Database Design and Implementation

A practical introduction using Oracle SQL

Howard Gould

bookboon
The eBook company

Howard Gould

Database Design and Implementation

A practical introduction using Oracle SQL

Database Design and Implementation: A practical introduction using Oracle SQL
1st edition

ISBN 978-87-403-1046-7
Peer reviewed by Dr Mark Dixon, senior lecturer, School of Computing,
Creative Technologies & Engineering, Leeds Beckett University

Contents

Acknowledgements

I should like to express my gratitude to colleagues at Leeds Beckett University (formerly known as Leeds Metropolitan University) for reviewing the manuscript and making helpful suggestions. Particular thanks go to Dr Mark Dixon for developing the QSEE CASE tool. The idea for this book evolved from teaching an introductory level databases module to undergraduate computing students for many years. The main material used by the module was a workbook developed and delivered by a number of staff at Leeds Beckett University, and this book is based on some of the ideas and content of the workbook; I would like to thank colleagues past and present for their contributions to the original student workbook, and apologise to those who I have been unable to formally acknowledge here.

Trademarks

Foreword

This book has been written to provide a practical introduction to relational database design and database development for students studying computing-related courses and anyone else who needs to work with relational databases, either as users, designers or developers. Similarly, people who are commissioning a database may benefit from an understanding of this content.

This book is based on an approach that has been used successfully over a number of years to teach many undergraduate computing students, and is presented in a concise form that will make it easy for you to grasp the essential principles and techniques and apply these in any relational database environment.

An established data modelling methodology is explained which includes Entity Relationship models. These are presented using the long-established crow's foot notation; however, as the Unified Modelling Language (UML) has become an industry standard, the Class Diagram notation is also introduced to show how it can also be used for ER modelling, though there are differences (Hay & Lynott, 2008).

The modelling diagrams used in this book have been drawn using the QSEE SuperLite v1.1.2 CASE tool which is available to download from http://www.leedsbeckett.ac.uk/qsee/.

The Oracle database management system and Oracle Application Express (APEX) development environment are used to introduce the industry standard Structured Query Language (SQL). An APEX user account can be obtained from apex.oracle.com for web access; alternatively, a copy of Oracle Database 11g Express Edition and Oracle APEX can be downloaded from Oracle.com for installation on your own system.

Further supporting materials can be found at the author's website howard-gould.co.uk.

1 Introduction to database development

On completion of this chapter you should be able to:

- be aware of the database development lifecycle
- understand the purpose of data modelling
- identify the key terms used in data modelling.

Databases are at the centre of most information systems in everyday use, therefore it is important that they are designed and built using appropriate methods to ensure that they meet users' requirements whilst being robust and maintainable. A database system is usually regarded as the database which contains related tables of data maintained by a database management system (DBMS), along with applications that provide controlled access to the database.

In order to build an effective database system it is important to understand and apply the database development lifecycle, which includes the following phases:-

1. Strategy and planning
2. Requirements analysis
3. Design
4. Development
5. Deployment/implementation
6. Operations and maintenance.

1. Strategy and planning – typically the cycle starts with the strategy and planning phase to identify the need and scope of a new system.

2. Requirements analysis phase – a more detailed requirements analysis will be carried out which will include identifying what the users require of the system; this will involve conceptual analysis.

3. Design phase – this will involve producing a conceptual, logical and physical design. To undertake these processes it is important to be able to understand and apply the data modelling techniques which are covered in this book. When a suitable logical design has been obtained the development phase can begin.

4. Development phase – this involves creating the database structure using an appropriate Database Management System (DBMS) and usually includes the development of applications that provide a user interface consisting of forms and reports which will allow controlled access to the data held in the database. This book will show how the Oracle relational database management system and the Oracle Application Express (APEX) application developer tool can be used for this purpose.

5. Deployment/implementation – when the system has been developed it will be tested, it will then be deployed ready for use.

