The University of Manchester

School of Computer Science

COMP23111: Fundamentals of Databases

Learning Activities Manual

Alvaro A A Fernandes

✉ a.fernandes@manchester.ac.uk

December 2016 (V1.14::V1.15)

Contents

List of Figures

Introduction

The teaching activities in COMP23111: Fundamentals of Databases are restricted to lectures, the plan for which is shown in Figure 1, later in this document.

This document focusses on the detail of the various learning activities that make up the bulk of your (and the course unit staff's) effort in COMP23111.

Kinds of Learning Activity There are three kinds of learning activity:

S, R **self-study** tasks and **reading** assignments that you are expected to carry out on your own but on which you can seek feedback from the course unit staff if you need;

EC timetabled **examples classes**, where you work, possibly as a group, on tasks that are applications of concepts and techniques taught in the course unit (and interact with staff to get feedback) but to which no submission, and hence no formal assessment, is associated;

EX timetabled **lab sessions**, where you can seek face-to-face feedback on any learning activity, but, in particular, on the set exercises to which a submission, and hence formal assessment, is associated.

Types of Feedback and Types of Assessment In the case of self-study tasks, you carry out the learning activity in your own time[1]. Most are weekly mandatory reading tasks, but some of them are very important practice and familiarization tasks with techniques or tools that are required for other activities. For self-study tasks, the feedback is *responsive*, i.e., you should use the next examples class or the next lab session to seek feedback from the staff. There is no formal assessment of self-study tasks but subsequent learning activities assume you have done them and learned from them. So, don't ignore them or you will likely struggle with examples classes and lab sessions (and hence with the final exam).

In the case of examples classes, as usual, you attend the class as specified in your timetable, your attendance is taken, you carry out the tasks set for you and you are given *interactive* feedback, i.e., the staff will interact with you there and then to provide feedback as you carry out the set tasks. There is no formal assessment of the work you do in examples classes but subsequent learning activities (and, in particular, the exam you will sit) assume you have done them and learned from them. This course unit uses examples classes to provide you with an opportunity to deepen your understanding of concepts and techniques that are examinable but are hard to turn into lab material. Although they don't contribute marks for coursework, examples classes are crucial for you to do well in the exam. So, once again, don't ignore them.

Finally, in the case of lab sessions, as usual, you attend the session as specified in your timetable and your attendance is taken. Note that you are expected to have started your work the exercises in your own time, outside of, and prior to, lab sessions. You can get *interactive* feedback, i.e., the staff will interact with you there and then, in the lab session.

Submission and Assessment of Coursework For every exercise, there is coursework to be submitted. Submission of work is done through Blackboard, which is then assessed off-line, by the course unit staff, and **not** face-to-face, as you're perhaps more used to.

Some submissions are the subject of *formative* assessment, others are the subject of *summative* assessment. The difference between formative and summative assessment is that the former does not contribute

[1] But, note, all learning activities have pre-/co-requisites (lecture attendance, readings and practice) and deadlines. This is because there are, of course, dependencies, i.e., you must learn a concept or technique A before you can learn another concept or technique B.

to your coursework mark for the course unit whereas the latter does contribute. The assessment of your submission (both in the formative and in the summative case) comes alongside written feedback that is accessible through Blackboard. Further feedback (e.g., clarification) can be had in responsive mode, i.e., you can use the next examples class or the next lab session to seek interactive feedback from the staff on the written assessment you have received through Blackboard.

Policy on Submission and Extensions In this course unit, the policy on submission of coursework for summative assessment and on penalties for late submission of such coursework is as the University stipulates. You should familiarize yourself with the Policy on Submission of Work for Summative Assessment on Taught Programmes →Download.

In this course unit, the policy on extensions to deadlines on coursework that is subject to summative assessment is as the University stipulates, i.e., there are no extensions available other than on grounds that characterize mitigating circumstances. You should familiarize yourself with the Policy on Mitigating Circumstances →Download.

If there are mitigating circumstances that have affected your ability to meet a deadline, you should contact the School's Student Support Office (SSO). This is their Mitigating Circumstances webpage →Read online.

The next few tables in this document contain crucial information about the course unit, including submission deadlines.

Plans for Learning Activities Plans The table in Figure 1 tells you what the **lecture plan** is, on a week-by-week basis. Other than for Week 01, when there are two, this course unit only has one lecture per week.

The reason you have fewer lectures than usual is because you are correspondingly given a heavier than usual mandatory **reading assignments** as well as three important **self-study tasks** you must do in Weeks 01 and 02.

So, there is a lot to learn, and therefore work to be done, from the very start. Don't delay engaging with the course unit or you may find yourself struggling surprisingly soon.

The table in Figure 2 describes what tutorials you must do, what material you must read and when. There are links to material that is either available here in this document or else is readable or downloadable online. You should also have a look at the references section at the end of this document: it contains links to the University Library catalogue so that you can see the up-to-date availability of physical copies of books that the course unit refers you to.

The table in Figure 3 describes the timetabled activities, i.e., **examples classes** and **lab sessions** planned for this course unit. Note the *Pre-/Co-Requisites* column: it tells you that you need to keep up with readings and lectures in order to make the most of your learning opportunities. This is an intense course unit, so try not fall behind and if you do, seek help from the staff. Figure 5 is meant to help you check whether you're up to date with your learning or, alas, falling behind. More details about the timetabled activities (including assessment type and deadlines) are given in Figure 4.

The table in Figure 4 complements the information in Figure 3. For each timetabled activity, it tells you the feedback type that you can expect, whether the activity is the subject of formative or summative assessment, and what deadline is associated with it.

Finally, set aside sometime to study Figure 5: it is a dependency graph for the course unit, therefore it tells you on a week-by-week basis how lectures, readings, examples classes and lab sessions are related to one another. An arrow from a node N to a node N' means that N' depends on N, where *depends* can

Week	Lecture	Topic
W01	L01	Introduction to Data Management
"	L02	The Entity-Relationship Model
W02	L03	The Enhanced Entity-Relationship Model
W03	L04	The Relational Model
W04	L05	Relational Languages
W05	L06	Conceptual to Logical Mapping
W06	—	(Reading Week)
W07	L07	Functional Dependencies
W08	L08	Normalization
W09	L09	Advanced SQL: Stored Procedures and Triggers
W10	L10	An Overview of Transaction Processing
W11	L11	File Organization and Indexing
W12	L12	DBMS Architectures

Figure 1: Lecture Plan

Week	Activity	Material	Download	Read Online	Deadline	Timing
W01	RA1	Appendix A	(this document)	(this document)	W02	start of EC01
"	RA2	Appendix B	(this document)	(this document)	W03	start of EX01
"	RA3	Appendix C	(this document)	(this document)	W05	start of EX02
"	R01	[SKS06], pp. 1-35	→Download	(not available)	W01	start of L01
"	R02	[EN13], pp. 201-245	→Download	→Read online	W02	start of L02
W02	R03	[EN13], pp. 246-286	(not available)	→Read online	W03	start of L03
W03	R04	[GUW14], pp. 13-62	→Download	(not available)	W04	start of L04
W04	R05	[Mor+13], pp. 1-24	→Download	(not available)	W06	start of L05
W05	R06	[EN13], pp. 287-308	(not available)	→Read online	W07	start of L06
W06	R07	[BS05]	→Download	(not available)	W06	Friday
"	R08	[Eur12]	→Download	(not available)	W06	Friday
W07	R09	[RG03], pp. 605-648	→Download	(not available)	W08	start of L07
W08	—	—	—	—	—	—
W09	R10	[CB15], pp. 271-290	→Download	→Read online	W10	start of L09
W10	R11	[KBL05], pp. 455-486	→Download	(not available)	W11	start of L10
W11	R12	[CB15], pp. F1-F24	→Download	(not available)	W12	start of L11
W12	R13	[SKS06], pp. 769-795	→Download	(not available)	W12	start of L12

Figure 2: Self-Study Learning Activities

Week	Activity	Topic	Pre-/Co-Requisites
W01	—-	—	—
W02	EC01	Eliciting Data Requirements	RA1, R01, R02, R03, L03, L02, L01
W03	EX01	Mapping Data Requirements to an EER Model	RA2, EC01
W04	EC02	Writing Relational-Algebraic Expressions	R04, R05, L05, L04
W05	EX02	Mapping an EER Model to a Relational Model	RA3, EX01, EC02, R06, L06
W06	—-	(Reading Week)	—
W07	EC03	Reasoning about Normal Forms and Dependencies	R09, L07
W08	EX03	Writing SQL Queries	EX02
W09	EC04	Normalizing a Relational Logical Model	L08
W10	EX04	Writing SQL Views and Triggers	EX03, R10, L09
W11	EC05	Reasoning with Transactions	R11, L10
W12	EX05	Writing SQL Stored Routines	EX04

Figure 3: Timetabled Learning Activities: Topic

Week	Activity	Feedback Type	Assessment Type	Deadline	Timing
W01	—-	—	—	—	—
W02	EC01	Interactive	N/A	W02	end of class
W03	EX01	Interactive/Responsive	Summative	W04	Friday, 5 pm.
W04	EC02	Interactive	N/A	W04	end of class
W05	EX02	Interactive/Responsive	Formative	W07	Friday, 5 pm.
W06	—-	—	—	—	—
W07	EC03	Interactive	N/A	W07	end of class
W08	EX03	Interactive/Responsive	Summative	W09	Friday, 5 pm.
W09	EC04	Interactive	N/A	W09	end of class
W10	EX04	Interactive/Responsive	Summative	W11	Friday, 5 pm.
W11	EC05	Interactive	N/A	W11	end of class
W12	EX05	Interactive/Responsive	Formative	W12	Friday, 5 pm.

Figure 4: Timetabled Learning Activities: Other Details

be taken to mean that you need to have studied/attended/done the source activity in order to be able to fully understand the destination activity.

The rest of this document is organized as follows. There is one section per timetabled activity in the form, broadly, of a task sheet. These need to be read in the context of Figures 1 to 5. There follow a few very important appendices: one is a discussion of how data requirement specifications can be derived from data flow diagrams; the next two are tutorials for you to take in Weeks 01-02, one on SQL*Plus and one on Dia; the final one is on plagiarism and on caution you must exercise when working together with friends and colleagues.

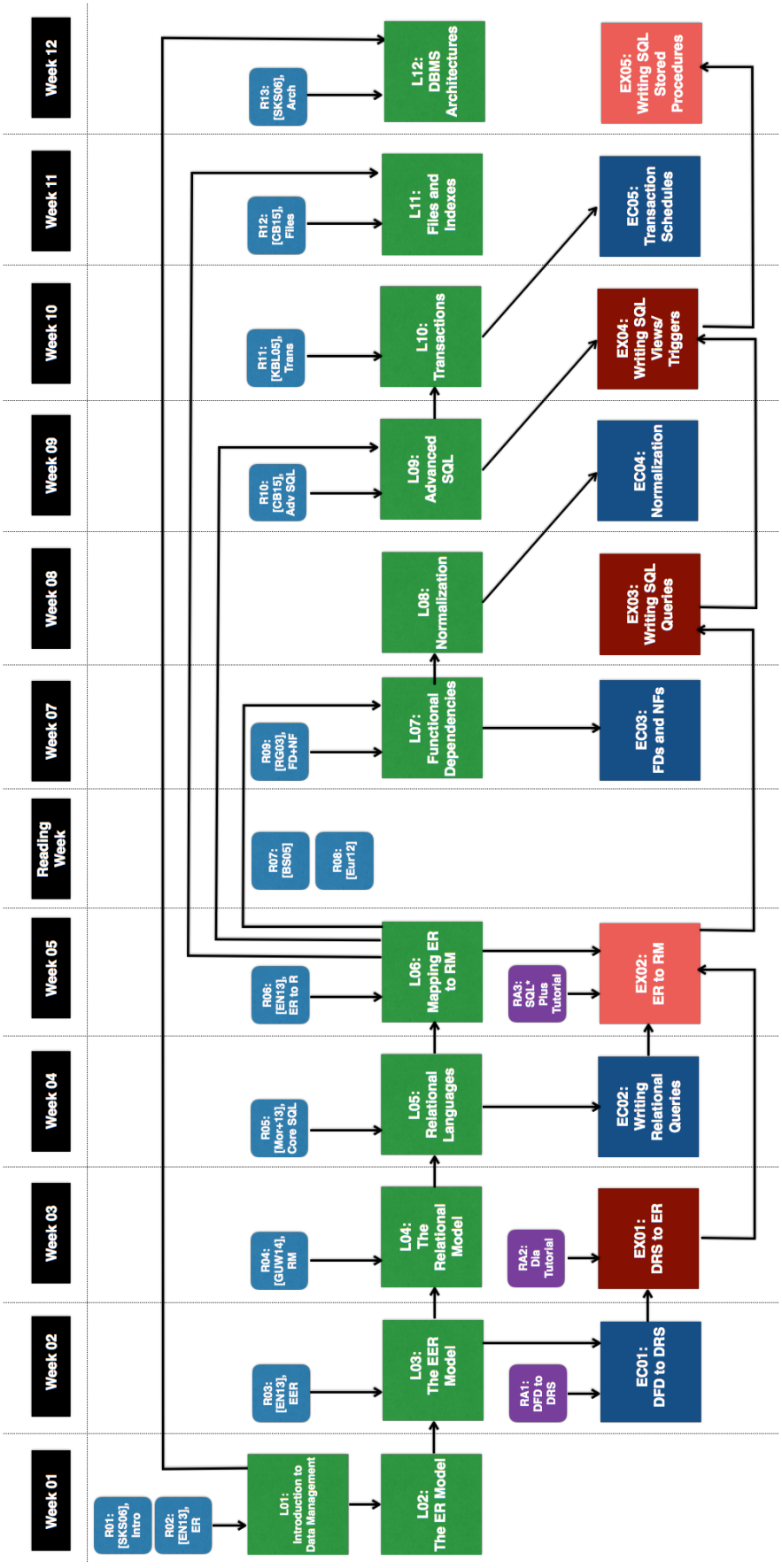

Figure 5: COMP23111: Dependency Graph

Example Class 1: Eliciting Data Requirements (1 class)

Week 02

Goal The goal of this activity is for you to practice the technique of deriving a **data requirements specification** (DRS) from a process model captured as a level-0 **data flow diagram** (DFD).

This technique is often carried out in real-world software development by a data analyst and is preparatory to engaging in database design. In practice, one or more data analysts have one or more joint, interactive face-to-face sessions with one or more stakeholders to understand the process model captured in the level-0 DFD. From such interactions, gradually, through a few iterations, a DRS emerges.

Note that, in this course unit, we cannot teach you process modelling, and hence, you won't be taught the techniques for drawing DFDs (which, of course, also means that you won't be examined on your ability to draw DFDs). However, a data analyst must work from a process model, so you do need to learn how to read a DFD, which is a popular formalism for process modelling.

Appendix A, in this document, explains how to interpret a DFD such as the one in Figure 6. The appendix also explains the basic approach to deriving a DRS from a DFD that you're going to practice in this examples class. The material in that appendix was *not* covered in any lecture. By the time this examples class is held, you should have read and studied the appendix at your own time.

In this activity, the narrower goal is for you to practice the skill of inferring from the DFD questions that, when posed to the stakeholders, elicit answers that allow you, as a data analyst, to capture one or more data requirements to go into the DRS.

Pre-/Co-Requisites Check the pre-/co-requisites in the dependency graph in Figure 5.

Materials The DFD (from [HGV08], with whom copyright rests) we will base ourselves on in this activity is shown in Figure 6.

Tasks

1. Using what you have learned by studying Appendix A, write down five questions from the question types in Figure 17.
2. Provide common-sense, intuitive answers (that conform with Figure 6) for the questions you wrote down in response to the previous task.
3. Write down the DRS items that follow from the answers you gave in response to the previous task.

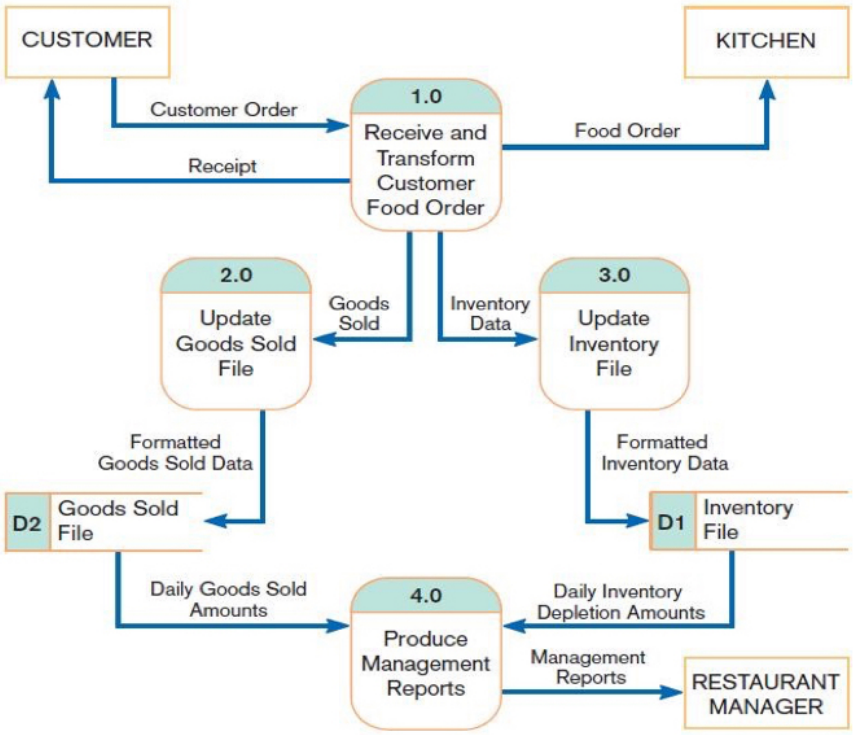

Figure 6: A Level-0 DFD for a Restaurant

Exercise 1: Mapping Data Requirements to an EER Conceptual Model (1 lab session)

Week 03

Goal The goal of this activity is for you to practice the technique of deriving a conceptual model in the form of an **(enhanced) entity-relationship** (EER) diagram from a data requirements specification (DRS), such as you have practiced deriving from a data flow diagram (DFD) in Examples Class 1.

This technique is often applied in real-world software development by a data analyst and is typically the first concrete step in the database design process.

The outcome of this step is a conceptual model from which a logical model of the data can then be derived, as you will practice in a forthcoming activity.

