more than one telephone number.)

If I were recording this information on a piece of paper, I 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's 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'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? Heck, 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 madness.

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). But there are two kinds of objections to this design: It will be awkward to add a sixth phone number; it is awkward to query.

An alternative design, which 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 has a fatal weakness when it comes to doing 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. (And 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. The second holds 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 – that is, 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 --

[END OF COURSEWORK ASSIGNMENT 1]