6. Operations and maintenance – following the system release for use it will be maintained until it reaches the end of its useful life, at this stage the development lifecycle may restart.

1.1 Conceptual data modelling

Why do you need to model?

In many environments modelling is used to ensure that a product will satisfy the user's requirements before it is produced. For example, an architect may use a scale model of a building so the client can see what it will look like before it is built. This allows for any changes to be made to the design following feedback and before any expensive building work takes place. Similarly, a modelling approach is needed when designing a database system so that interested parties can check that the design will satisfy the requirements.

How do you model a database system?

In order to design an effective database system you need to be able to understand an organisation's information needs and, in particular, identify the data needed to satisfy these needs. Entity Relationship modelling (Chen P. 1976) is an important top-down analysis technique which is used to show the structure of the data used by a system. Initially, a **conceptual model** is produced which is independent of any hardware or DBMS system; this is achieved by using an Entity Relationship Diagram (ERD) or alternatively a UML Class Diagram (CD). This modelling technique will be used to determine how this business data is structured and show the relationships between the different data entities. The model forms the basis for the design of the database system that will be built.

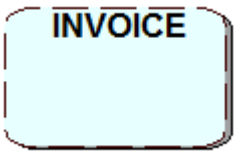
It is crucial to model an information system accurately before building it, to avoid costly mistakes.

1.2 The Entity Relationship Diagram (ERD)

The Entity Relationship Diagram (ERD) shows **"entities"** and the **"relationships"** that link them. The entities represent the data items needed by the system and the relationships show how the entities are related to one another. An **"entity"** is formally called an **"entity type"** and can be defined as:

> "A group of objects with the same properties which are identified by the enterprise as having an independent existence." (Connolly & Begg 2015, p. 406)

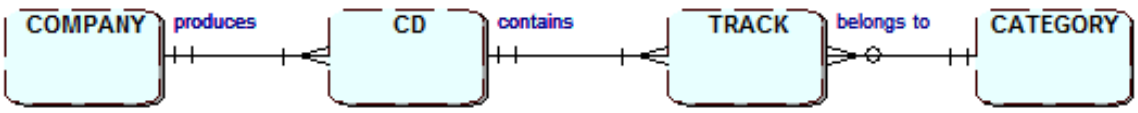
There are a number of notations used for drawing ERDs; this book will show you how to use the commonly used **Crow's foot** notation (Barker 1990). In addition, as the **Unified Modelling Language (UML)** (www.uml.org) is becoming more widely established, Appendix A shows how the UML Class Diagram can also be used for data modelling. Using the Crow's foot notation, each entity type is modelled on the ERD as a round-cornered box with the entity name inside it e.g.



Entity type

> You should always use **UPPER** case letters for entity names, and the name of the entity type should be written in the **singular**, e.g. **INVOICE** not **INVOICES.**
>
> Remember, the entity symbol is used to represent the entity **type**, not the number of occurrences; this information will be added to the ERD later.

An example ERD for the Music System from Appendix B is shown below. This shows four entities – represented by round edge boxes – which are needed; production COMPANY, their music CDs which consist of TRACKs (i.e. songs). Each track is classified by a music CATEGORY (e.g. Pop, Rock). The lines and their symbols linking the entities are the relationships which provide further information about the entities.



Music System ERD

1.3 Entity types

In order to produce an ERD you need to identify all the entity types that are **relevant** to the system being modelled. **Do not confuse an entity type with the occurrence of an entity.**

If modelling using the Unified Modelling Language (UML) then the term "instance" is used to refer to an entity occurrence.

> You should consider the entity type as the definition or template of what data is to be held, and an occurrence as a single set of actual data e.g. entity type: STUDENT, occurrence "Mike Jones".

Often many entities can be identified, although they are not always relevant to the needs of the system being considered, so care needs to be taken to ensure that only those that are needed are added to the ERD. The following are examples of typical entity types:

For a business system: CUSTOMER, ORDER, INVOICE.
For a university system: STUDENT, LECTURER, COURSE.