Pre-/Co-Requisites Check the pre-/co-requisites in the dependency graph in Figure 5.

Materials The DRS you will use is for a database with data about a **sports league**. You can think of it as a football league, but it may fit other sports as well (e.g., rugby, though perhaps not cricket).

R1 *There is a need to store data about players, referees and coaches.*

R2 *The kinds of person above can be grouped together as personnel.*

R3 *A player cannot be a referee nor a coach and a referee cannot be a coach either.*

R4 *There is also a need to store data about:*

- *(a) teams and divisions;*
- *(b) matches and incidents in matches (e.g., any unusual interruption of play).*

R5 *Every personnel has a globally unique id, a name, a date of birth, an age [derived from date of birth].*

R6 *We are interested also on:*

- *(a) the many positions a player plays in;*
- *(b) the level at which a referee referees;*
- *(c) the qualification date of a coach.*

R7 *A player plays for one team only but may not play for any. A team has many players that play for it and must have at least one. We record the dates (from and to) during which a player plays for a team.*

R8 *A coach coaches one team only but may not coach any. A team must have one and only one coach. We record the dates (from and to) during which a coach coaches a team.*

R9 *A match has a globally unique code, a date in which it took place and the attendance that was recorded.*

R10 *An incident occurs in a match. A match may have many incidents or none. An incident must have one and only one match in which it has occurred.*

R11 *An incident has a type, a time and a description. The combination of type and time identify the incidents of a given match.*

R12 *A referee is in charge of a match. A referee may have been in charge of many matches or none. A match must have one, and only one, referee that is in charge of it.*

R13 *A team has a name, a city, and the ground in which it plays home games. Names are not globally unique, but become so when combined with the city.*

R14 *A division has a globally unique name.*

R15 *A team must play in one and only one division. A division has many teams that play in it and must have one team (well, two actually) that plays in it.*

R16 *A team is the home team in many matches and must be in at least one. A match must have a home team and only one.*

R17 *A team is the away team in many matches and must be in at least one. A match must have an away team and only one.*

R18 *A player is in play at many matches but may not have in play at any. A match has many players in play at it and must have at least one (well, it varies from sport to sport, and then there are substitutions). We record the time (from and to) during which a player is in play at a match.*

R19 *A player may score in many matches and s/he may do so many times in a match. A match may have many (or no) player that score in it. We record the time at which a player scored and whether it was an own goal or not.*

Tasks

1. Use Dia (you will be familiar with it from the tutorial, in Appendix B, which you should have completed by now) to design the conceptual model in the form of an EER diagram that follows from the DRS above.

2. Save the diagram in a Dia file using the filename that instantiates the following template:

   `EX01-<studentID>.dia`

   where (here and elsewhere) you should replace `<studentID>` with your student ID, of course.

   Below, we refer to this file as the *proof-of-work* file.

3. Export the diagram as a PNG file using the filename that instantiates the following template:

   `EX01-<studentID>.png`

   Below, we refer to this file as the *for-insertion* file.

Assessment Type This activity is subject to **summative** assessment, therefore your submission will be marked and you will get the associated feedback.

The marks you obtain (see below) count for up to 30 % of your overall coursework mark for this course unit.

Marks There are 30 marks available.

- Marks are awarded for correct, complete entity types (plus attributes) and relationship types (plus attributes), where specializations/generalizations count as the latter, of course.
- Roughly (i.e. not strictly), each correct, complete entity type (plus attributes) is worth one mark, and each each correct, complete relationship type (plus attributes) is worth two marks.
- If the marker finds that any of your submitted files is unreadable, you lose all the marks, so, test every one of them beforehand.
- If the marker finds that any of your submitted files is hard to read (e.g., poor or no legibility, or congested/overlapping items/lines, etc.) or is missing information (say, there is no cover page), you may lose marks up to 10% of the available marks.

Submission Procedure

1. Create a document file, with a cover page stating the course unit (COMP23111), the academic year (2016-2017), the exercise number (EX01) your name, and your student ID.
2. Insert the *for-insertion* file in the document you're preparing.
3. Type in any assumptions and/or comments you wish to make.
4. Save/Export the document as a PDF file naming it as

   `EX01-<studentID>.pdf`

   Below, we refer to this as the *submission* file.
5. Submit both the *proof-of-work* and the *submission* files using Blackboard. If you fail to submit either file, your mark will be reduced by 1%. If you make the same mistake again, the penalty rises to 3%. Any further failure incurs a penalty of 10%.

Deadline/Extensions/Penalties

The deadline for submission is

17:00 on Friday in Week 04

Please, read the relevant section in the Introduction for this course unit's policy on extensions and penalties.

Example Class 2: Writing Relational-Algebraic Expressions (1 class)

Week 04

Goal The goal of this activity[2] is for you to practice the technique of writing relational-algebraic expressions against a relational database schema.

This technique is often carried out in real-world software development by a database administrator in order to obtain a more abstract view of the computations that the DBMS will be asked to carry out than the one the corresponding SQL query provides.

This is useful because, inside the DBMS, the query optimizer translates SQL into relational algebra so as to facilitate rewriting (in the search for heuristically more efficient equivalent queries) and, later, estimate the cost of evaluating the rewritten query in the light of the alternative access paths to the data (e.g., available indices).

Pre-/Co-Requisites Check the pre-/co-requisites in the dependency graph in Figure 5.

Materials

THE UNIVERSITY DATABASE The database against which you should write the relational-algebraic expressions in this examples class is the University Database (from [SKS06]).

Its schema is represented in diagrammatic form in Figure 7.

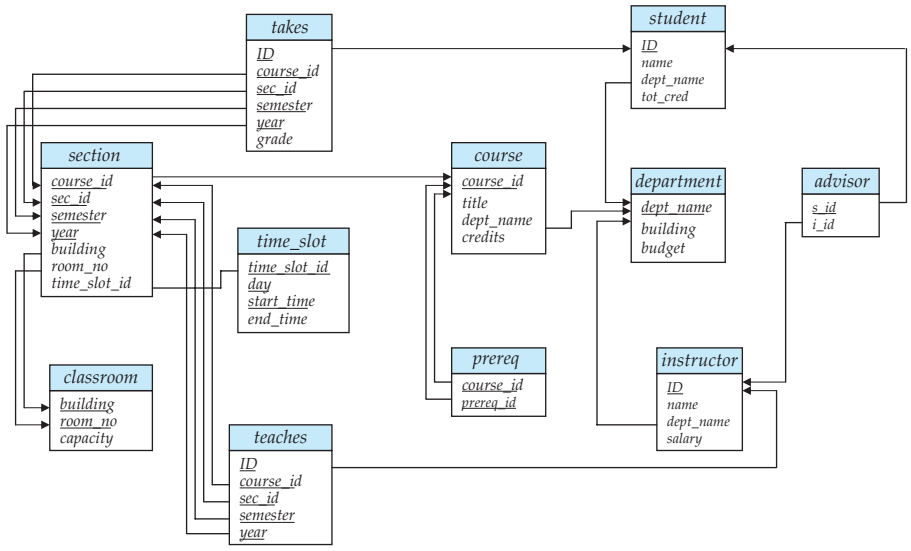

Figure 7: The University Database [SKS06]

The notation used in Figure 7 is as follows:

- the database schema is a directed graph;

[2]This is based on [SKS06], with whom copyright rests. Errors are this author's responsibility.

- a node denotes a relation schema (where the relation name is in the darker header, e.g., `student`, `advisor`, etc.) and lists the columns (whose names are in the lighter area, e.g., `tot_cred`), with columns that are key, or part thereof, underlined (e.g., `dept_name`);
- a directed edge from a source node R to a destination node S denotes that the attribute x of R references the attribute y of S (e.g., `s_id` in `advisor` references `id` in `student`).

Tasks Code[3] solutions to the following problems as relational-algebraic expressions against the University Database schema:

1. Retrieve the data about the instructors in the Physics department.
2. Retrieve the ID, name and salary of the instructors in the university.
3. Retrieve the course ID of all the courses taught in the Fall 2009 semester, or in the Spring 2010 semester, or in both.
4. Retrieve the course ID of all the courses taught in the Fall 2009 semester but not in the Spring 2010 semester.
5. Retrieve the course ID of all the courses taught in the Fall 2009 semester and in the Spring 2010 semester.
6. Retrieve the combination of the available data for each instructor and for each course taught.

   Hint: this is a Cartesian product, insofar as, here, it is not required that the combination is only between courses C taught by an instructor I, i.e., every instructor there is is combined with every taught course there is.
7. Retrieve the salaries of instructors that are less than some other instructor salary.

   Hint: you will need to do a self-product (i.e., take the product of a relation with itself) and this requires you to use renaming (ρ) to be able to refer separately and independently to each copy that is an input to the self-product.
8. Now, use expression you've produced in (7) above to compute the largest salary (i.e., a salary such that is not less than any other salary in the university).
9. Now, rewrite the expression you've produced in (8) above using assignment to make clear the prior step in its construction.
10. Retrieve the names of all instructors in the Physics department along with the course ID of all courses they have taught.
11. Use the natural join to retrieve all the data about a student for every student who takes some course.
12. Now, rewrite the expression you've produced in (11) above replacing the natural join with a Cartesian product followed by a selection.
13. Retrieve the names of all instructors in the Comp. Sci. department together with the course titles of all the courses that the instructors teach.

ID	*name*	*dept_name*	*salary*
10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	60000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	62000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000

ID	*course_id*	*sec_id*	*semester*	*year*
10101	CS-101	1	Fall	2009
10101	CS-315	1	Spring	2010
10101	CS-347	1	Fall	2009
12121	FIN-201	1	Spring	2010
15151	MU-199	1	Spring	2010
22222	PHY-101	1	Fall	2009
32343	HIS-351	1	Spring	2010
45565	CS-101	1	Spring	2010
45565	CS-319	1	Spring	2010
76766	BIO-101	1	Summer	2009
76766	BIO-301	1	Summer	2010
83821	CS-190	1	Spring	2009
83821	CS-190	2	Spring	2009
83821	CS-319	2	Spring	2010
98345	EE-181	1	Spring	2009

Figure 8: Tables `instructor` and `teaches` from the University Database

14. Assume that at time t the state of the `instructor` and the `teaches` tables was as shown in Figure 8.

    Now let I' and T' be the result of evaluating the following expressions over Figure 8 at time t:

$$I' \leftarrow \sigma_{ID=10101 \vee ID=12121 \vee ID=15151}(\pi_{ID,name,dept\_name}(instructor))$$

$$T' \leftarrow \sigma_{course\_id='CS-101' \vee course\_id='FIN-201' \vee course\_id='BIO-101'}(\sigma_{ID=10101 \vee ID=12121 \vee ID=76766}(\pi_{ID,course\_id}(teaches)))$$

    (a) Write down the extent of I' and T'.

    (b) Write down the table that results from evaluating the expression $I' \bowtie T'$ at time t.

    (c) Write down the table that results from evaluating the expression $I' ⟕ T'$ at time t.

    (d) Write down the table that results from evaluating the expression $I' ⟖ T'$ at time t.

    (e) Write down the table that results from evaluating the expression $I' ⟗ T'$ at time t.

15. Assume the `salary` column in the `instructor` table is an annual salary. Retrieve the name and monthly salary of every instructor.

16. Retrieve the average annual salary per department.

[3] If any of the specifications for these (or any other) coding tasks in this manual seem ambiguous to you, talk to the course staff, preferably in person (in an examples class or a lab session, or in the open office hour for this course unit) to clarify it. If you can't talk face-to-face, use email, but, please, don't just ignore it as you may misinterpret the task and lose marks as a result.

Exercise 2: Mapping an EER Conceptual Model to a Relational Logical Model (1 lab session)

Week 05

Goal The goal of this activity is for you to practice the technique of deriving a logical model in the form of a relational database schema from a conceptual model in the form of an **(enhanced) entity-relationship** (EER) diagram, such as you have practiced deriving from a data requirements specification (DRS) in Exercise 1.

This technique is often applied in real-world software development by a database designer after there is agreement with the stakeholders that the conceptual model captures all the data assets of relevance to an organization's information systems.

The outcome of this step is a relational database schema that can then be translated into a script in the implemented data definition language of a specific DBMS platform (e.g., SQL in Oracle). Such a script can then be used to create all the database objects envisaged in the conceptual model. This, in turn, allows the database to be populated with actual data. In this activity, we limit ourselves to the derivation of the relational database schema.

Pre-/Co-Requisites Check the pre-/co-requisites in the dependency graph in Figure 5.

Materials

THE ORINOCO DATABASE The EER diagram you will use is the complete version of the Orinoco database, a part of which you will be familiar with from the SQL*Plus tutorial, in Appendix C, which you should have completed by now. Start by reminding yourself of the requirements expressed there.

The complete version of the diagram in Figure 9 captures the following additional requirements:

- *We relate an album with the finished tracks it contains. An album must have at least one, and normally many, finished tracks. A finished track may appear in many albums. We also capture the sequence in which finished tracks appear in an album. Note that the same finished track may appear in different albums in different sequences.*
- *The system must keep an online collection of catalogue entries, each of which references an existing album. An album must be listed as a catalogue entry at least once, and may be listed as several catalogue entries. Each catalogue entry is differentiated by its release date (to account for re-releases of the album). Both the price and the stock for each catalogue entry are also stored.*
- *We record the name of buyers as well as the date in which they have registered with the Orinoco store. We also record a buyer ID to uniquely identify buyers.*
- *A buyer can place any number of orders. An order must be placed by a buyer and is identified by an order number. We also store the date an order has been placed on by a buyer and the dispatched date of an order. An order comprises several catalogue entries. A catalogue entry can be part of many orders.*

- *There exist two types of artists: solo artists and group artists. A group artist can have as members more than one solo artist. Solo artists and group artists form disjoint entity sets.*
- *We also capture the date a solo artist has joined a particular group and the fact that a solo artist can be a member of more than one group.*
- *For solo artists, we would also like to store their real names and when they had their first performance.*
- *For group artists we would like to store the date a particular group has been formed.*
- *We also model the fact that an album is distributed as a vinyl album or a tape album or a CD album. Again, these form pairwise disjoint entity sets*
- *A vinyl album can have many distinct colours, a CD album can have many different extras (e.g., an associated videoclip), and a tape can have exactly one label.*

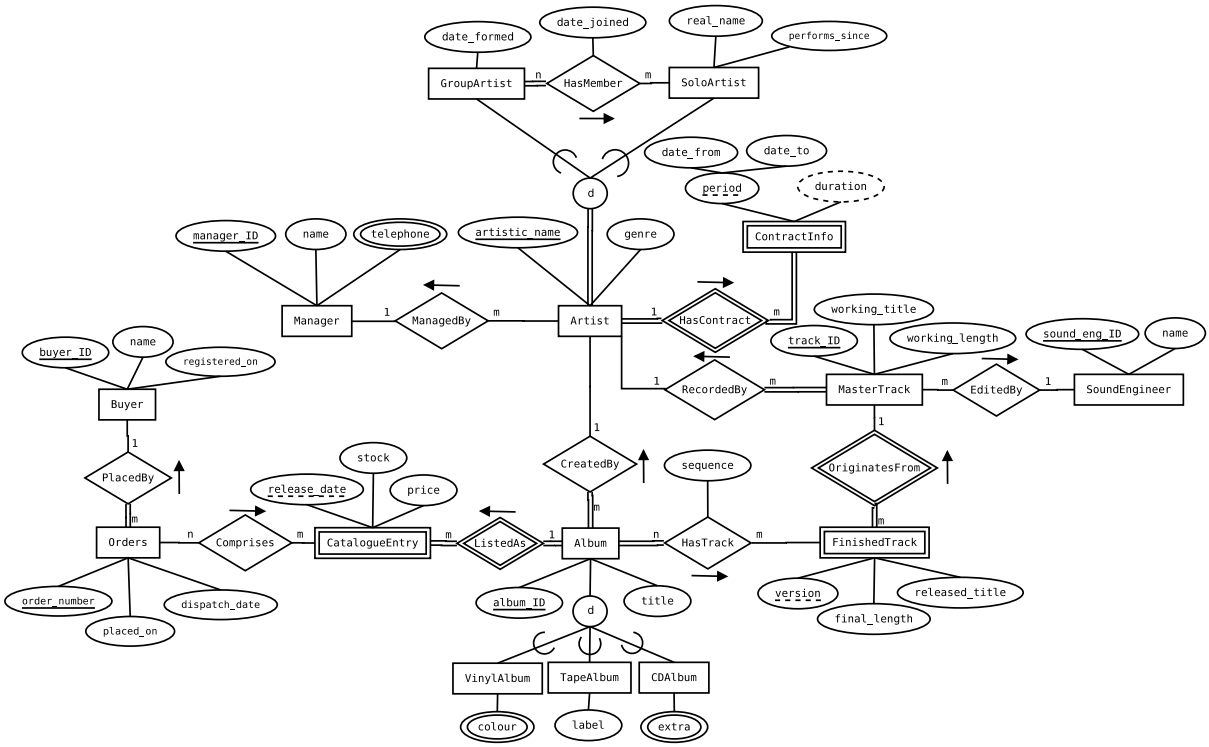

Figure 9: Orinoco [Complete]: Conceptual Model as an ER Diagram

Tasks

1. Remind yourself of the partial database schema corresponding to the partial conceptual model of the Orinoco database in Figure 24 (Appendix C). Your submission should follow the notation used there.

2. Complete the partial relational database schema in Figure 24 (Appendix C) by taking into account the additional requirements captured by the ER model in Figure 9. You should use the procedure you were taught in this course unit.

3. Save the relational database schema – preferably complete, i.e., add the relations in Figure 24 (Appendix C) to the new ones you have just derived – in a text file using the filename that instantiates the following template:

`EX02-<studentID>.txt`

where (here and elsewhere) you should replace `<studentID>` with your student ID, of course.

Below, we refer to this as the *proof-of-work* file. In the case of this exercise, this file is also what we refer to, below, as the *for-insertion* file.

Note that there is no standard, established notation for representing relational schemas textually.

When attribute names are unambiguous across the whole database schema, underlining keys works (solid for primary and dashed for foreign, unless the latter also become part of the primary).