Entities often fall into one of the following categories:
Physical – CAR, BUILDING
Human – CUSTOMER, EMPLOYEE
Place – FACTORY, SCHOOL
Group – DEPARTMENT, TEAM
Document – INVOICE, PAYSLIP

1.4 Producing the ERD

When you have identified the entity types, these need to be added to the **Entity Relationship Diagram (ERD)**. Although ERDs can be drawn by hand, it is good practice to use a **Computer Aided Software Engineering (CASE) tool** to ensure your models can be amended easily and presented in a professional form to others. There are many CASE tools available to support modelling.

The QSEE tool can be used to draw ERDs and UML Class Diagrams and is available to download from http://www.leedsbeckett.ac.uk/qsee/

CASE tools are more than just drawing tools; they are used to hold information about the data entities and their attributes (Meta data) and can be used to assist in the development stage of the database development lifecycle.

Exercise 1

Identify which of the following are likely to be entity types and which occurrences. If an occurrence, suggest a suitable entity type; if an entity type suggest a suitable occurrence:

PARIS, MODULE, AMIN KHAN, CUSTOMER, STUDENT, CITY

Exercise 1 feedback

ENTITY TYPE	ENTITY OCCURRENCE
CITY	Paris
MODULE	Introduction to Databases
CUSTOMER or STUDENT	Amin Khan

1.5 Entity attributes

When you have identified your entity types you then need to identify their **attributes**. An attribute is defined as follows:

"A property of an entity or a relationship type" (Connolly & Begg 2015, p. 413).

Each entity will usually have a number of attributes. These are the individual items of data that you need to hold for each occurrence of an entity type. In some situations a relationship between a pair of entities may also yield attributes; this situation will be discussed later.

The entity type INVOICE may include the following attributes: Invoice number, Invoice date, Invoice amount and Customer code.

An example entity occurrence of the entity type INVOICE would be as follows:

Invoice number	1102
Invoice date	12-Jan-2015
Invoice amount	1000
Customer code	C101

Exercise 2

Which of the following are likely to be attributes and which are likely to be entity types?
CUSTOMER, PRODUCT, ORDER, ORDER DATE, SIZE, QUANTITY, NAME.

Exercise 2 feedback

ENTITY TYPES: CUSTOMER, PRODUCT, ORDER.
ATTRIBUTES: ORDER DATE, SIZE, QUANTITY, NAME.

1.6 Entity selection and validation

In order to produce the ERD you need to ensure you have identified the entities that are suitable for inclusion. The entities initially selected are usually referred to as **"candidate entities"** as not all may be suitable for inclusion. Entity names are normally **nouns** not verbs. The candidate entities are usually identified by referring to a written system description, a set of requirements, or perhaps the notes from a discussion with a person who has knowledge of the system under consideration. Read through the relevant documents and underline, highlight or draw a box around each noun (an item you can store information about). These nouns will form the candidate entity list.

To ensure that a candidate entity is valid for inclusion on the ERD it should satisfy the following three checks:

1. **It should not be the name of the system being modelled**
   It is a common mistake to include an entity which has the name of the system or organisation that is being modelled. For example, if you were producing a model of "Yorkshire University" it would not be appropriate to include an entity type called YORKSHIRE UNIVERSITY or even UNIVERSITY as there is only one occurrence of this university. The whole model would, in reality, represent the university. However, if you were modelling a system that needed to hold data for more than one university, then you would need to include an entity type called UNIVERSITY.

2. **The object should be of importance to the system being studied**
   There are likely to be many objects in the system being studied but you have to decide whether the object is relevant. This usually means determining if the system users are likely to need to retrieve information about the object. For example, if you were designing a university student information system is a "litter bin" likely to satisfy the check? The answer would be no, but are there any circumstances in which it might? If the purpose of the system was to record all university assets, then you might need to record information about the litter bins. In that case you would need an entity type to represent this information, though the entity type would be called ASSET and bin would be an entity occurrence.

3. **There should be data attributes that can be associated with the entity**
   There must be at least two attributes for an entity type. If you cannot identify any or only one attribute for the entity then you may need to consider whether, in fact, it is actually an attribute of another entity type.

Exercise 3

Identify the candidate entities for the following brief business system description. Then remove any of the candidates that do not satisfy the checks for an entity.

"Customers of the Yorkshire Supplies Co. order high and low value products. Most customers use a computer and so have an email address."

Exercise 3 feedback

"**Customers** of the **Yorkshire Supplies Co.** place **orders** for high and low value **products**. Most customers use a **computer** and so have an **email address**."

An initial search for candidate entities may produce the following list:

CUSTOMER
YORKSHIRE SUPPLIES CO.
ORDER
PRODUCT
COMPUTER
EMAIL ADDRESS

If you now apply the three checks, YORKSHIRE SUPPLIES CO. would be eliminated as it fails the first check. There is only one, i.e. the system being modelled is the "Yorkshire Supplies Co."

COMPUTER would be eliminated as it would not satisfy the second check, as it is unlikely that there would be a need to store details about a customer's computer.

EMAIL ADDRESS would be eliminated too, as it fails the third check, however it is likely to be an attribute of CUSTOMER.

So you would be left with the following entity types which would be added to the ERD:
CUSTOMER, ORDER, PRODUCT.

Once you have identified the entities needed for the system these can be added to the ERD. You should now try and identify the entity attributes. Following the entity checks some attributes may have come to light, either as a result of applying check two where the item may in fact be an attribute rather than an entity, or check three because you only have one attribute. To ensure you have identified all the required attributes you will need to analyse the system documentation which was used to identify the entities and discuss the system with its users to identify and extract all the items of data that are needed for each entity type. The entity and attribute information is usually recorded in a CASE tool repository to make it easier to reference during the system design and development phases. It is unlikely that you will identify all the attributes initially so you should always try and check with the users of the system to ensure that you have not missed any.

All entities require an **entity identifier**. This is an identifying attribute (or attributes) which is used to **uniquely identify** an occurrence of an entity type.

For example:

A motor VEHICLE entity type can use the vehicle registration number as its unique identifier.

Often an attribute presents itself as the natural identifying attribute as in the example above. However, if there is no natural candidate for this you must introduce an artificial one. This is usually achieved by creating a reference code, e.g. Client ID.

Q. Why can you not use 'student name' as the entity identifier for the STUDENT?

A. Because it is possible that you could have two or more students with the same name. In this case, the artificial identifier **student id** could be used as the unique identifier.

Exercise 4

What would be the identifying attribute in each of the following examples of entity types where some attributes have been identified?

CUSTOMER – Customer name, Address, Postcode, Customer code, Phone number.
INVOICE –. Invoice Date, Invoice number, Invoice Amount, Customer code.

Remember, wherever possible you should use an existing attribute for the entity identifier and make sure that it will **always** uniquely identify an occurrence of an entity type.

Exercise 4 feedback

For **CUSTOMER** the obvious choice would be **Customer code**.

Although Customer name might be considered, there may be more than one customer with the same name. Postcode may also be considered but this should be rejected as more than one customer may have the same postcode. Likewise, whilst no two customers should have the same phone number, this would not be a suitable choice as not all customers may have a phone number.

For **INVOICE**, the obvious choice would be **Invoice number** as this would be unique for each invoice. The other attributes would not be suitable as their values could be duplicated amongst all the invoices.

Exercise 5

For a university student information system list two entity types.

1. For each entity type list some of their attributes including an identifying attribute.
2. Produce a sample entity occurrence for each entity type.