An alternative, but still compact, notation that copes with potential ambiguity and doesn't require underlining is exemplified by:

```
R(r, b, c): pk[r]
S(s, d, e): pk[s,d],  fk[s -> R.r]
T(t, e, f): pk[t,e],  fk[t -> R.r]
U(u, g, h): pk[u,g,h]
V(v, i)   : pk[v],    fk[i -> U.u]
```

This notation also has the advantage of corresponding closely to SQL DDL statements, as follows (assuming column types):

```
create table R (r integer, b integer, c integer,
  primary key (r)  );
create table S (s integer, d integer, e integer,
  primary key (s,d),
  foreign key (s) references R(r)  );
create table T (t integer, e integer, f integer,
  primary key (t,e),
  foreign key (t) references R(r)  );
create table U (u integer, g integer, h integer,
  primary key (u,g,h)  );
create table V (v integer, i integer,
  primary key (v),
  foreign key (i) references U(u)  );
```

Assessment Type This activity is subject to *formative* assessment, therefore your submission will not be marked but you will get the associated feedback. In other words, for this activity, your submission does not contribute to the overall coursework mark for this course unit.

Submission Procedure

1. Create a document file, with a cover page stating the course unit (COMP23111), the academic year (2016-2017), the exercise number (EX02) your name, and your student ID.
2. Insert the *for-insertion* file in the document you're preparing.
3. Type in any assumptions and/or comments you wish to make.
4. Save/Export the document as a PDF file naming it as

`EX02-<studentID>.pdf`

Below, we refer to this as the *submission* file.

5. Submit both the *proof-of-work* and the *submission* files using Blackboard.

Deadline/Extensions/Penalties

The deadline for submission is

17:00 on Friday in Week 07

Since this activity is only subject to formative assessment, extensions and penalties are not applicable.

Example Class 3: Reasoning about Normal Forms and Functional Dependencies (1 class)

Week 07

Goal The goal of this activity[4] is for you to practice the technique of deriving a relational database schema directly from a data requirements specification (DRS), such as you have practiced deriving from a data flow diagram (DFD) in Example Class 1.

Notice that, in this case, we do not first derive an EER model from which the relational database schema is obtained by application of the mapping procedure you have practiced in a previous activity. We do it directly from the DRS.

In this case, the database schema derivation technique uses functional dependencies and normalization to arrive at the logical model without going through the conceptual model.

This technique is often applied in real-world software development by a database designer when a relational database schema already exists but needs expansion, or else needs to be revised if it has been otherwise modified. Therefore, unlike the mapping from EER to relational, this technique is better seen as a verification/validation one, and, therefore, in practice, more likely to be deployed in maintenance tasks, rather then construction for first deployment (from scratch).

The outcome of this step is a normalized relational database schema.

Pre-/Co-Requisites Check the pre-/co-requisites in the dependency graph in Figure 5.

Materials We will assume that the target is an (*American*) College Transcript Database (from [EN13]). This is not to be confused with the University Database (from [SKS06]) that we use for other activities in this course unit.

The DRS for the College Transcripts Database that you were provided with is as follows:

R1 The college keeps track of each student's name (`SNAME`), student number (`SNUM`), social security number (`SSSN`), current address (`SCADDR`) and phone (`SCPHONE`), permanent address (`SPADDR`) and phone (`SPPHONE`), birthdate (`BDATE`), sex (`SEX`), class (`CLASS`) (i.e., `freshman`, `sophomore`, ..., `graduate`), major department (`MAJORDEPTCODE`), minor department (`MINORDEPTCODE`) (if any), and degree program (`PROG`) (i.e., `B.A.`, `B.S.`, ..., `Ph.D.`). Both the social security number and the student number have unique values for each student.

R2 Each department is described by a name (`DEPTNAME`), department code (`DEPTCODE`), office number (`DEPTOFFICE`), office phone (`DEPTPHONE`), and school (`DEPTSCHOOL`). Both name and code have unique values for each department.

R3 Each course has a course name (`CNAME`), description (`CDESC`), code number (`CNUM`), number of semester hours (`CREDIT`), level (`LEVEL`), and offering department (`CDEPT`). The value of code number is unique for each course.

[4] This is based on [EN13], with whom copyright rests. Errors are this author's responsibility.

R4 Each section has an instructor (`INSTRUCTORNAME`), semester (`SEMESTER`), year (`YEAR`), course (`SECCOURSE`), and section number (`SECNUM`). Section numbers distinguish different sections of the same course that are taught during the same semester/year; its values are 1, 2, 3, ...; up to the number of sections taught during each semester.

R5 A grade record refers to a student (`Ssn`), refers to a particular section, and grade (`GRADE`).

Tasks Design a relational database schema for this database application using the following steps:

1. Using your intuition and common-sense knowledge, go through the requirements and note down any unspecified or ambiguous information, then make appropriate assumptions to make the specification complete and precise for you.
2. Show all the functional dependencies that, from the requirements (made complete and precise by you), you believe hold among the attributes.
3. Design relation schemas for the database that are each in 3NF or BCNF. Explain your decisions.
4. Specify the primary key and foreign key(s) of each relation. Explain your decisions.

Exercise 3: Writing SQL Queries (1 lab session)

Week 08

Goal The goal of this activity[5] is for you to practice the technique of writing SQL queries against a relational database, such as resulted from creation and population SQL scripts we provide you with.

The creation SQL script would have been composed taking into account a relational database schema that, in its turn, would have been derived from a conceptual model, a technique you have practiced in an earlier activity.

The population script has example data only, and does not insert a lot of data, so your queries should run fast, once you have got them syntactically and semantically correct.

The outcome of this step is a set of SQL queries that comprise a subset of the queries with which the database designer would validate the logical model against the DRS that was agreed between the data analysts and the stakeholders.

The technique of writing queries in SQL is, therefore, highly valuable. In real-world software development, a database designer would respond to requests by application designers to write the SQL queries that, when incorporated within the application code, would retrieve the data that application needs to perform its function in the context of an organization's information systems.

Pre-/Co-Requisites Check the pre-/co-requisites in the dependency graph in Figure 5.

Materials

THE UNIVERSITY DATABASE The database against which you should write the SQL queries in the first part of the exercise is the University Database (from [SKS06]).

Its schema is represented in diagrammatic form in Figure 10, repeated here from Example Class 2 (where the notation was explained).

In order to code and debug your SQL queries, we provide you with SQL*Plus scripts to create the tables, populate the tables, and drop the tables.

Of course, if something gets garbled somehow, you can recreate the tables from scratch, by running the script to drop the table, then the script to create them, and, finally, the script to populate them again. This suggests that your queries should, of course, be written into a different script (i.e., a different `.sql` file).

The scripts are available as:

```
/opt/info/courses/COMP23111/create-University-tables.sql
/opt/info/courses/COMP23111/populate-University-tables.sql
/opt/info/courses/COMP23111/drop-University-tables.sql
```

[5] This is based on [SKS06], with whom copyright rests. Errors are this author's responsibility.

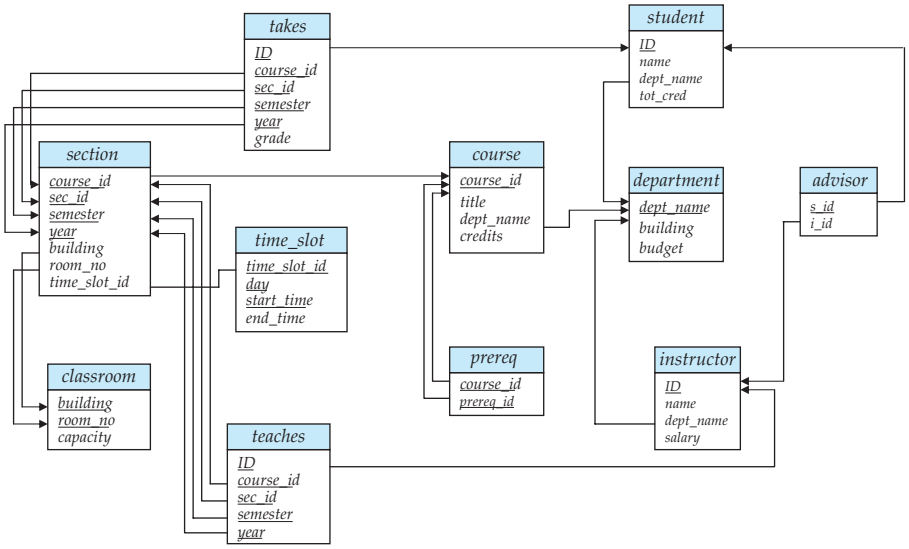

Figure 10: The University Database [SKS06] [same as Figure 7]

THE ACCIDENT DATABASE The database against which you should write the SQL queries in the second part of the exercise is the Accident Database (from [SKS06]).

Its schema is the following:

```
person(driver_id, name, address):
   pk[driver_id]
car(license, model, year):
   pk[license]
accident(report_number, accident_date, location):
   pk[report_number]
owns(driver_id, license):
   pk[driver_id, license],
   fk[driver_id -> person.driver_id,
      license -> car.license]
participated(report_number, license, driver_id, damage_amount):
   pk[report_number, license],
   fk[report_number -> accident.report_number,
      license -> car.license]
```

We provide you with SQL*Plus scripts to create the tables, populate the tables, and drop the tables:

```
/opt/info/courses/COMP23111/create-Accident-tables.sql
/opt/info/courses/COMP23111/populate-Accident-tables.sql
/opt/info/courses/COMP23111/drop-Accident-tables.sql
```

Tasks

1. Use the University Database for this part.

   (a) Code[6] solutions to the following problems as SQL statements against the University Database schema:

[6] If any of the specifications for these (or any other) coding tasks in this manual seem ambiguous to you, talk to the course

i. Find the names of all students who have taken at least one computer science course, making sure there are no duplicate names in the result.

ii. Find the IDs and names of all students who have not taken any course offering before Spring 2009.

iii. For each department, find the maximum salary of instructors in that department. You may assume that every department has at least one instructor.

iv. Find the lowest, across all departments, of the per-department maximum salary computed by the preceding query.

(b) Code solutions to the following manipulation problems as SQL statements against the University Database schema:

i. Create a new *CS-001* course in computer science, titled *Weekly Seminar*, with 10 credits.

ii. Create a new *CS-002* course in computer science, titled *Monthly Seminar*, with 0 credits.

iii. Add a comment to your script explaining the error you get in trying to do this using as evidence the Oracle documentation and the script that created the tables, then comment out the offending SQL statement in your script.

iv. Create a section of the *CS-001* course in Fall 2009, with section id of 1.

v. Add a comment to explain what you have assumed regarding missing columns using as evidence the Oracle documentation and the script that created the tables.

vi. Enrol every student in the Computer Science department in the section you created in the previous statement.

vii. Delete all enrolments in the above section where the student's name is *Zhang*.

viii. Delete all takes tuples corresponding to any section of any course with the word *database* as a part of the title (you should make sure that your code is case-insensitive when matching the word with the title).

ix. Delete the course *CS-001*.

x. Add a comment to your script explaining what allows the previous statement to run without error using as evidence the Oracle documentation and the script that created the tables.

2. Use the Accident Database for this part.

(a) Code solutions to the following query problems as SQL statements against the Accident Database schema:

i. Find the number of accidents in which the cars belonging to Jane Rowling were involved.

ii. Update the amount of damage for the car with license number KUY 629 in the accident with report number 7897423 to 2500.

iii. List the name of the persons that participated in accidents along with the total damage caused (from the largest to the smallest) but only include those whose total damage is above 3000.

iv. Create a view that returns the locations where accidents have occurred along with the average amount of damage in that location. Call this view `average_damage_per_location`.

v. Use the `average_damage_per_location` location you have just created to find the location that has the highest average damage.

staff, preferably in person (in an examples class or a lab session, or in the open office hour for this course unit) to clarify it. If you can't talk face-to-face, use email, but, please, don't just ignore it as you may misinterpret the task and lose marks as a result.

3. After you have debugged your work, collect all the code with comments in a single SQL*Plus script, extend this with echoing and spooling commands (you are expected to have learned to do this in doing the tutorial in Appendix C), then run this extended script to produce a log file making sure that its filename instantiates the following template:

   `EX03-<studentID>.sql`

   where (here and elsewhere) you should replace `<studentID>` with your student ID, of course.

   Below, we refer to this as the *proof-of-work* file. In the case of this exercise, this file is also what we refer to, below, as the *for-insertion* file.

Assessment Type This activity is subject to **summative** assessment, therefore your submission will be marked and you will get the associated feedback.

The marks you obtain (see below) count for up to 35 % of your overall coursework mark for this course unit.

Marks There are 35 marks available.

- Marks are awarded for correct, complete SQL expressions/statements and for correct explanations in the case of comments.
- Each correct, complete SQL expression/statement is worth two marks whereas each correct explanation in the form of required comments is worth one mark.
- If the marker finds that any of your submitted files is unreadable or (in the case of a script) not executable, you lose all the marks, so, test every one of them beforehand.
- If the marker finds that any of your submitted files is hard to read (e.g., poor or no indentation, or congested/overlapping items/lines, etc.) or is missing information (say, there is no cover page), you may lose marks up to 10% of the available marks.

Submission Procedure

1. Create a document file, with a cover page stating the course unit (COMP23111), the academic year (2016-2017), the exercise number (EX03) your name, and your student ID.
2. Insert the *for-insertion* file in the document you're preparing.

   Given that the *for-insertion* file is a SQL*Plus script, (i.e., code) you must make sure that it's typeset in a **monospaced font**[w] (e.g., Courier, Courier New, Lucida Console, Monaco, etc.)[7] so that the indentation in your code is respected. You'll lose marks due to poor presentation if you don't.
3. Type in any assumptions and/or comments you wish to make.
4. Save/Export the document as a PDF file naming it as

   `EX03-<studentID>.pdf`

[7] Consult the course staff if you're in doubt here.

Below, we refer to this as the *submission* file.

5. Submit both the *proof-of-work* and the *submission* files using Blackboard. If you fail to submit either file, your mark will be reduced by 1%. If you make the same mistake again, the penalty rises to 3%. Any further failure incurs a penalty of 10%.

Deadline/Extensions/Penalties

The deadline for submission is

17:00 on Friday in Week 09

Please, read the relevant section in the Introduction for this course unit's policy on extensions and penalties.

Example Class 4: Normalizing a Relational Logical Model (1 class)

Week 09

Goal The goal of this activity[8] is for you to practice the technique of using functional dependencies to normalize a relational database schema.

In this example class, since it is not unwieldy, we will start with the universal schema, i.e., a single-relation database schema, where all the attributes of interest belong to the single relation in the database schema. However, we could start from a partially normalized schema as well.

This technique is often applied in real-world software development by a database designer when a relational database schema already exists but needs expansion, or else needs to be revised if it has been otherwise modified.

The outcome of this step is a normalized relational database schema.

Pre-/Co-Requisites Check the pre-/co-requisites in the dependency graph in Figure 5.

Tasks Consider the universal relation

$$R = \{A, B, C, D, E, F, G, H, I\}$$

and the set of functional dependencies

$$\begin{aligned} F = \{&\{A, B\} \rightarrow \{C\}, \\ &\{A\} \rightarrow \{D, E\}, \\ &\{B\} \rightarrow \{F\}, \\ &\{F\} \rightarrow \{G, H\}, \\ &\{D\} \rightarrow \{I, J\} \\ \} \end{aligned}$$

1. What is the key for R? Explain your answer.
2. Decompose R into 2NF, then 3NF relations. Explain your decisions.

[8] This is based on [EN13], with whom copyright rests. Errors are this author's responsibility.

Exercise 4: Writing SQL Views and SQL Triggers (1 lab session)

Week 10

Goal The goal of this activity[9] is for you to practice the technique of writing SQL views and triggers against a relational database, such as resulted from creation and population SQL scripts we provide you with.

The creation SQL script would have been composed taking into account a relational database schema that, in its turn, would have been derived from a conceptual model, a technique you have practiced in an earlier activity.

The population script has example data only, and does not insert a lot of data, so your queries should run fast, once you have got them syntactically and semantically correct.

The outcome of this step is a set of SQL views and triggers that comprise a subset of the DBMS-supported functionality with which the database designer would enforce some aspects of the semantics captured by the logical model. This functionality is what stakeholders see as business rules, which express the way in which the organization either chooses to operate or is compelled (e.g., by legislation) to operate. As such, they must not be violated.

The technique of writing views and triggers in SQL is highly valuable. In real-world software development, a database designer would respond to requests by application designers to write the SQL code that would, when incorporated within the application code, monitor or enforce organization-wide access and/or update policies that application assume to hold in order to perform its function in the context of an organization's information systems.

Pre-/Co-Requisites Check the pre-/co-requisites in the dependency graph in Figure 5.

Materials

THE ECLECTIC-ECOMMERCE DATABASE The database against which you should write the SQL queries in this exercise is the Eclectic-Ecommerce (from [DU04]). Its description and conceptual model are given in Figs. 11 and 12.

The logical model can be gleaned from the relational schema given in SQL*Plus script form. We provide you with SQL*Plus scripts to create the tables, populate the tables, and drop the tables:

```
/opt/info/courses/COMP23111/create-Eclectic-Ecommerce-tables.sql
/opt/info/courses/COMP23111/populate-Eclectic-Ecommerce-tables.sql
/opt/info/courses/COMP23111/drop-Eclectic-Ecommerce-tables.sql
```

[9] This is based on [DU04], with whom copyright rests. Errors are this author's responsibility.

The ECLECTIC mail-order company would like to provide a means for its customers to shop and place orders over the Web. ECLECTIC provides a wide range of products, such as thing-a-ma-bobs, deely-bobs, and widgets.

ECLECTIC customers have typical information: name (last name and first name), address (street, city, state, and zip code), phone number, and email. To place an order, a customer must be a registered user having a unique login name and password.

The customer shops the online store by category, where a category has a unique code and description. Each type of item in the online store belongs to exactly one category. For example, books might represent one category, while electronics and sporting goods might represent other categories.

An item type is described by a unique item number, a name, and a price and has an associated graphic for display on the Web page. Since inventory items may come in different colors and sizes, a customer selects a specific inventory item by also specifying its color and size. Inventory items are represented in the database by a code that is unique within the item number. The database also records the current quantity in stock of that inventory item for a given color and size. For example, a thing-a-ma-bob is an item type in the electronics category. A thing-a-ma-bob, however, comes in different sizes and colors. The online store may have 50 small, green thing-a-ma-bobs in stock and may have only 30 large, blue thing-a-ma-bobs in stock.