Exercise 5 feedback

Entity type:	STUDENT
Attributes:	Student ID, Name, Address, Post Code, Date of Birth
Identifier:	Student ID
Occurrence:	S101, Paul Adams, 4 Long Row Leeds, LS6 3QS, 12-Jan-1960
Entity:	COURSE
Attributes:	Course code, Course name, Start date
Identifier:	Course code
Occurrence:	COMP, BSc. Computing, 01-Sep-2015

1.7 Entity definitions

It is important to ensure that anyone involved with designing the system is clear about the meaning of the entities being modelled. This is achieved by clearly documenting each entity type with a concise unambiguous definition. The list of definitions is referred to as the **"data dictionary"** or **"data repository"** and is usually stored within a CASE tool as this provides a central reference point and allows for easy searching, amendment and reporting. Other information relating to the entities and their attributes may also be added to the data dictionary such as the number of likely entity occurrences and the data types and sizes of the attributes. This information may be needed in the design phase.

Here are examples of full definitions for some entity types.

Entity type	Entity definition	Entity Attributes
CUSTOMER	A person or organisation who purchases products or services from the business.	Customer name, Address, Postcode, Customer code, Phone number.
INVOICE	A request to a customer for payment for products or services supplied by the business.	Invoice Date, Invoice number, Invoice Amount, Customer code.

1.8 Validating the model

The model should be checked with the client or system users to ensure that all relevant entities have been identified, along with the required attributes. This process may need to be repeated a number of times until everyone is satisfied that all requirements have been met.

Remember it is easier and cheaper to change a model than it is to change a developed system.

Exercise 6

For the following business descriptions identify the candidate entities and eliminate any which are not entities, giving a reason for this.

For each remaining entity provide an entity definition and some suitable attributes including the identifying attribute.

1. A large business consists of a number of divisions. Each division has a number of departments. Each employee works for a department.

2. A car hire company's customers make bookings to hire its cars. When a booking is made it is for a specific model of car and includes the collection and return dates. At the time of the booking the company may assign a particular car, however at the time of collection a different car may be provided. The company needs to keep records of which car each customer actually hired.

Activity 6 feedback

The identifying attributes are shown in **bold**. The # symbol is used to represent a number e.g. Employee number.

1. BUSINESS is eliminated as there is only one occurrence; you are modelling the business.

   DIVISION — Attributes: **Division Name**, location, …
   Description: A major functional area of the business.

   DEPARTMENT — Attributes: **Department Name**, office location, …
   Description: A functional section within a division of the business.

   EMPLOYEE — Attributes: **Employee**#, name, office, grade, ….
   Description: A member of staff who works for one of the departments within the business.

2. CAR HIRE COMPANY is eliminated as you are only modelling the one.

   CUSTOMER — Attributes: **Customer**#, name, address, Tel no. …
   Description: A person registered to hire cars from the company.

   BOOKING — Attributes: **Booking Ref**, booking date, Customer#, car, collection date, return date, …
   Description: A booking of a car for use by a customer of the company.

2 Entity Relationships

On completion of this chapter you should be able to:

- understand what is meant by an entity relationship
- identify and model entity relationships.

2.1 Introduction

In the previous chapter you started to use analysis techniques to try and identify a system's entities and their attributes in order to model the system. As entities do not exist in isolation you need to be able to identify the relationships that link them together and so complete the entity relationship diagram (ERD).

2.2 Relationships

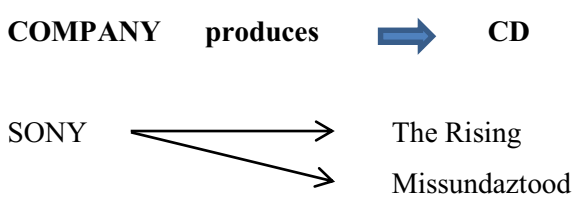
In order to see what is meant by a relationship, consider the following example which uses the music system entity types COMPANY, CD, TRACK and CATEGORY.

There are a number of relationships between these entity types as follows:

- A COMPANY produces CDs
- A CD contains music TRACKs
- A TRACK belongs to a music CATEGORY

It is important to understand how one occurrence of an entity relates to an occurrence of another entity in order to define the relationship between them accurately.

To help you understand the nature of a relationship you may initially find it helpful to see the entity occurrences in a graphical format. If you consider the relationship "COMPANY produces a CD" the diagram below shows how one occurrence of COMPANY relates to two occurrences of CD when looked at from the viewpoint of the COMPANY, which is at the one end of the relationship. From this direction the relationship can be read as "A COMPANY produces CDs."



You now need to consider the relationship from the other direction, the CD to COMPANY. This relationship direction could be described as "A CD is produced by a COMPANY." As shown below:

COMPANY ⟸ **is produced by** **CD**

SONY ⟸ The Rising
Missundaztood

By looking at the relationship between two entities in both directions you can define the relationship with meaningful labels. Do not confuse relationships between entity types and entity occurrences.

Exercise 1

Which of the following are relationships between entity types and which are relationships between occurrences?

1. CUSTOMER receives INVOICE
2. YORKSHIRE SUPPLIES sells TV
3. STUDENT studies COURSE
4. JOHN JACKSON studies COMPUTING

Exercise 1 feedback

1 and 3 are relationships between entity types, 2 and 4 are relationships between entity occurrences.

Exercise 2

The following table shows an extract from the records of a food store. It shows CUSTOMERs and the PRODUCTs that they purchased.

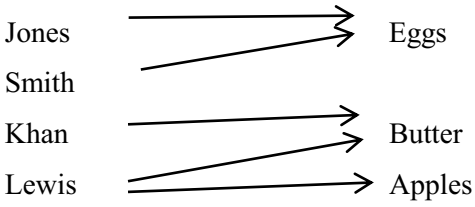
CUSTOMER	PRODUCT
Jones	Eggs
Smith	Eggs
Lewis	Apples
Lewis	Butter
Khan	Butter

Draw a diagram showing:

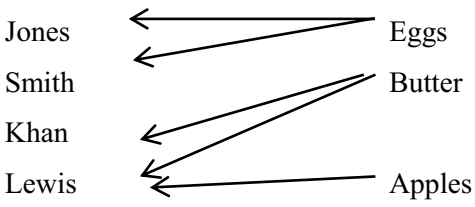
a) how each occurrence of CUSTOMER is related to occurrence(s) of PRODUCT
b) how each occurrence of PRODUCT is related to occurrence(s) of CUSTOMER

Exercise 2 feedback

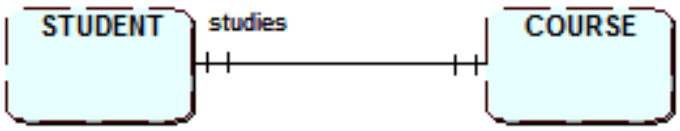
a) CUSTOMER is related to occurrence(s) of PRODUCT



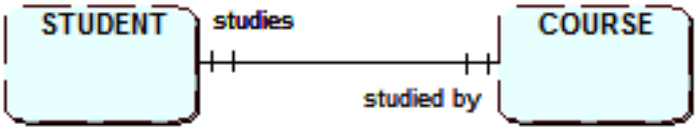
b) PRODUCT is related to occurrence(s) of CUSTOMER



If there is a relationship between an occurrence of one entity type and an occurrence of another entity type, then it is shown on the entity relationship diagram as a line linking the two entity symbols. The relationship between the two entities should be labelled by using a suitable verb. For example the relationship "STUDENT Studies a COURSE" would be represented as follows:



As it is important to consider a relationship from **both directions** you should also label the relationship from COURSE to STUDENT as follows:



The relationship labels should be positioned as above, near to the relevant entities to aid readability.

Always use UPPER case text to label entities and lower case text to label relationships.