Figure 11: The Eclectic-Ecommerce Database (part 1 of 2) [DU04]

A customer places inventory items in a shopping cart, specifying the quantity of that item being placed in the cart. There is a unique number associated with the cart, along with the date and total price, which is calculated as the sum of multiplying the price of each inventory item in the shopping cart by the quantity ordered. The contents of the shopping cart can be updated until the customer confirms the cart contents by placing an order. The quantity of each item in the shopping cart can also change before the customer places the order. When an order is placed, the shopping cart is reclassified as an order that is ready for shipment. The price of each shopping cart item at the time of the order is recorded for historical purposes and for calculating the final total price of the order, which is the sum of multiplying the price of each inventory item in the shopping cart by the quantity ordered.

A customer can have at most one shopping cart, but many orders. A shopping cart and an order are associated with exactly one customer. When a shopping cart becomes an order, the shopping cart is emptied to prepare for a future shopping session. Information about customer orders is maintained in the database for a period of three years.

Figure 1.1 presents the ER diagram developed as a result of analyzing the requirements of the ECLECTIC ONLINE SHOPPING ENTERPRISE.

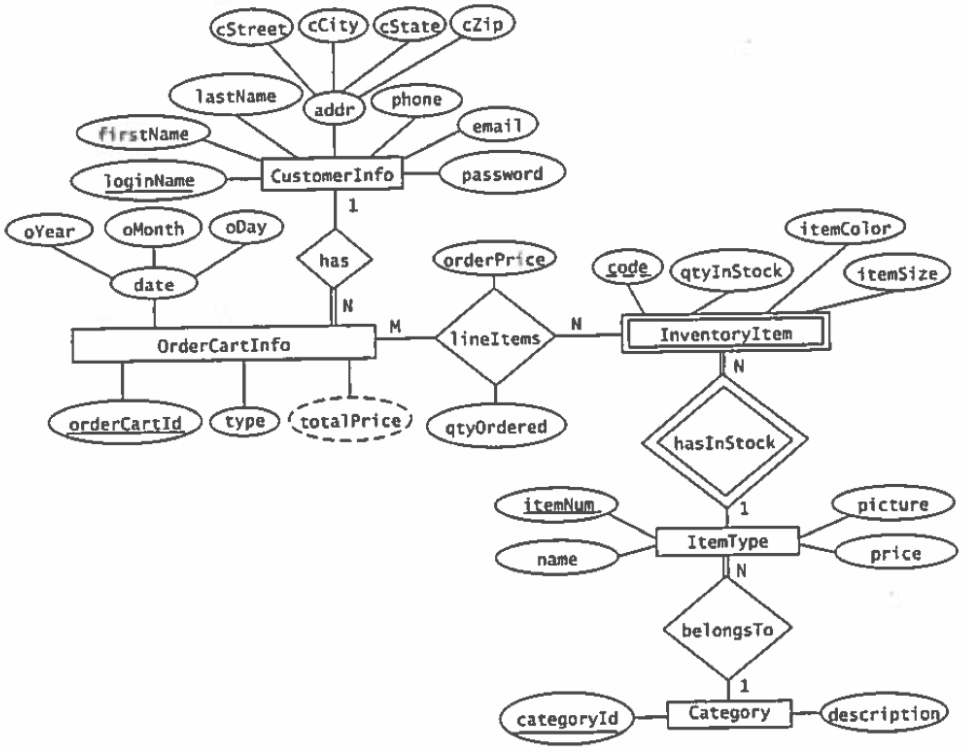

Figure 1.1 ER Diagram for the ECLECTIC ONLINE SHOPPING ENTERPRISE

Figure 12: The Eclectic-Ecommerce Database (part 2 of 2) [DU04]

Tasks

1. Code[10] solutions to the following problems as SQL statements against the Eclectic-Ecommerce schema:

   (a) Create a view showing the first and last names of customers with shopping carts, then write a query that return its full extent.

   Note that when creating a view, the view may already exist, in which case Oracle offers a `CREATE OR REPLACE` form of the statements that is the one you should always use for creating views.

   (b) Create a view showing the code, item number, category id and quantity in stock of inventory items that need to be reordered (where an inventory item needs to be reordered if the quantity in stock drops below 25), then write a query that return its full extent.

   (c) Create a view showing the login name, first and last names, order cart id and total price of each order, then write a query that return its full extent.

   (d) Create a view showing the login name, first and last names, and total of all orders by a customer, then write a query that return its full extent.

   (e) Create a view to return the number of carts per customer, then use this view in a query with a `CASE` statement in the `SELECT` clause that, for each customer, returns the login name and an outcome (represented as a string), which is either `'BR-1 satisfied'` if that customer has no more than two carts in the database, or `'BR-1 violated'` otherwise.

   (f) Now, rather than define a view, use query nesting in a similar problem, as follows. Firstly, define a query (we'll refer to it as $Q2$) that returns item num, item size, item colour and a count of how many items of a given color and size there are. Secondly, define a query (we'll refer to it as $Q1$) that nests $Q2$ in its `FROM` clause and (somewhat similarly to the previous task) has a `CASE` statement in the `SELECT` clause that returns as outcome the string `'BR-2 satisfied'` if the item number does not occur more than once with the same color and size, or `'BR-2 violated'` otherwise. Finally, define a (top-level, as it were) query that uses $Q1$ and when executed returns only the item number, color and size that violate the `BR-2` business rule.

   (g) Code a SQL trigger that raises an error if the price of an item is set to a value that is more than four times the value of the least expensive item.

   In doing this task, make sure that in your final submitted script you have a sequence such as follows:

      i. a SQL query that retrieves all the tuples in the `itemType` table (so that you know the state before the trigger is fired)
      ii. one or more SQL `INSERT` statements that cause the trigger to fire, followed by SQL queries that retrieve all the tuples in the table (so that you know the state after the `INSERT` was attempted on a valid and an invalid case)
      iii. one or more SQL `UPDATE` statements that cause the trigger to fire, followed by SQL queries that retrieve all the tuples in the table (so that you know the state after the `INSERT` was attempted on a valid and an invalid case)

   Note that, as with views, when creating a trigger, the trigger may already exist, in which case Oracle offers a `CREATE OR REPLACE` form of the statement that is the one you should always use for creating triggers.

[10] If any of the specifications for these (or any other) coding tasks in this manual seem ambiguous to you, talk to the course staff, preferably in person (in an examples class or a lab session, or in the open office hour for this course unit) to clarify it. If you can't talk face-to-face, use email, but, please, don't just ignore it as you may misinterpret the task and lose marks as a result.

Note also that you should add a single slash as the first and single character in the final line of a trigger. This is so that the parser does not take any `';'` character in the body of the trigger as the end of the statement.

One important additional piece of information that the SQL*Plus tutorial didn't cover is that if you get an error on a trigger named `<triggerName>` (and, of course, you're likely to), you can get more information on what caused the error by using `SHOW ERRORS TRIGGER <triggerName>`.

One additional thing to pay attention to here is that when you're debugging your triggers, an error may leave the database in an unexpected and undesirable state. Because of this, always think it through and consider the need to take action to reverse the changes and how to do so.

2. After you have debugged your work, collect all the code with comments in a single SQL*Plus script, extend this with echoing and spooling commands (you are expected to have learned to do this in doing the tutorial in Appendix C), then run this extended script to produce a log file making sure that its filename instantiates the following template:

   `EX04-<studentID>.sql`

   where (here and elsewhere) you should replace `<studentID>` with your student ID, of course.

   Below, we refer to this as the *proof-of-work* file. In the case of this exercise, this file is also what we refer to, below, as the *for-insertion* file.

Assessment Type This activity is subject to **summative** assessment, therefore your submission will be marked and you will get the associated feedback.

The marks you obtain (see below) count for up to 35 % of your overall coursework mark for this course unit.

Marks There are 35 marks available.

- Marks are awarded for correct, complete SQL expressions/statements.
- Each correct, complete SQL expression/statement is worth five marks.
- If the marker finds that any of your submitted files is unreadable or (in the case of a script) not executable, you lose all the marks, so, test every one of them beforehand.
- If the marker finds that any of your submitted files is hard to read (e.g., poor or no indentation, or congested/overlapping items/lines, etc.) or is missing information (say, there is no cover page), you may lose marks up to 10% of the available marks.

Submission Procedure

1. Create a document file, with a cover page stating the course unit (COMP23111), the academic year (2016-2017), the exercise number (EX04) your name, and your student ID.
2. Insert the *for-insertion* file in the document you're preparing.

   Given that the *for-insertion* file is a SQL*Plus script, (i.e., code) you must make sure that it's typeset in a **monospaced font**[w] (e.g., Courier, Courier New, Lucida Console, Monaco, etc.)[11] so

[11] Consult the course staff if you're in doubt here.

that the indentation in your code is respected. You'll lose marks due to poor presentation if you don't.

3. Type in any assumptions and/or comments you wish to make.
4. Save/Export the document as a PDF file naming it as

   `EX04-<studentID>.pdf`

   Below, we refer to this as the *submission* file.
5. Submit both the *proof-of-work* and the *submission* files using Blackboard. If you fail to submit either file, your mark will be reduced by 1%. If you make the same mistake again, the penalty rises to 3%. Any further failure incurs a penalty of 10%.

Deadline/Extensions/Penalties

The deadline for submission is

17:00 on Friday in Week 11

Please, read the relevant section in the Introduction for this course unit's policy on extensions and penalties.

Example Class 5: Reasoning with Transactions (1 class)

Week 09

Goal The goal of this activity[12] is for you to practice the technique of reasoning with transactions.

This technique is often applied in real-world software development by database designers when they want to explore and analyse collections of transaction in order to understand, e.g., whether they could interfere with each other, thereby leading, potentially, to performance problems and even outright error.

Pre-/Co-Requisites Check the pre-/co-requisites in the dependency graph in Figure 5.

Tasks

1. Let transactions T_1 and T_2 be defined as follows:

```
T1 ≡
      read_item(X);
      X := X − N;
      write_item(X);
      read_item(Y);
      Y := Y + N;
      write_item(Y);
```

```
T2 ≡
      read_item(X);
      X := X + M;
      write_item(X);
```

   (a) What is the total number of possible schedules for T_1 and T_2?

   (b) List them.

   (c) Determine which are conflict serializable and which are not.

2. Let transactions T_1, T_2 and T_3 be defined as follows:

```
T1 ≡
      read_item(X);
      write_item(X);
      read_item(Y);
      write_item(Y);
```

```
T2 ≡
      read_item(Z);
      read_item(Y);
      write_item(Y);
      read_item(X);
      write_item(X);
```

[12] This is based on [EN13], with whom copyright rests. Errors are this author's responsibility.

T_3 ≡

```
read_item(Y);
read_item(Z);
write_item(Y);
write_item(Z);
```

(a) What is the total number of possible *serial* schedules for a set of N transactions?

(b) What is the total number of possible *serial* schedules for T_1, T_2 and T_2?

(c) List them.

Exercise 5: Writing SQL Stored Routines (1 lab session)

Week 12

Goal The goal of this activity is for you to practice the technique of SQL triggers and PL/SQL stored routines over a logical model in the form of a relational database schema from a conceptual model in the form of an **(enhanced) entity-relationship** (EER) diagram, such as you have practiced deriving from a data requirements specification (DRS) in a previous activity.

This technique is often applied in real-world software development by a database designer after there is agreement with the stakeholders that the conceptual model captures all the data assets of relevance to an organization's information systems.

The outcome of this step is a set of SQL triggers and procedures that comprise a subset of the DBMS-supported functionality with which the database designer would delegate application functionality to the DBMS. This is very useful in that, among other benefits: (a) it allows code to be hosted and managed by the DBMS, which relieves application programmers of, potentially, having to replicate such functionality in different components across many programs, (b) in doing so, it allows the DBMS to execute them faster, because we bring code to the data rather than bringing data to the code, thereby reducing communication costs, and (c) maintenance is simplified (one maintenance event in a centralized repository rather than potentially many such events in potentially many distincts repositories).

Pre-/Co-Requisites Check the pre-/co-requisites in the dependency graph in Figure 5.

Materials

THE ORINOCO DATABASE The database you will use is the complete version of the Orinoco database, which you have already seen in Exercise 2. The EER diagram that underlies it is reproduced here in Figure 13.

Recall that you mapped the conceptual model into a logical one, in the form of a relational schema, and submitted it in a previous activity.

If you didn't submit anything for that activity, now is the time to practice your mapping skills. If you find it hard, seek help from the course staff.

If you find yourself in trouble, the scripts are available as:

```
/opt/info/courses/COMP23111/create-Orinoco-tables-complete.sql
/opt/info/courses/COMP23111/populate-Orinoco-tables-complete.sql
/opt/info/courses/COMP23111/drop-Orinoco-tables-complete.sql
```

Tasks

1. Code[13] a SQL trigger that sets the value of the `duration` column when a tuple is inserted or updated in the `ContractInfo` table.

[13] If any of the specifications for these (or any other) coding tasks in this manual seem ambiguous to you, talk to the course

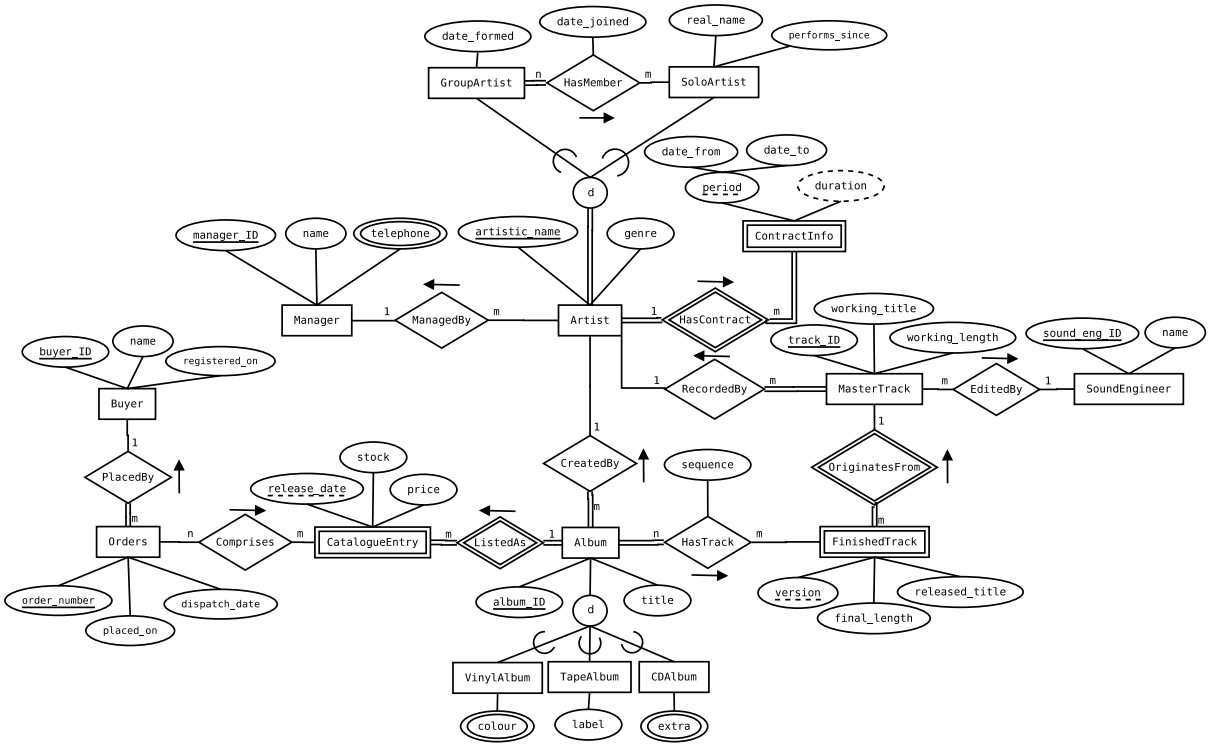

Figure 13: Orinoco [Complete]: Conceptual Model as an ER Diagram [same as Figure 9]

In doing this task, make sure that in your final submitted script you have a sequence such as follows:

(a) a SQL query that retrieves all the tuples in the `ContractInfo` table (so that you know the state before the trigger is fired)

(b) one or more SQL `INSERT` statements that cause the trigger to fire, followed by SQL queries that retrieve all the tuples in the table (so that you know the state after the trigger is fired)

(c) one or more SQL `UPDATE` statements that cause the trigger to fire, followed by SQL queries that retrieve all the tuples in the table (so that you know the state after the trigger is fired)

Note that, as with views and triggers, when creating a procedure, the procedure may already exist, in which case Oracle offers a `CREATE OR REPLACE` form of the statement that is the one you should always use for creating procedures.

Note also that, as with triggers, and for the same reasons as stated when we noted this in Exercise 4, you should add a single slash as the first and single character in the final line of a procedure.

As also already mentioned in Exercise 4, if you get an error on a trigger named `<triggerName>` (and, of course, you're likely to), you can get more information on what caused the error by using `SHOW ERRORS TRIGGER <triggerName>`. Likewise, for a PL/SQL procedure named `<procedureName>`, you can get more information on what caused the error by using `SHOW ERRORS PROCEDURE <procedureName>`

Again, while debugging, consider the need to take action to reverse the changes and how to do so.

2. Code a PL/SQL procedure that wraps the insertion of tuples in the `ContractInfo` table so as to make sure that the start and end dates of the new tuple do not lie between the start and end dates

staff, preferably in person (in an examples class or a lab session, or in the open office hour for this course unit) to clarify it. If you can't talk face-to-face, use email, but, please, don't just ignore it as you may misinterpret the task and lose marks as a result.

of any contract already present in the table for the given artist, and that the from and to dates are in the correct temporal order. The procedure takes as input the name of the artist whose contract this is, the start date and the end date of the contract.

Note that the `BETWEEN` operator in SQL may come handy here (so, look it up in the Oracle online documentation).

Note also that if you only need to test that a relation R violates a condition θ then a `COUNT(*)` query with R in the `FROM` clause and θ in the `WHERE` clause precisely expresses your need.

3. Code a SQL view, named `AlbumDistribution`, that retrieves album information along with a new attribute called `is_distributed_as` to store information as to whether an album can be distributed as a CD, on vinyl, or as a cassette tape (e.g., the values this attribute would take would be `'c'`, `'v'`, or `'t'`, respectively).

4. It is often the case that a buyer will wish to see a list of tracks on an album in the order of the play list. Using the `AlbumDistribution` view, code a PL/SQL procedure takes as input parameters the album type (e.g., `'c'`, `'v'`, or `'t'`) and the album title. Very importantly, in the case of the album title, your procedure must allow the caller to pass wildcard sequences, (e.g., `'%e%'`, or `'%t%'`). Finally, your procedure must produce a formatted listing, with the `sequence` followed by a comma, then the `title` of the `track` followed by a greater-than character (i.e., right angle-bracket), followed by the `album-title`.

   Remember that, in SQL*Plus, you will need to set the server output on to see the printed lines in your screen (and in your spool file).

   In the coding tasks above, make sure that in your final submitted script you have a sequence of statements that shows the state before and after execution for both valid and invalid cases, so that you demonstrate the capabilities you have captured.

5. After you have debugged your work, collect all the code with comments in a single SQL*Plus script, extend this with echoing and spooling commands (you are expected to have learned to do this in doing the tutorial in Appendix C), then run this extended script to produce a log file making sure that its filename instantiates the following template:

   `EX05-<studentID>.sql`

   where (here and elsewhere) you should replace `<studentID>` with your student ID, of course.

   Below, we refer to this as the *proof-of-work* file. In the case of this exercise, this file is also what we refer to, below, as the *for-insertion* file.

Assessment Type This activity is subject to *formative* assessment, therefore your submission will not be marked but you will get the associated feedback.In other words, for this activity, your submission does not contribute to the overall coursework mark for this course unit.

Submission Procedure

1. Create a document file, with a cover page stating the course unit (COMP23111), the academic year (2016-2017), the exercise number (EX05) your name, and your student ID.

2. Insert the *for-insertion* file in the document you're preparing.

Given that the *for-insertion* file is a SQL*Plus script, (i.e., code) you must make sure that it's typeset in a **monospaced font**[w] (e.g., Courier, Courier New, Lucida Console, Monaco, etc.)[14] so that the indentation in your code is respected. You'll lose marks due to poor presentation if you don't.

3. Type in any assumptions and/or comments you wish to make.
4. Save/Export the document as a PDF file naming it as

   `EX05-<studentID>.pdf`

   Below, we refer to this as the *submission* file.
5. Submit both the *proof-of-work* and the *submission* files using Blackboard.

Deadline/Extensions/Penalties

The deadline for submission is

17:00 on Friday in Week 12

Since this activity is only subject to formative assessment, extensions and penalties are not applicable.

[14] Consult the course staff if you're in doubt here.

A Data Requirements Specifications from Data Flow Diagrams

A.1 Introduction

In order to design a conceptual data model (as part of a software development project), the first step is to produce a data requirements specification (DRS).

Here, due to the limitations inherent in an educational environment, we must assume that a DRS is derived from a process model, rather than independently, by database designers, through a process of face-to-face interaction with stakeholders. In doing so, we assume that it is the systems analysts that have engaged in such interaction and that, as data analysts/database designers, we are operating downstream from the process modelling phase, in terms of the software development life cycle.

Thus, we assume that in order to produce a DRS, the database designers work from a process model and, if necessarily, elicit further clarification from the systems analysts who generated the latter from face-to-face contact with stakeholders.

These notes briefly introduce the notion of process models that are represented as data flow diagrams (DFDs) using the Gane-Sarson notation. Our goal here is to explain the syntax and semantics of DFDs with the specific purpose of allowing the reader, who is assumed to be a database designer, to extract from one such diagram a DRS, from which, in turn, a conceptual data model can later be derived.

A.2 Interpreting a Process Model

A **process model** is an abstract representation of a system centred on its functions, also referred to as processes, and how they capture, transform, store and distribute the data among the components of the system, as well as across the latter's boundary and into the external environment.

A **data flow diagram** (DFD) depicts a process model in graphical form. A DFD is a graph with three types of nodes and one type of edge. The node types denote *functions*, *data stores* and *external entities* (also referred to as sources or sinks). External entities can be real agents (e.g., people, and organizations) in the real world or other processes that comprise a larger process of which the one described by the DFD is itself a component. Edges are directed and denote the *flow of data* from one node to another (possibly more than one in either case).

Nodes and edges are labelled. An edge label is a noun (or noun phrase)[15] denoting the data item that is incoming or outgoing from a node. In the case of external entities, the label is a noun (or noun phrase) that indicates the agent, or system, that it represents. In the case of functions, the label is a verb phrase [16] that indicates how the function transforms the incoming edges into the outgoing edges. In the case of data stores, the label is a noun (or noun phrase) that indicates (and often generalizes on) what data items flow into and out of the data store.

[15]The slightly-adapted Wikipedia definition of noun is: "A **noun** is a word that functions as the name of some specific thing or set of things, such as living creatures, objects, places, actions, qualities, states of existence, or ideas." One example is the word *cat*, denoting the set of animals of the family *Felidae*. Another example is the word *mat* denoting the set of things that are flat pieces of coarse material one uses, e.g., to wipe one's feet on entering a room. Other examples are *customer*, *account*, etc. The slightly-adapted Wikipedia definition of noun phrase is "A **noun phrase** is a phrase based on a noun [...] optionally accompanied by modifiers such as [...] adjectives." An example noun phrase is *lazy cat*: it's made of the noun *cat* modified by the adjective *lazy*. Other example noun phrases are *new customer* and *savings account*.

[16]The slightly-adapted Wikipedia definition of verb is: "A **verb** is a word that conveys an action, an occurence, or a state of being." Examples of actions are *bring*, *read*, *walk*, *run*, *learn*, *sit*, *register*, *send*. Examples of occurrences are *happen*, *become*. Examples of state of being are *be*, *exist*, *stand*."Without going too technical, "a **verb phrase** is a phrase based on a verb accompanied its adjunct, which is often a noun or noun phrase, optionally accompanied by modifiers such as adverbs." An example verb phrase is *sits on a mat*: it's made of the verb *sits on* accompanied by its adjunct *a mat*. Other example verb phrases are *register new customer* and *open savings account*.

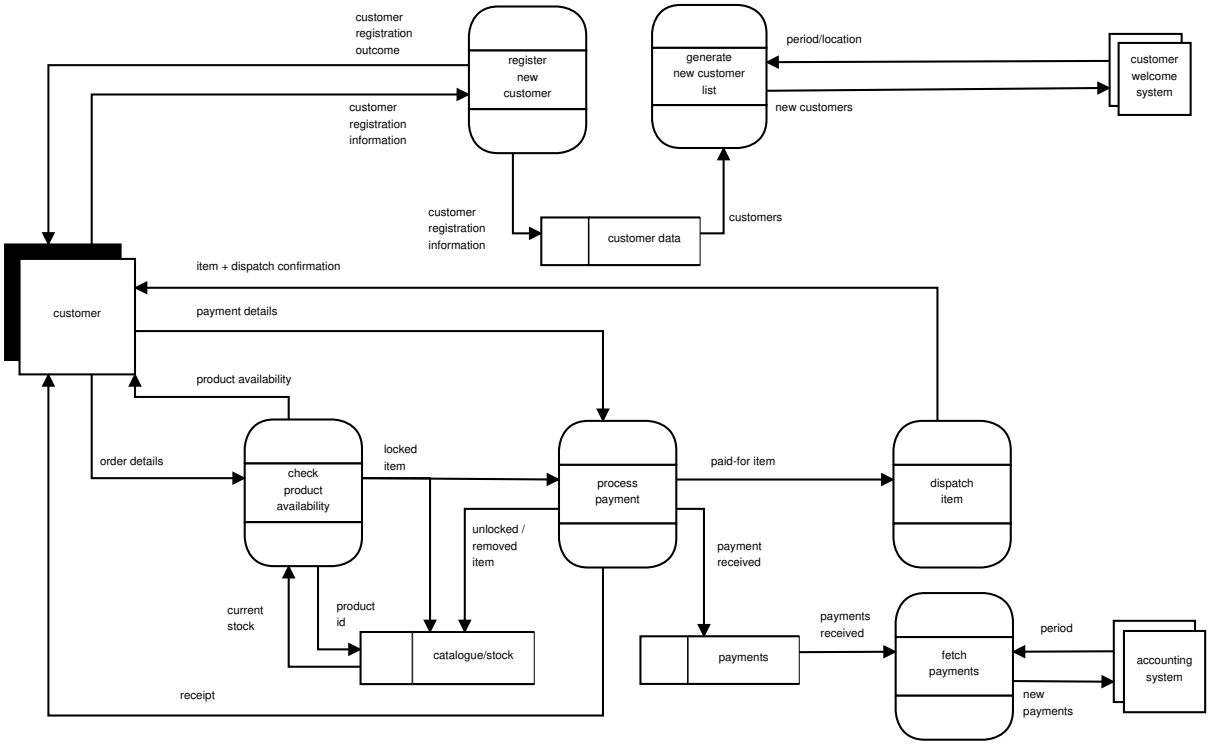

Figure 14: An Example DFD for (Part of) an E-Commerce System

Consider the example DFD in Figure 14, depicting a very small part of an e-commerce system.

In this example, one type of external entity is a customer. Another, is the customer welcome system, which, for example, sends a printed welcome pack containing information about terms and conditions, and so on.

In this example, at the top half of the DFD, there are two functions: one is to register new customers, the other is to generate a new customer list for the benefit of the customer welcome system.

The register new customer function has an incoming edge from the customer external entity labelled customer registration information and two outgoing edges: one, labelled customer registration information, into a data store labelled customer data and another, labelled customer registration outcome, into the customer external entity.

The generate new customer list function has two incoming edges and one outgoing edge. One incoming edge from the customer welcome system makes available the parameters that decide who goes into the list of new customers in terms of their location (say, Manchester) and the registration date (say, in the last calendar month). The other incoming edge, from the customer data store, indicates that the function (potentially) processes all existing customers whose information is held in the customer data store. The outgoing edge into the customer welcome system is the list of new customers in the requested period and location.

The bottom half of the DFD is more complex, as it covers the sale process itself, from an order to payment receipt and item dispatch. Make sure you can read the notation and understand the process being modelled, but be careful not to allow yourself to impose on it your own intuitions as what is the right or wrong way to do it. This is important in practice, but not here, not now.

The best intuition about DFDs is that they represent how data, stemming from external entities acting as sources, enters the process, is transformed by functions, possibly resting in data stores at different stages of the overall flows, and eventually leaves the process, destined to external entities acting as sinks.

C1 Nodes and edges are identified by their name, so if a node or edge appears more than once in a DFD, typically to avoid graphical congestion and confusion, then it is the same node or edge. In the case of nodes, it is customary to draw a mark (e.g., a little diagonal line in the top-left corner) to indicate that this node also appears elsewhere.

C2 No function can have only inputs.

C3 No function can have only outputs.

C4 Data cannot directly flow from one data store to another, or from an external entity to another, or from an external entity into a data store, or from a data store to an external entity: only a function can move data, or for one function to that same function: in all such cases, an explicit function must mediate.

C5 A function transforms inputs into outputs, therefore the incoming edges must be different than the outgoing ones. If a function merely retrieves data (e.g., call it D) from a store S in order to distribute it to an external entity E without transforming, then we qualify, e.g., retrieved-D-from-S-for-E. Likewise, if a function merely obtains data from an external entity in order to pass it on to a function F or to store it in a store S, then we qualify, e.g., obtained-D-from-E-for-S (or F, if that is the case).

C6 A data flow is never bidirectional: two flows must be drawn and qualification is used to distinguish them.

C7 One data flow may fork into n flows, with the semantics the collection is identically replicated n times, one for which destination.

C8 n data flows may join into one flow, with the semantics that the different collections are merged in a non-specified manner (e.g., by set, or bag, union).

C9 A flow into a store has general update semantics (i.e., create, replace, update or delete).

C10 A flow out of a store has the semantics of retrieve (or fetch) without modification.

Figure 15: DFD Well-Formedness Constraints

Notice that the edges denote collections of data, not individual records, or data items. These collections can, potentially, be singletons, but are always thought of as collections. This implies that data store operations are assumed to be collection-based and that a function is conceived of as operating on an entire collection of items (i.e., it would not be inappropriate to think of functions as iterative control structures, or, possibly, recursive computations).

Notice that a data store represents data at rest, in the sense that the flow from one function to another is interrupted, or decoupled in time, by placing the data in the store.

Notice, finally, that a DFD has a logical boundary defined by its external entities. This means that, from the viewpoint of modelling the process that the DFD represents, we are unconcerned with that use was made of the inflowing data as well as what use is made of the outflowing data. So, we do not consider any interactions between external entities and, it also follows from the fact that they lie outside the process boundary that they are not allowed to act on stored data directly: there must be functions that mediate the interaction of external entities with the stored data.

One can draw DFDs at different levels of abstraction but the most crucial ones for software system development are those referred to as **level-0 DFDs**, which define how a particular system performs the major functions of an information system. So, a level-0 DFD defines the external entities that act as sources of data and how to capture data from them as well as also defining the functions that transform and distribute the resulting data products to external entities that consume them, possibly whilst requiring data to rest in data stores for particular purposes and reason (e.g., time decoupling, sharing, etc.).

Level-0 DFDs capture the primary functions in a process, i.e., those that are most essential to articulate the fundamental nature of the information system, and, in an organizational context, the strategy for adding value to the inputs and generate outputs that allow the organization to deliver what its stakeholders need and, hence, compete strongly in its target market.

It is important to bear in mind that DFDs are not meant to model timing, therefore it is best to draw, and interpret, DFDs as capturing a process that is in constant flow, and has never started nor will ever end.

For a DFD to be well-formed the constraints in Figure 15 must hold.

In process modelling, top-down stepwise refinement is typically used, implying that each function in a level-n DFD can itself be refined into a level-$n+1$ DFD. In which case, several constraints apply which we do not discuss here, since, in these notes, we are only concerned with interpreting a level-0 DFD (see [HGV08], Ch. 7).

A.3 Deriving a DRS from a Level-0 (New, Logical) DFD

Recall that our specific purpose in these notes is to explain to the reader, who is assumed to be a database designer, how to derive a DRS from a DFD, from which, in turn, a conceptual data model can later be derived.

In software development, we often develop logical models and physical models, where the latter abstracts from the implementation details that is the purpose of the latter to capture. Also, for each of logical and physical models we often develop one model of the current system (if any) and another model of the new, desired system.

For the purpose in hand, i.e., deriving a DRS from a DFD, we will assume we have a logical level-0 DFD of the new, desired system.

A.3.1 What does a DRS captures?

Given a logical level-0 DFD of the new, desired system, the derivation of a DRS depends on obtaining answers to questions that elicit the necessary information that goes in a DRS. In order to motivate the questions, we now consider in more detail what a DRS captures.

A DRS aims to explicitly record (among other kinds of information of less relevance here) the information characterized by the questions in Figure 16.

I1 What entity types (or classes of objects) will have data about them stored?

I2 What subtypes or supertypes or composite types or union types of entities of interest are of interest themselves?

I3 What relationship types there exist between the entity types of interest?

I4 Is the participation of every entity of a certain type in a given relationship type mandatory or optional (i.e., can, or cannot, there be an entity in the store that does not participate in a relationship of that type)?

I5 When an entity of a certain type participates in a given relationship type, does it do so once only or multiple times?

I6 What attributes characterize the data that is stored about entities or relationships of a certain type?

I7 Is any of an entity's attributes identifying and, if so, which?

I8 Is the scope of identification of an identifying attribute global (i.e., unique with global scope) or partial (i.e., dependent on the identifying attribute of some other entity, meaning that it needs the latter to become globally unique)?

I9 Is any attribute is multivalued?

I10 Is any attribute computable from other attributes?

I11 Is any attribute complex, i.e., composed of other, component, attributes?

I12 What is the type of each attribute and what rules, if any, constrain the values it can take?

I13 What constraints, if any, characterize the valid states of the stored data and hence must be satisfied at all times?

I14 Does the history of any data item need to be recorded?

Figure 16: The Information Content of a DRS as a Set of Questions

A.3.2 How does a DFD help elicit a DRS?

A data analyst would use the logical level-0 DFD of the new, desired system that has resulted from process modelling as an interface between her and the stakeholders to carry out the first step of data modelling, viz., eliciting a DRS.

In doing so, a data analyst would ask the kind of questions illustrated in Figure 17. Note that they are closely related to those in Figure 16 (though not all questions in the earlier figure are represented in the

later one). However, the data analyst uses the logical level-0 DFD of the new, desired system to phrase them in terms that are closer to the way the stakeholders view the workings of their organization.

A.3.3 A Concrete Example: Using the DFD to Ask the Right Questions

Now, consider again the simple e-commerce system in Figure 14. Example questions that a data analyst may be prompted to ask when we're guided by that level-0 DFD and her common-sense intuitions are shown in Figure 18. These are keyed to the question types in Figure 17, but, note that, here, we don't have a real stakeholder to interview, so we'll assume some likely answers based on common knowledge, for the sake of practicing formulating these kinds of questions. Note, also, that the examples below are not exhaustive, the example questions do not systematically traverse all of the DFD in Figure 14: in real software development, you would do so, but then, the interaction and reflection processes that go into the generation of a complete DRS typically take in the order of weeks or months.

A.3.4 A Concrete Example: Transforming the Answers into a DRS

Figure 19 illustrates how a data analyst would use the answers obtained from the stakeholder to derive a DRS for the new system. This DRS, in turn, would be the basis for designing the database to support the new system, as we shall explore later in this course unit.

A.4 Practice Tasks

Consider the example level-0 DFD in Figure 20[17], depicting a simple web store front-end system.

1. Using Figure 20 as your reference, and Figure 18 as an example to guide you, write down one or more questions of each of the types (i.e., **Q1** to **Q8**) in Figure 17.
2. Using Figure 18 as an example to guide you, provide common-sense, intuitive answers (that conform with Figure 20) for the questions you wrote down in response to the previous task.
3. Using Figure 19 as an example to guide you, write down the DRS items that follow from the answers you gave in response to the previous task.

[17] This diagram is from [HGV08], with whom copyright rests.

Q1 For each data flow, what is inside it? Information about a stakeholder (e.g., an account holder)? Is it perhaps a product (e.g., insurance) or service (e.g., currency exchange) bought or sold? Or is it a document (e.g., the record of a transaction in an ATM)? Must this information be kept stored? Or is it used and discarded?

Q2 For each identified stakeholders, products, services, documents, etc., does it have supersets or subsets that we take an interest on? For example, the various products (i.e., current accounts, savings accounts, insurance, investment funds, etc.) can be grouped as types of account since they share some common properties (e.g., an opening date, a balance, etc.) while having specific properties of their own (e.g., current accounts have an interest-due rate for overdrafts, savings accounts have an interest-paid rate, etc.). As an example of subsetting, some investment accounts are on stocks, others on commodities, and so on.

Q3 Do the identified stakeholders, products, services, documents, etc., have a unique, strong, global identifying property? For example, in the UK, a person typically has a National Insurance number that is a unique, strong, global identifying property of that person. In contrast, an address is only possibly unique if one takes the postcode and adds the house number. Also, a flat number is unique only in the context of a given block, i.e., it's not a strongly, globally identifying, it's only weakly, partially so because it must be concatenated with the identifying property of the building before we be certain which flat is being referred to.

Q4 For each stakeholder, product, service, document, etc., what information about it is flowing (e.g., for a service like currency exchange, these could be the source currency, the target currency, the exchange rate, the amount, the date and time, the source and target accounts, etc.)?

Q5 Which, if any, of these properties composite (i.e., made of component properties, like an address might be composed of house number, street name, and flat number)? Which, if any, is multivalued (e.g., a block of flats may have several entrances, e.g., front, left, right, back)? Which, if any, is derivable from others (e.g., the total insurable value of a building is the sum of the insurable values of each flat in it plus the insurable value of the common areas)?

Q6 For the identified stakeholders, products, services, documents, etc., how do they relate to each other? For example, an account holder contracts insurance.

Q7 For each such identified relationship type, must it always hold for each stakeholders, products, services, documents, etc.? For example, is it the case that every account holder must have contracted an insurance product, meaning that unless some insurance is contracted accounts cannot be held with the company, or is it instead the case that someone can hold an account without contracting any insurance? In the other direction, is it the case that every insurance product must have some account holder that has contracted it, or is it the case that there can be insurance products that have no contractors yet?

Q8 For each identified relationship type, can my stakeholders, products, services, documents, etc. participate in it? For example, is it the case that an account holder can contract many insurance products or just one? In the other direction, is it the case that an insurance product can be contracted by many account holders or exclusively by one account holder only?

Figure 17: Some of the Questions a Data Analyst Asks so as to Derive a DRS

- [**Q1**]: Must information about customer external entities be stored?

  A: Yes, we store data about our customers.

- [**Q1**]: Is the customer data data store just a temporary resting place in the flow of data or does it really constitute a data asset of the organization?

  A: It is a data asset for us.

- [**Q2**]: Are there different types of customer? For example, premium and freemium?

  A: No, there aren't different subsets of interest among our customers.

- [**Q3**]: Is there some piece of information that distinguishes one customer from all the others?

  A: Yes, when a customer registers, we assign it a customer id number whose uniqueness we enforce rigorously.

- [**Q4**]: What data do you want to hold about a customer?

  A: Besides the customer id, the password, the name, the address, the date of birth, the age, the phone numbers.

- [**Q5**]: Are any of these composite, i.e., made of parts?

  A: Yes, we break down the name into first name and last name. Also, we break down address into postcode, house number, and flat number.

- [**Q5**]: Do I take it that you store more than one phone number for a customer?

  A: Yes, we do.

- [**Q5**]: And do you differentiate between, say, landline and mobile numbers?

  A: No, we don't.

- [**Q5**]: Am I right in assuming that we can compute the age of the customer on any given date using the date of birth?

  A: Yes, I suppose you are right.

- [**Q6**]: How would you describe the sales process?

  A: A customer makes an order. The order list the products of interest, which are checked for availability. If the products are available, the customer sends us payment details. These are then processed, so that a receipt reaches the customer. Paid-for items are then dispatched to the customer with a dispatch confirmation.

  [**Q6**]: Can I confirm the relationships and/or documents I have identified? The relationships are: customer makes order, order lists product, customer sends payment, product is prepared for dispatch, customer receives product.

  A: This seems right, in a nutshell.

  [**Q6**]: Am I right in assuming that, therefore, over and above data about customers, you want to hold data about products in a catalogue, orders and payments received and items dispatched?

  A: Yes, this is correct.

- [**Q7**]: Now, going back to the customer makes order and the order lists product relationships. Am I right in assuming that a customer can have many orders in your data stores but each order can only refer to a single customer, and also that a order can list many products and that each product can be listed in many orders?

  A: Almost all of it is right. The unusual detail is that we actually allow an order to refer to more than one customer.

- [**Q8**]: Now, let stay with the customer makes order and the order lists product relationships. Am I right in assuming that customer data is kept even if the customer may not have made any order, but that an order must have been made my some customer, and also that a order must list at least one product and but that product data is kept even though it may not have been listed in any order?

  A: Yes, you got all of it right this time.

Figure 18: Data Analyst Questions Guided by Figure 14

R1 There is a need to store data about each customer.

R2 There are no sub- or supersets of interest for customer.

R3 The identifying property of customer is a company-assigned customer id.

R4 Besides the customer id, a customer has the following data about it stored: password, name [composite of first name and last name], address [composite of postcode, house number, flat number], date of birth, age [derived from date of birth, the phone numbers [multivalued].

R5 *Here, by analogy with customer, there would be data requirements specs for order, product, payment, dispatch. In what follows, we assume they exist.*

R6 A customer makes an order. A customer can make many orders but there may be a customer that has not made any orders. An order can be made by many customers and there may not be an order that has not been made by some customer.

R7 An order lists a product. An order can list many products and a product can be listed in many orders. An order must list a product but a product may not have been listed in any order.

R8 *Here, all the other relationships would have been considered.*

R9 *Also, the data analyst would consider whether more needs to be recorded about each relationship, e.g., the date an order was made, the credit/debit card number used to make a payment, etc.).*

R10 *Finally, we're omitting here the many constraints that data needs to obey, from valid values (e.g., of dates, or postcodes, etc.) to business rules (e.g., that there is a day-limit on the total amount charged of a single credit/debit card number).*

Figure 19: (Partial) Data Requirements Specification from Figure 14

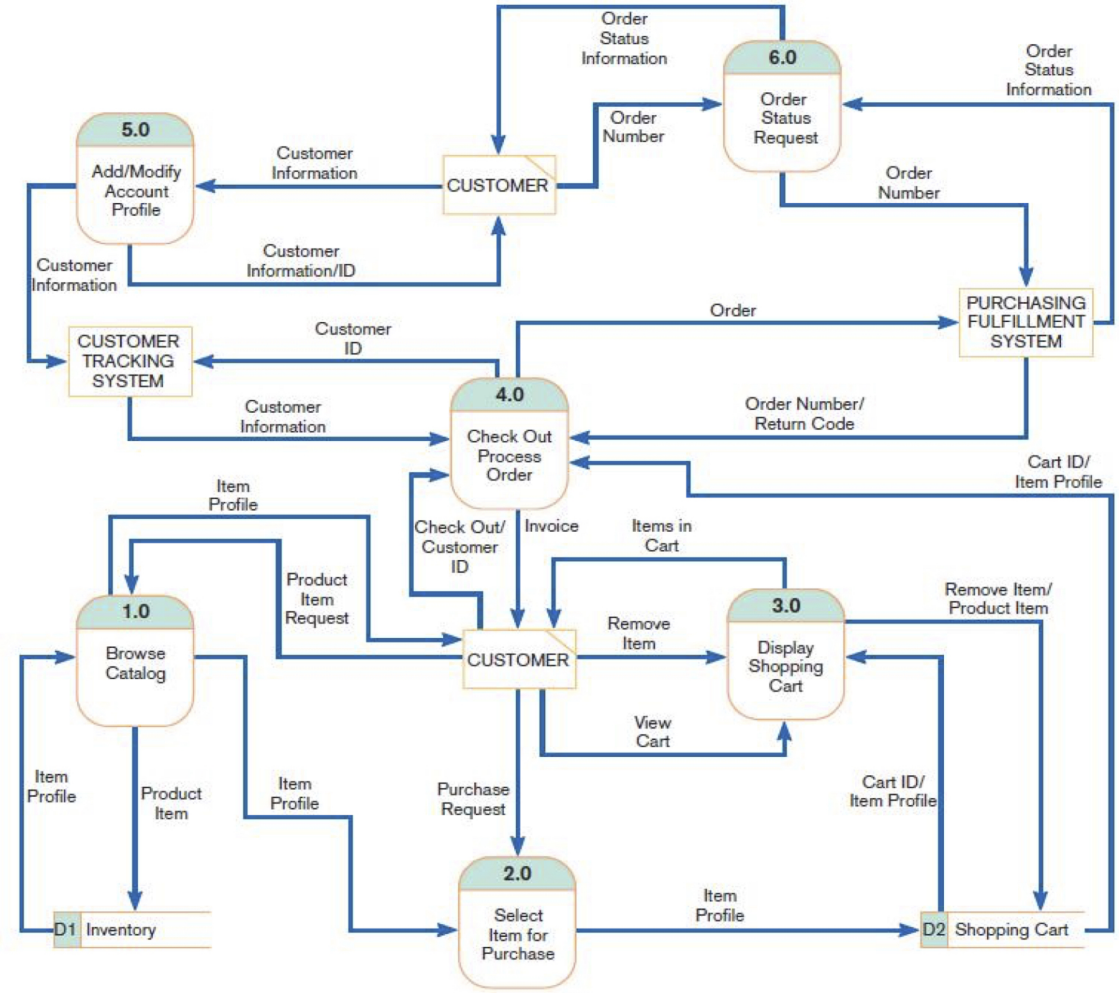

Figure 20: A Level-0 DFD for a Web Store

B Using the Dia Diagramming Tool for ER Modelling

B.1 Introduction

The goal of this learning activity is to provide you with an opportunity to interact with the Dia program for drawing structured diagrams.

You will be asked to reproduce an (E)ER diagram using Dia, just to ensure that you become familiar with its support for (E)ER modelling.

B.2 Overview

Dia is a Linux application for creating technical diagrams, whose interface and features are loosely based on Visio, the Microsoft Windows program.

Dia is easy enough to learn without much hassle and flexible enough to make power users feel at home when compared to commercial tools, but it has some limitations too.

B.3 Interacting with Dia

The first thing to do with Dia is to know where to find the Dia documentation →Read online. Then, you should play with the tool and familiarise yourself with it.

To launch Dia from a terminal/shell window in Linux, just enter `dia` (you'll normally want to add an ampersand to run it in the background).

This should open two windows[18], as follows.

Toolbox Window The first window is a detached tool/command box. It has the usual facilities from drawing lines, shapes, etc. Just beneath the 4x4 button grid, you find a pull-down. This allows you to select the graphical formalism you want to use. You should select `ER`.

With `ER` selected, you see five buttons, corresponding to entities (in two forms), attributes, and a double line icon you will use to draw edges with certain decorations.

At the bottom of the toolbox make sure you select line types that are solid and without arrowheads, since most of the time this is what you will want in drawing ER diagrams.

Drawing Window The second window is where you really draw the diagram.

The basic sequence of actions is to clink on a shape (say, for entity) in the toolbox, then click on where you want to place it in the drawing window, which causes the shape to be drawn, then double click on the drawn shape so that you can enter its properties (e.g., the label/name, etc.). Of course you can highlight than drag shapes around.

For edges, give preference to the button with two parallel vertical lines. Again, click on the toolbox button, then click on the best control point (these appear as light marks around the shape) of the source shape, which causes the edge to be drawn. (Notice that it snaps to a control point so that if you move the shape later, the edge moves with it.) You can then drag the other end to the best control point in the

[18] Sometimes the two windows discussed below are merged into one window, in which case, adjust the instruction to refer to areas of the window rather than independent windows.

destination shape (again, it snaps). Use your intuition as to how you do things (e.g., highlight a region of the diagram and drag it) and try to see if it works.

Toolbox buttons Now, consider the example ER diagram in Figure 21. While you may not yet have covered ER notation in the course unit, you should try to have a go at simply reproducing it.

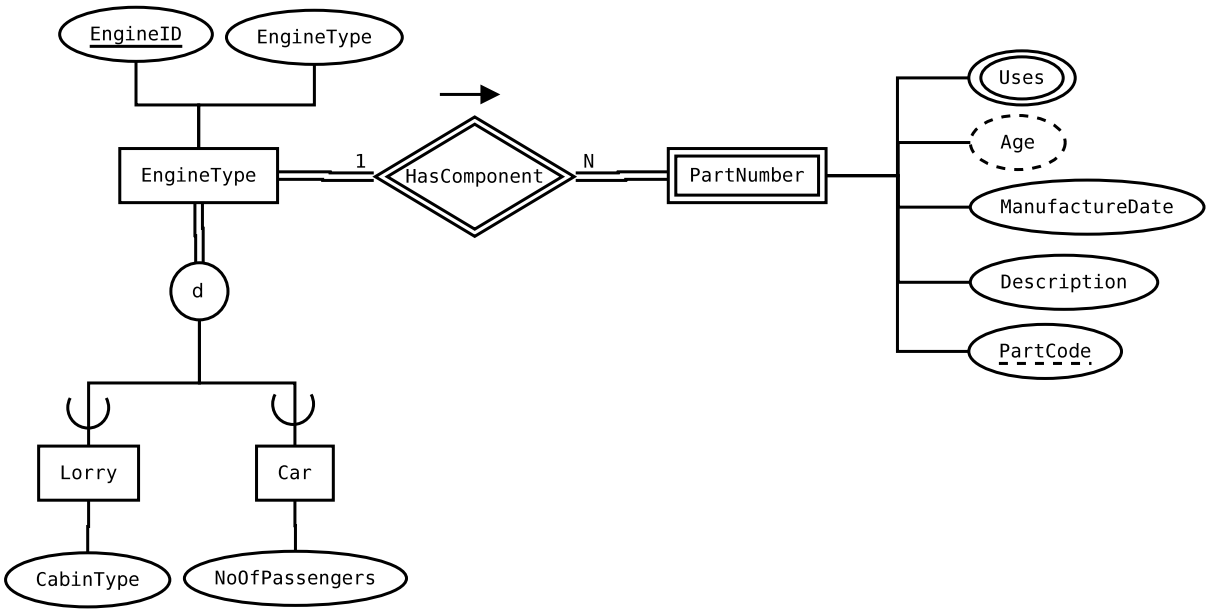

Figure 21: Example (Partial) ER Diagram for an Auto Manufacturer

In the toolbox, there are two kinds of rectangles: one labeled `E` and drawn with a single-line and another, again labeled `E` but drawn with a double line. The former is for drawing (strong) entity types, like `EngineType` in the example; the latter is for drawing weak entity types, like `PartNumber` in the example.

The lozenge (i.e., the diamond shape) is for drawing relationship types, like `HasComponent` in the example. When you draw a relationship you have the opportunity to (double-click and) enter the left and right cardinalities (e.g., in the example, `HasComponent` is `1` on the left side, that of `EngineType`, and `N`, on the right, the side of `PartNumber`.

The ellipse (i.e., the oval shape) is for drawing attributes, like `EngineID` or `Age` in the example. There are properties you need to enter. The default is a normal attribute (like `Description`. Some are key attributes (e.g., `EngineID`) and appear underlined with a solid line. Others are weak (or partial) keys (e.g., `PartCode`) and appear underlined with a dashed line. Others still are derived attributes (e.g., `Age`) and in this case the oval is drawn with a dashed line. Finally, others are multivalued attributes (e.g., `Uses`) whose oval is then drawn with double line.

Now, when you draw edges, you can alter the property called *mandatory* to make Dia draw a double line, as is the case for both sides of `HasComponent` in the example.

Making do Dia doesn't have explicit support for the additional constructs in the `Enhanced` ER model, such as specialization and generation hierarchies. So, we make do.

To see how, consider that `EngineType` has two specializations, i.e., `Lorry` and `Car`. The circle that denotes specialization/generalization is not part of the ER-specific buttons, so we use the one in the 4x4 matrix above the pull-down. But how we add the little `d` inside. Well, we use the text button and adjust

the position.

Then, what about the *contains* (or *subset*) symbol from set theory (i.e., ⊂)? One way to do it is to take the arc button (in third row top to bottom, third column left to right) and draw it. By adjusting the end points we can, just about, imitate the ⊂ symbol. We can then rotate the shape before placing it in the correct position over the desired edge (as was done in the example above).

And then, what about the arrow on top of the relationship type which tells us in which direction to read it? Well, as an exception to the preference for the parallel-line button in the ER template, here you should use the straight-line button and double-clicking on it add the arrowhead in the appropriate extremity.

B.4 Practice Tasks

1. Draw the diagram that appears in the SQL*Plus tutorial [Appendix C, Figure 23].
2. Draw the diagram in Figure 22[19].

But note that, in both the above cases, for drawing the edges between an entity type and its attributes, the straight-line button was used (thus drawing slanted lines more easily) rather than the parallel-line button in the ER template, which, in that example, was only used for the relationship edges.

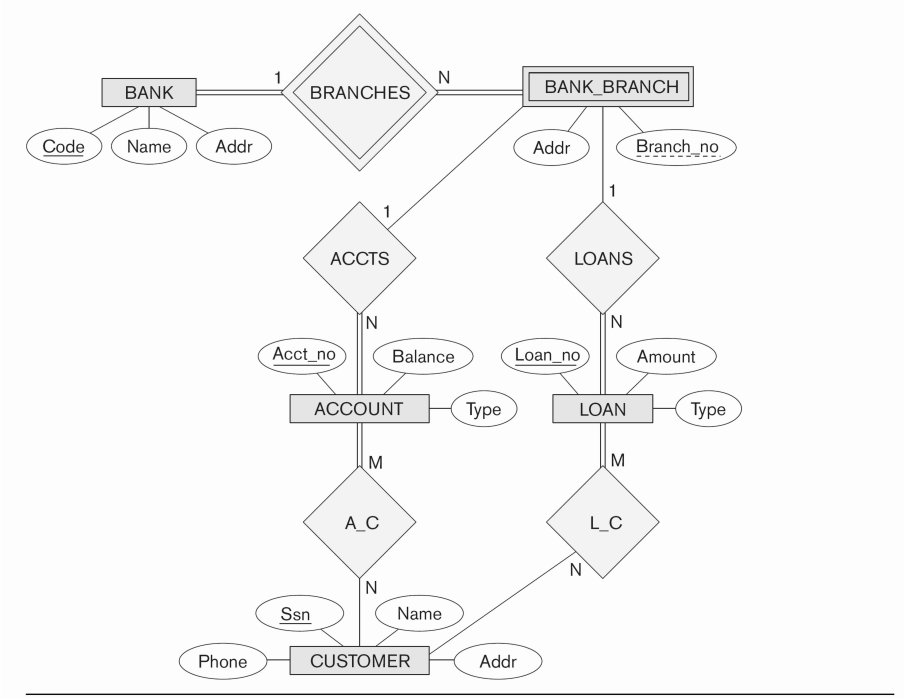

Figure 22: An ER Diagram for an Bank

This is the end of this Dia tutorial. Do explore further on your own, at your own time.

[19]This diagram is from [EN13],with whom copyright rests.

C Interacting with SQL*Plus, the Command Line Interface to Oracle

C.1 Introduction

The goal of this learning activity is to provide you with an opportunity to interact with the Oracle DBMS using Oracle SQL*Plus, the command-line interface provided by Oracle.

Oracle usernames and passwords You will be using the Oracle DBMS under Linux (and, in particular, its command-line interface, Oracle SQL*Plus) in lab exercises (as well as for self-learning). For that, you will need an Oracle username and password.

We assign these to you. They are then sent to you using your University student email. If you have not received yours by the start of Week 3, email both lecturers to let them know (their emails are available in the course unit webpage).

Online documentation The online version of the Oracle SQL*Plus reference manual that you should bookmark and use to clarify your doubts in this course unit is:

> **SQL*Plus®User's Guide and Reference** →Read online

The online documentation for SQL as implemented by Oracle is in:

> **Oracle Database SQL Language Reference** →Read online

Error messages can be searched for online in:

> **Oracle Database Error Messages** →Read online

Online books for support Two books to which you have right of access as a University of Manchester student may also help you. These can be downloaded in their entirety, or chosen chapters. You shouldn't need to download or read any of the two in their entirety. They are: [Haa+14] →Download and [Mor+13] →Download .

C.2 Overview

You will be given SQL*Plus scripts that allow you to create/populate/drop some tables from a case-study database and populate them with tuples. Using SQL*Plus, this will then allow you to explore how SQL is implemented by Oracle.

You will gain a basic understanding of the data dictionary component of the Oracle DBMS and will take the initial steps in using the data definition and manipulation aspects of SQL, as implemented in Oracle, as well as basic query forms.

Since you're doing this as self-study, if you encounter difficulties that cannot be solved by learning from the online documentation and the online books, try to push forward into the tutorial but do make a note of questions so that you can ask the course unit staff at the earliest opportunity.

C.3 The Orinoco Case Study

The learning activities in this course unit use several, different short scenarios from different application domains.

This learning activity uses a hypothetical case study, the preamble of which is as follows:

> *Orinoco is a medium-sized record producer that operates on niche markets and offers music on CDs, tape, and vinyl. The firm has recently bought a number of smaller recording companies and is planning to expand.*
>
> *There is a great deal of information that remains available only in the brochures that Orinoco produces, along with an accompanying price guide. Some information is stored on electronic files on an ad hoc basis. This situation has now become very cumbersome as such files lie in different computers, leading to inconsistency and mutual incompatibility. As a result, from the point of view of the software applications that the company aims to develop to underpin its expansion plans, the Orinoco data assets seem incomplete and are difficult to exploit to the full (particularly due to the great heterogeneity injected by the lack of proper assimilation of the data assets that came with the recent takeovers). For example, it is particularly difficult to find music that matches a customer's specific requests unless brochures are browsed carefully, which is time-consuming and error-prone.*
>
> *The management at Orinoco have decided to computerise the information in their brochures, to reconsider the data stored in files, to include purchase information, and to sell directly over the web. There is a need, therefore, to make it easier for software applications to:*
>
> - *Gather specific information from customers and answer questions about products*
> - *Check the availability of specific products*
> - *Give costs of products*
> - *Keep information about artists, albums, and tracks*
> - *Keep information about the back catalogues of the companies recently taken over*
> - *Keep contract information about artist signings etc.*
>
> *You may assume that the system is stand-alone and does not have to integrate with any other. For the purposes of this course unit, other simplifying assumptions have been made (e.g., that price information is fixed and is not flexible, etc.).*
>
> *The following data requirements have been elicited:*
>
> - *Each artist has an artistic name, works within a musical genre and is managed by a manager who has an ID, a name and can have several contact telephone numbers.*
> - *Artists also have a contract with Orinoco, for which the following information is recorded: the period of the contract in terms of the date from which, as well as the date up to which, the contract holds. The duration of the contract in days is also stored.*
> - *An artist records a number of master tracks, each of which has an ID, a working title, a working length. Information on the artist who recorded it and the sound engineer who edited it are also kept.*
> - *A finished track originates from a master track and has a version number (as there may be different versions of it), a released title and a final length.*
> - *A sound engineer has an ID and a name.*

- *An album groups together finished tracks into a sequence (i.e., a play list). An album has an ID, a title. Information is kept about the artist who created it.*

In this initial model, we are not including information that links an album with the finished tracks it contains. We will do later in this course unit, when we will also store information as to whether an album can be distributed as a CD, on vinyl, or as a cassette tape.

At this early stage in the course unit, you may not have been taught yet how a database design (in the form of, first, a conceptual schema, and then, a relational schema) is derived from a data requirements specification, but this tutorial is just about your getting familiarized with one specific tool to interact with the Oracle DBMS, viz., the SQL*Plus command-line interface. Therefore, we will tell you the conceptual schema and the relational schema (in simplified form) that would be derived from the requirements above.

Figure 23 shows a conceptual model satisfying the data requirements listed. Don't worry if you can't yet interpret an ER diagram at this stage. The course unit will quickly get there. The narrower goal here is for you to start learning how to interact with Oracle using SQL*Plus, so, this diagram is just for the sake of completeness. You can go back to it once you've learned the techniques required to draw ER diagrams.

Figure 24 shows the logical model (i.e., a relational schema in our case) that would be derived from the conceptual model in Figure 23. Don't worry if such technical details as the underlining and the annotations of the form `[fk: ...]` and `[derived as ...]`) are not clear to you at this stage. Again, the narrower goal here is for you to start learning how to interact with Oracle using SQL*Plus. Later on, the course unit will teach how Figure 24 follows from Figure 23.

One of the things you will do is to run some SQL*Plus scripts that will create and populate these tables so that you have something to work with in this tutorial.

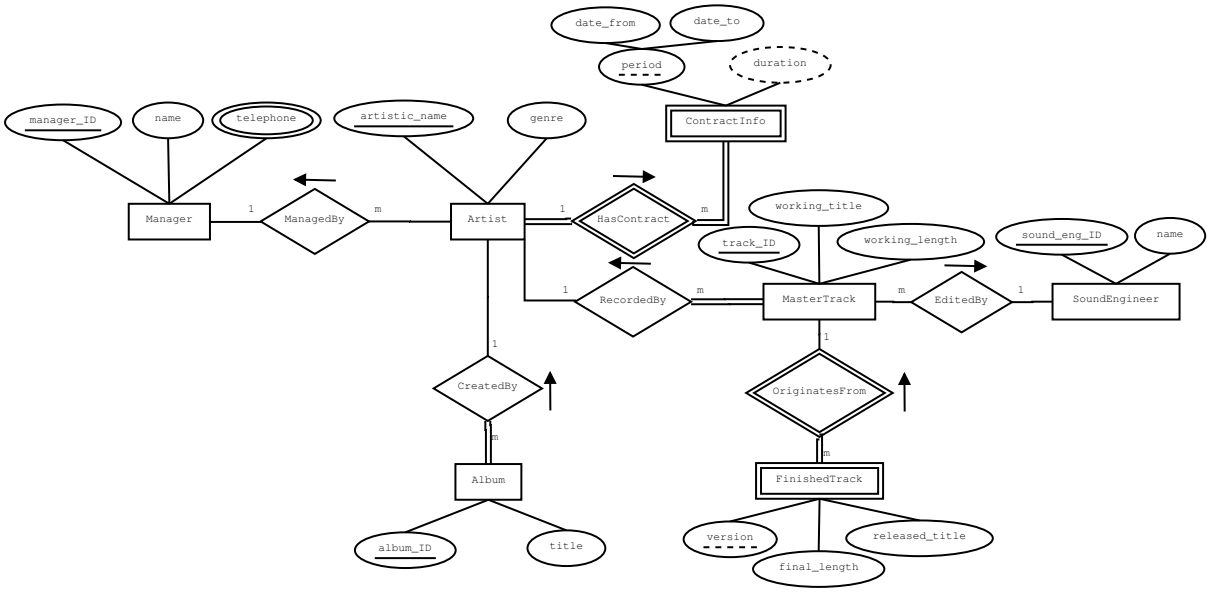

Figure 23: Orinoco [Partial]: Conceptual Model as an ER Diagram

C.4 Interacting with SQL*Plus in the SCS/UoM Labs

Launching SQL*Plus Follow the example in Figure 25 to launch SQL*Plus. When asked, you should use the Oracle user name and Oracke password sent to you by email. If you find a problem with the path

```
Manager       (manager_ID,name) :
               pk[manager_ID]

ManPhone      (manager_ID,telephone) :
               pk[manager_ID,telephone],
               fk[manager_ID -> Manager.manager_ID]

Artist        (artistic_name,genre,managedBy) :
               pk[artistic_name],
               fk[managedBy -> Manager.manager_ID]

Album         (album_ID,title,createdBy) :
               pk[album_ID],
               fk[createdBy -> Artist.artistic_name]

MasterTrack   (trac_ID,working_title,working_length,recordedBy,editedBy) :
               pk[track_ID],
               fk[recordedBy -> Artist.artistic_name,
                  editedBy -> SoundEngineer.sound_eng_id]

SoundEngineer (sound_eng_ID,name) :
               pk[sound_eng_ID]

ContractInfo  (hasContract,date_from,date_to,
               duration [derived as (date_to - date_from)]) :
               pk[hasContract,date_from,date_to],
               fk[hasContract -> Artist.artistic_name

FinishedTrack (originatesFrom,version,released_title,final_length) :
               pk[originatesFrom,version],
               fk[originatesFrom -> MasterTrack.track_ID]
```

Figure 24: Orinoco [Partial]: Logical Model as a Relational Schema

even though you are running a Linux shell in a SCS UG lab machine, let the lecturers or TAs know. (If the full path does not work, try typing just `sqlplus_lab`.)

When you see the SQL*Plus prompt (i.e., `SQL>`), you can simply type in a SQL*Plus command. (Try `HELP TOPIC`, and you will have something to explore.)

We will aim to be precise in differentiating SQL*Plus commands (which are specific to Oracle) and SQL statements (which are, by and large, but not always, standard across many DBMSs). Also, we often refer to a SQL query, and ideally this would only apply to SQL statements that do not cause the database to change, but the use of the more precise word as a synonym for statement is very widespread and we'll probably fail to be consistent in our usage.

You can also paste commands/statements after typing and copying them in a text editor. (This is highly recommended, because retyping is tedious and SQL*Plus is not always as usable as, say, a modern Unix shell. For example, it doesn't support completion, though you can scroll up and down the command history.)

Separate words with spaces or tabs (choose one approach and use it consistently). Commands and statements can appear in upper or lower case (again, choose one approach and use it consistently), but literals (as we will see) are case-sensitive.

There are also special SQL*Plus commands for formatting, setting options and editing and storing SQL commands (see the documentation for details and explore with the HELP command.). It is highly recommended that you explore.

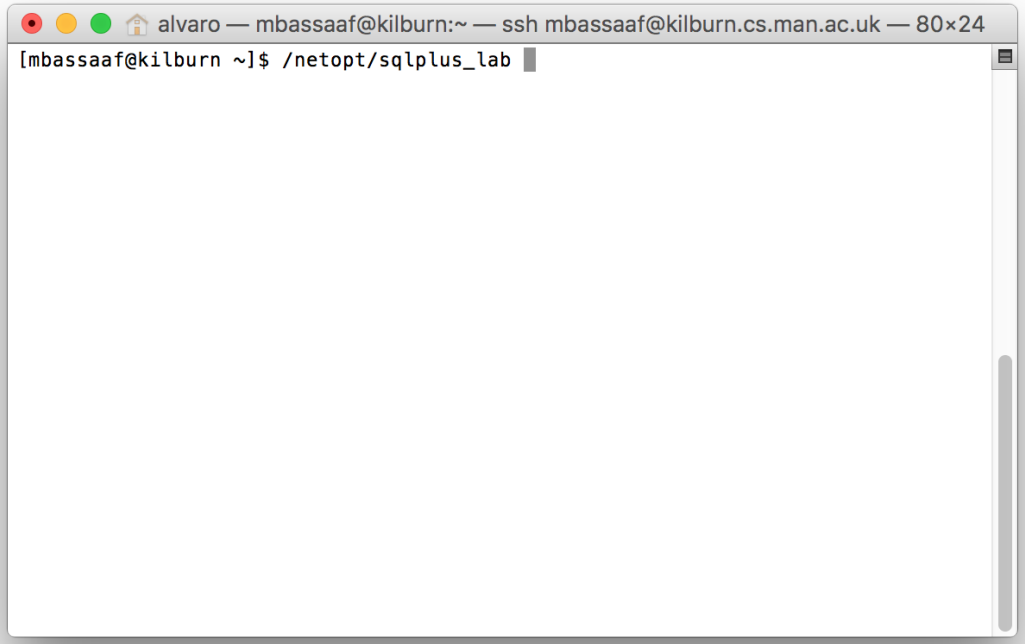

Figure 25: Launching SQL*Plus

Spooling work With the `SPOOL <filename>` command, you can log all commands/statements plus their result into the text file `<filename>`. When you want to stop logging, use `SPOOL OFF`.

Running SQL*Plus scripts You can run a textual sequence of SQL statements stored in a file as a SQL*Plus script (using the `*.sql` suffix). In SQL*Plus, the `START <filename>` command executes the SQL commands in `<filename>`.

In SQL*Plus scripts, a pair of dashes (–) signals that it and all the remaining content of the line where it appears is a comment. You will need to be adept at using comments in the SQL*Plus scripts you will submit for assessment.

We have provided you with three files with SQL*Plus scripts to give you something to play with. Their full paths are:

```
/opt/info/courses/COMP23111/create-Orinoco-tables-partial.sql
/opt/info/courses/COMP23111/populate-Orinoco-tables-partial.sql
/opt/info/courses/COMP23111/drop-Orinoco-tables-partial.sql
```

Use the SQL*Plus `START` command to execute the commands in the files above as you need. The first step is, of course, to create the tables. This produces lots of output. Just browse through it to make sure that no errors occurred. If so, then continue exploring.

Listing a table definition One of the tables you now own is `Manager`. (Later in this tutorial, you will learn how to find all the tables you own.) Use the SQL*Plus command `DESCRIBE` to find out

how `Manager` is represented in the Oracle Data Dictionary (i.e., what is the structure of the `Manager` table). You should see what is shown in Figure 26, i.e., the column names and the types of the values stored (plus an indication as to whether the columns can store null values).

```
SQL> DESCRIBE Manager
 Name                                      Null?    Type
 ----------------------------------------- -------- ----------------------------
 MANAGER_ID                                NOT NULL NUMBER(38)
 NAME                                      NOT NULL VARCHAR2(50)

SQL>
```

Figure 26: Example SQL*Plus DESCRIBE Command

Make a mental note of the fact that SQL*Plus is printing lots of white space between value columns, the reason being that, in the data dictionary, the column where the Name attribute is stored is defined to be quite wide. You will learn to control what is printed, and you will need to exercise judgement as to readability and good layout in your lab exercise reports if you are to merit the full marks for good report presentation. So, be attentive and make time to explore and learn on your own.

Ending a SQL statement You generally will use a semicolon (`;`) to end a SQL statement. Hitting the enter/return key causes the statement to be executed. A statement can span many lines: so, it is the semicolon that denotes its end and it is needed, therefore.

Halting the execution of a SQL statement If execution hangs (which should be very very very rare), use *Ctrl-C* to try and halt an execution.

Selecting all the rows and all the columns in a relation To select all columns and all rows in a table, instantiate the following SQL template:

```
1 SELECT *
2 FROM   <table_name>;
```

The $\star$ character stands for 'all columns'. Try it by replacing `<table_name>` with `Manager`.

Selecting some of columns from some of the rows To select some of the columns only, you list their names in the `SELECT` clause.

You can also select some of the rows only. If you also add a `WHERE` clause that specifies a predicate (i.e., a Boolean-valued expression) then a tuple must satisfy that predicate in order to be part of the output.

The full template is:

```
1 SELECT <col_name_1>, ..., <col_name_n>
2 FROM   <table_name>
3 WHERE  <predicate>;
```

Try finding out the columns that comprise the `Artist` table. Then retrieve the entire content of the table. Finally, retrieve just the names of the artists that satisfy the following predicate `Genre='Dance'`.

The single quotes to enter a string literal are important. Note also that while SQL and SQL*Plus are not, in general, case-sensitive, when you're passing literals in a statement (such as 'Dance') case does matter.

When using quotation marks for literals, be attentive to situations in which your text editor is too clever and generates an illegal character instead. Also, be careful when you copy and paste from PDF files; sometimes the character that gets pasted is not what it looks like.

When you've tried the above, this is the result you should see what is shown in Figure 27.

```
SQL> DESCRIBE Artist
 Name                                      Null?    Type
 ----------------------------------------- -------- ----------------------------
 ARTISTIC_NAME                             NOT NULL VARCHAR2(40)
 GENRE                                     NOT NULL VARCHAR2(20)
 MANAGEDBY                                 NOT NULL NUMBER(38)

SQL> SELECT * FROM Artist;

ARTISTIC_NAME                             GENRE                MANAGEDBY
----------------------------------------- -------------------- ----------
Goldfrat                                  Indie                         1
Simon Palm                                RB                            1
Flut                                      Soul                          2
John Cliff                                CW                            3
Jay Blancard                              Soul                          3
Palmer John                               Indie                         4
Zero7                                     Techno                        4
JZ                                        Techno                        4
Scandal                                   Dance                         5
JK Rowl                                   Dance                         5
PJ Blox                                   Dance                         5

11 rows selected.

SQL> SELECT Artistic_name FROM Artist WHERE Genre='Dance';

ARTISTIC_NAME
----------------------------------------
Scandal
JK Rowl
PJ Blox

SQL>
```

Figure 27: Example SELECT Query into SQL*Plus

After displaying the results, SQL*Plus displays the command prompt again.

Ordering the results You can use the additional template clause (after the `WHERE` clause, or the `FROM` clause if no `WHERE` clause is given)

```
1 ORDER BY <col_name_1>, ..., <col_name_m>
```

to sort the returned tuples in ascending, cascading order. Try sorting the results of your previous SQL statement by artistic name. Use the same statement as above but now add the appropriate `ORDER BY` clause.

Removing duplicates Sometimes the results contain duplicates, but you can eliminate them, if you wish, by adding the keyword `DISTINCT` immediately after the `SELECT` keyword.

Creating and modifying SQL statements Typically you will enter an SQL statement across several lines, execute it and then you will often want to alter it (as we all make mistakes, and more so when we are learning) and try again.

It is highly recommended that you create and modify your statements using a text editor (avoid word processors, as they might tamper with, e.g., quote characters).

Type your statements into the text editor, then copy and-paste them into the SQL*Plus prompt. The line numbers may flash up and make the query look a bit of a mess, but they do not affect its execution. In this way, you can also maintain a history of different queries you have tried and edit/retry complex queries until you get the syntax right.

The most common problems are likely to be errors in the syntax that are difficult to identify from the error messages. You are strongly advised to break your queries into separate lines, and build a complex SQL statement up slowly, bit by bit. This will also help you gain a feel for compositionality, i.e., how a query, for example, can be made of subqueries (i.e., nested queries).

You can also make use of the arrow keys to scroll up and down the command/statement history, but although this is often quicker, it is not as good for documenting your work as using the editor, so, use both methods and get the best of both worlds.

Dealing with errors SQL*Plus has syntax rules for commands and so does SQL for statements in the language. If you break a syntax rule, you will get an error and the command/statement will not execute.

Most error messages will, in the usual way, give you a clue as to what you may have done wrong. If you get an error message that you cannot understand, learn how to use the online Oracle documentation (see the link at the start of this specification). If this is not helpful, ask the lecturers or TAs for help.

Try, for example, misspelling `Manager` as `Minager` on purpose while asking SQL*Plus to `DESCRIBE` it. You'll get an error. Then, in a web browser, go to webpage (see the link above) where all the Oracle database error messages are described and, using the error code as a search term in the appropriate page, find out more about the error by typing in the error code you got. Give or take some inessential differences, you should find in the documentation an explanation like the one shown in Figure 28.

ORA-04043: object *string* does not exist

Cause: An object name was specified that was not recognized by the system. There are several possible causes:
- An invalid name for a table, view, sequence, procedure, function, package, or package body was entered. Since the system could not recognize the invalid name, it responded with the message that the named object does not exist.

- An attempt was made to rename an index or a cluster, or some other object that cannot be renamed.

Action: Check the spelling of the named object and rerun the code. (Valid names of tables, views, functions, etc. can be listed by querying the data dictionary.)

Figure 28: Example Description of an Error

Saving your queries, and the results of your queries, for inclusion in reports For solutions to the tasks in some of the lab exercises, you should incrementally create `.sql` (i.e., SQL*Plus script) files in a text editor (for example, *ned*, or *vi*, or *vim*, or *emacs*, or some other), run the script(s) and, when you're satisfied with it, run it complete, making sure that you spool the result(s) into a separate `.lst` file(s) for inclusion in your report.

Getting the results to look better formatted When the results are returned in a format that makes them difficult for you to even read them, you must use the various commands that SQL*Plus offers to adjust display characteristics. To do so, explore them with `HELP INDEX`.

Use `HELP` to learn about the various things you can `SET`. Also look at the `COLUMN` command. Use `HELP` to learn how you can specify how columns get displayed, and so on. Finally, learn about the `CLEAR` command too. If needed, go beyond the help facility in SQL*Plus and search the Oracle online documentation.

Sensible use of the above can mean the difference between your reports being well-presented or not. If they aren't, you are likely to lose marks.

C.5 The Oracle Data Dictionary

Like most DBMSs, Oracle has a data dictionary containing all the management information on the objects (i.e., tables, views, indexes, etc) in the database. You are not given authorisation to alter this dictionary directly, but you can look at it: this information is held in the form of tables too, so the usual SQL statements also work for the data dictionary tables.

The Oracle online documentation has copious amounts of information on that, as you would expect, but you shouldn't need to delve into it for the purposes of this course unit.

Listing the tables you own The data dictionary table (well, it is actually a view) `user_tables` contains information on all the tables the user (i.e., you) own.

You can try asking SQL*Plus to `DESCRIBE` it, but you'll find that a lot of very technical information about the data dictionary comes flooding out. For the moment, the most interesting column is `table_name`. Since `user_tables` is just a table, you can access the information stored in it using SQL statements. For example, try this SQL statement:

```
1 SELECT table_name
2 FROM   user_tables;
```

Notice that the data strings in the dictionary are in upper case, so any `WHERE` clause that uses literals will have to be aware of this. This is so irrespective of what case was used in the strings that appeared in the SQL statement that created, altered or populated the database.

Use `DESCRIBE` to have a look at `user_tables` and see just how much metadata (i.e., data about data) an Oracle table has.

Integrity constraints in the data dictionary The `DESCRIBE` command accesses information in the Oracle data dictionary. Notice that this does not tell us anything about keys or any integrity constraints defined on the table. We have to actually query the data dictionary to find out such information.

To find out the integrity constraints on, say, the `Manager` and the `ManPhone` tables, we need to look at two dictionary views: `user_constraints` for constraints on the tables, and `user_cons_columns` for constraints on the columns in the tables.

Starting with the tables Here is a query against the Oracle Data Dictionary:

```
1 SELECT constraint_name, constraint_type,
2        table_name, r_constraint_name,
3        delete_rule, status
```

```
4  FROM   user_constraints
5  WHERE  table_name = 'MANAGER' or table_name = 'MANPHONE';
```

You may find that it prints something difficult to read. This is the time to learn about formatting the output printed by SQL*Plus.

Find out in the online documentation how to use the `SET` command to adjust the line width and also the `COLUMN` to adjust how many characters of a column you want to print. A couple of hints for you: one is that adjusting the line width is rarely the complete answer, and another is that the greatest gain in legibility comes in adjusting the format of a column with the appropriate form of the `COLUMN` statement (but, note, very rarely you will need to tinker with numeric values, it's typically names, which tend to be conservatively too long, that cause the most problems). Once you've done your exploring and adjusting what needs to be adjusted, try again. You should see something like (not identical to) what is shown in Figure 29.

```
CONSTRAINT_NAME      C TABLE_NAME          R_CONSTRAINT_NAME    DELETE_RU STATUS
-------------------- - -------------------- -------------------- --------- --------
SYS_C0027948         C MANAGER                                              ENABLED
MP_ID_FK             R MANPHONE            M_ID_PK              NO ACTION ENABLED
MP_ID_PK             P MANPHONE                                             ENABLED
M_ID_PK              P MANAGER                                              ENABLED
```

Figure 29: Example Query Against the Oracle Data Dictionary

We need to go through this result. First of all, notice that the literal values in the dictionary table are all in upper case. This means that when they are queried for, they must be passed as upper case literals.

The constraint type[20] (i.e., the values in the the column named `C`) can be:

`P` : Primary key, i.e., an identity integrity constraint

`R` : Foreign key, i.e., a referential integrity constraint

`C` : Constraint, i.e., a domain integrity constraint in the form of an SQL expression such as `NOT NULL` or `CHECK IN ('cis','cs')`

Constraints named `SYS_xxxxxxxx` are created by Oracle for unnamed constraints (and will, most likely, be different in your case from what you see in Figure 29). Rows 2 and 3 state that the `Manager` and `ManPhone` have a primary key defined and that it is enabled. The constraints have been named by the owner (well, the SQL*Plus script we gave you did that) with specific constraint names. Row 4 states that `ManPhone` has a foreign key pointing to the primary key of `Manager`.

A word of caution here: although, as in this example, it's good practice to give the constraints an informative name, this is not enforced by the DBMS. So, rather than have a name like `M_ID_PK`, the designer could have given it a less helpful name like `C-112`. You would still know that it's a primary key constraint because of the corresponding value (`PK`) in the `C` column. The lesson here is: don't rely on the names alone, use the information in other columns instead, as they are controlled and enforced by the DBMS.

The admissible values in the `DELETE_RULE` column for referential integrity constraints include `NO ACTION` or `CASCADE` (once more, we will learn about these in due course). Finally, the `STATUS` of the constraint can be ENABLED or `not`.

[20] Again, don't worry that you don't know precisely what the meaning of this is. In the course unit, this will soo become clearer.

Now, we can find, e.g., all type C constraints using:

```
1 SELECT constraint_name, table_name, search_condition
2 FROM   user_constraints
3 WHERE  constraint_type = 'C' AND table_name NOT LIKE '%BIN%';
```

(Disregard, for now, the condition `table_name NOT LIKE '%BIN%'`. It's being used here to filter out the many constraints that Oracle generates of its own accord. If you are curious and want to have a look, just rerun the query without the conjunct that filters it and see what you would get otherwise. In this course unit, you can ignore the system generated information: the data dictionary is a repository of both user-level and system-level information, so querying it can be, as here, a bit messy.)

The tuples in the result all have, of course, a C-type constraint. The C stands for CHECK. In this case, to check that particular attributes are NOT NULL.

The `user_constraints` table/view just tells us on which tables there are constraints, but not the columns that the constraints act upon. To see this we have to look at the `user_cons_columns` table/view.

Use the following query now (but, once more, beware copying from PDF documents, sometimes quotes come out wrong and give errors when you paste them into SQL*Plus):

```
1 SELECT constraint_name, table_name, column_name, position
2 FROM   user_cons_columns
3 WHERE  table_name = 'MANAGER' or table_name = 'MANPHONE';
```

Again, this could print something quite hard to read. If it does, then the likely culprit is the column COLUMN_NAME is too wide. It is time to tinker with the way it is displayed. Recall that we suggested that you looked into what the COLUMN (abbreviated COL) command in SQL*Plus does. You should have learned to how use it, so set the column COLUMN_NAME to a narrower width, as follows:

```
1 COL COLUMN_NAME FORMAT A20
```

and try again.

We can now see that the constraint MP_ID_PK refers to the column Telephone in the table ManPhone. The name suggests (but only suggests) that this is a primary key constraint. It also tells us that MP_ID_FK in the ManPhone table refers to the column with name Manager_Id, suggesting (and only suggesting) that this is a foreign key constraint referring to the Manager table's primary key (identified by MP_ID_PK). We can infer much of this information from the constraint names if we always make an effort to give constraints sensible names. (Note that POSITION refers to the position of a column in a composite key. Again, you will learn about these very soon.)

Reloading the tables if they have become corrupted If you've managed to delete or otherwise damage the data in the tables, drop, create and populate again.

Structuring your SQL*Plus scripts Now that you have learned a few things about SQL*Plus, remember that you'll need to submit scripts for assessment. So, practice using the following structure to your scripting:

header where you include identification data

opening where your prepare the SQL*Plus execution environment

body where you include the active, substantive part of your work

close where you do what is needed to clean up after execution

footer where you finish off by matching the header data

Here is a template example that you may want to follow:

```
-- [header]
--
-- COMP23111 Fundamentals of Databases
-- Exercise <NN>
-- by <yourName>, ID <yourStudentId>, login name <yourLoginName>

-- [opening]

SET ECHO ON           -- causes the SQL statements themselves to be spooled
SPOOL <spoolfilename> -- sends everything to <spoolfilename>

-- here you can set the SQL*Plus parameters, such as column width,
-- that will allow the script to produce readable answers in the spool
-- file

-- [body]

-- here you include the active, substantive part of your work

-- [close]

SPOOL OFF

-- [footer]
--
-- End of Exercise <NN> by <yourName>
```

C.6 Practice Tasks

1. *Choose a couple of tables in the Orinoco database and find out how each is represented in the Oracle Data Dictionary.*

2. *Find out the primary key and foreign keys of every table in the Orinoco database (but, again, base yourself on something other than user-provided constraint names).*

3. *Choose a couple of tables in the Orinoco database and retrieve all the columns of every row in the table.*

4. *Choose a couple of tables in the Orinoco database and, using an appropriate criteria of your choice, retrieve some of the columns only from some of the rows only.*

5. *Generate a* `.sql` *script (with an appropriate name) that when executed produces the same output as the previous item above while spooling the result into a correspondingly named* `.lst` *file.*

This is the end of this SQL*Plus tutorial. Do explore further on your own, at your own time.

D On Plagiarism and Working Together

D.1 The Bottom Line

You **must**

1. always work individually
2. only consult and discuss things with your friends once you have gained a thorough understanding of what constitutes academic malpractice
3. never copy solutions from anyone else, or to pass solutions to anyone else, in any form (conversation, paper, electronic media, etc.), unless otherwise authorised explicitly.

D.2 The Official Word

If you have not done so yet, make absolutely sure that you understand and abide by the guidance on Plagiarism & Cheating →Read online.

D.3 Using External Sources

A specific issue to require your specific attention is this: sometimes it is appropriate to learn from code that you found in the Web, but, when you do so, you **must**

4. cite the URLs you've drawn upon as a comment in your submitted solution and
5. explain how you've used it and
6. whether you've used found code itself and, if so, in what way.

Example of Cite-and-Comment In this respect, here is one example of the kind of comment we'd expect from you in your submission:

```
1 -- http://www.tutorialspoint.com/plsql/plsql_procedures.htm
2 --
3 -- This tutorial illustrates how to write PL/SQL procedures.
4 -- No code was used, but some examples were adapted
5 -- in my answer to Question 2.2.
```

```
1 -- http://stackoverflow.com/questions/2621382/alternative-to-intersect-in-mysql
2 --
3 -- I used this link to understand how inner joins can express intersection.
4 -- In my submitted script, the answer to Question 3.1 is adapted from the above.
```

Our mentioning the example URLs above does not imply that we endorse or recommend what they say: they're really just examples.

D.4 On Working Together

You should remind yourself of, and be wise to follow, John Latham's thoughts on the issue of students working together. Here is a reminder (quoted, but very lightly adapted) for your convenience:

> *Many of you will be more than tempted to work together with your friends on the solutions to the problems. We have no objection to this in principle, but be aware of the fine line between working together and copying. We will not tolerate copying. If you do not actually do, on your own, with full understanding, every part you are supposed to do of each exercise that you get marked, then you have copied. Remember the requirement that you fully understand your work, and our reserved right to viva you on it without notice!*
>
> *When working together in an informal group of friends, pay special attention to the relative abilities of people in that group. The real point of laboratory work is not to get the answer, but to learn by doing it yourself, even if you actually get the wrong answer! If you find yourself always telling your friends the answers, then you are not much of a friend to them! You are holding them back and patronising them, but more to the point, you are undermining their learning. Equally, if you find yourself with a friend who keeps telling you the answers, don't be grateful! There is a good chance that he, or she, is simply trying to impress you. Don't be impressed!*
>
> *Everybody who works in an informal group should actually do the work themselves, individually. Such working together should be restricted to discussing ideas and getting the work off the ground. Anybody who cannot actually do the work, should get help from one of the teaching assistants or supervisors. These people are experienced at helping you in such a way that you can do the work yourself, rather than just giving you the answer.*
>
> *In determining the line between help and plagiarism, a good rule of thumb to stick to is this: if you get help from another student, or give help to another student, make sure that the passage of information between you is at a much higher level of abstraction than the final answer, no matter what the medium is (soft copy, typed, written, spoken, etc.). So, for example, for exercises which involve you writing some program code (and which is not supposed to be team work!), never show your code, or your detailed pseudocode, to another student: that amount of "help" is cheating. Thus, for example, if you and a few friends develop a solution to a laboratory exercise together, you are, by definition, cheating.*
>
> *On the other hand, you can help each other as much as you like about things that you have not been asked to create, such as explaining the exercise question to each other, or explaining material that has been covered in lectures.*

Read and heed the above. To reiterate once more: we do not tolerate copying.

References

[Atz+99] Paolo Atzeni et al. *Database Systems: Concepts, Languages and Architectures*. (→Download.) McGraw Hill, 1999.

[BS05] Elisa Bertino and Ravi Sandhu. "Database Security – Concepts, Approaches, and Challenges". In: *IEEE Transactions on Dependable and Secure Computing* 2.1 (2005). (→Download.), pp. 2–19.

[CB15] Thomas Connolly and Carolyn Begg. *Database Systems: A Practical Approach to Design, Implementation, and Management*. Global Edition. (→Read online, and for available printed copies, →See here.) Pearson, 2015.

[DU04] Suzanne W. Dietrich and Susan D. Urban. *An Advanced Course in Database Systems: Beyond Relational Databases*. Pearson, 2004.

[EN13] Ramez Elmasri and Shamkant Navathe. *Fundamentals of Database Systems*. New International Edition. (Available at →Read online, and for available printed copies, →See here.) Pearson, 2013.

[Eur12] European Commission. *Safeguarding Privacy in a Connected World: A European Data Protection Framework for the 21st Century*. Brussels. (→Download.) 2012.

[GUW14] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. *Database Systems: The Complete Book*. New International Edition, 2nd Edition. (No online copies available. For available printed copies, →See here.) Pearson, 2014.

[Haa+14] Lex de Haan et al. *Beginning Oracle SQL*. 3rd Edition. (→Download. No printed copies available.) Apress, 2014.

[Haa05] Lex de Haan. *Mastering Oracle SQL and SQL*Plus*. (→Download. No printed copies available.) Apress, 2005.

[HGV08] J. A. Hoffer, J. F. George, and J. S Valacich. *Modern Systems Analysis and Design*. 5th. (No online copies available. For available printed copies, →See here.) Pearson, 2008.

[HK07] Lex de Haan and Toon Koppelaars. *Applied Mathematics for Database Professionals*. (→Download. No printed copies available.) Apress, 2007.

[KBL05] Michael Kifer, Arthur J. Bernstein, and Philip M. Lewis. *Database Systems: An Application-Oriented Approach (Introductory Version)*. 2nd Edition. (No online copies available. For available printed copies, →See here.) Pearson, 2005.

[Mor+13] Karen Morton et al. *Pro Oracle SQL*. 2nd Edition. (→Download. No printed copies available.) Apress, 2013.

[RG03] Raghu Ramakrishnan and Johannes Gehrke. *Database Management Systems*. 3rd Edition. (No online copies available. For available printed copies, →See here.) McGraw-Hill, 2003.

[SKS06] Abraham Silberschatz, Henry F. Korth, and S. Sudarshan. *Database System Concepts*. 6th Edition. (No online copies available. For available printed copies, →See here.) McGraw-Hill, 2006.