Exercise 3 Label the following relationship:- **CUSTOMER receives INVOICE** from the direction of **INVOICE** to **CUSTOMER**.

Exercise 3 feedback **INVOICE is sent to CUSTOMER**

2.3 Relationship cardinality

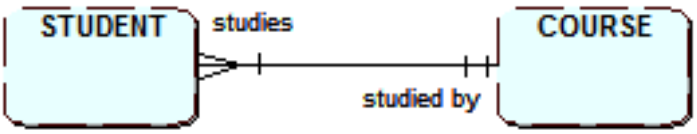
Once you have established a relationship between two entity types it is important to consider how many occurrences of one entity could be related to the other entity. This is referred to as **"cardinality"**.

There are three types of relationship cardinality:

One to One abbreviated as **1:1**
One to many abbreviated as **1:M**
Many to Many abbreviated as **M:M** or **M:N**

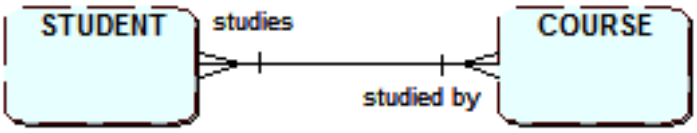
These will now be explained.

Using the earlier example of the relationship between STUDENT and COURSE, consider the relationship from the STUDENT's viewpoint. You can say that a student can study a course and if you then consider the relationship from the COURSE viewpoint, you can say that a COURSE can be studied by many STUDENTs. This would be a **"one to many"** (1:M) relationship and would be drawn on the ERD as follows:



A one to many (1:M) relationship

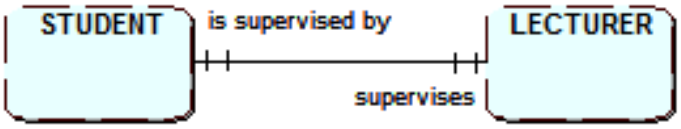
The **"crow's foot"** symbol is used to represent many and is placed at the "many" end of the relationship. The relationship would be read formally as "a student studies one and only one course and a course is studied by one or many students". If you now reconsider the relationship between STUDENT and COURSE but want to be able to show that a student may study more than one course, you now need to alter the relationship to show as a **"many to many"** (M:M or M:N). A M:N relationship is sometimes written as M:M though M:N is preferred so as to indicate that the number of occurrences at one end of the relationship can be different from number at the other end of the relationship. This is drawn on the ERD as follows:



A many to many (M:N) relationship

There are now crow's foot symbols at both ends of the relationship, because they are both "many" ends. This relationship would now be read as "a student studies one or many courses and a course is studied by one or many students".

The final cardinality type that needs to be examined is for the **"one to one"** relationship. If a STUDENT is assigned a LECTURER as a supervisor and the LECTURER only supervises one student, you can show this as follows on the ERD:



A one to one (1:1) relationship

In this case the relationship would be read as "a student is supervised by one and only one lecturer and a lecturer supervises one and only one student.

Exercise 4

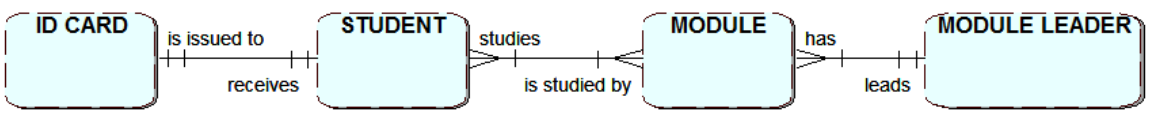
Draw a single Entity Relationship Diagram showing the four entities and three relationships for each of the following:

a) An ID CARD is issued to a STUDENT
   A STUDENT receives an ID CARD

b) A STUDENT studies one or more MODULEs
   A MODULE is studied by one or more STUDENTs

c) A MODULE LEADER leads many MODULEs
   A MODULE has one MODULE LEADER

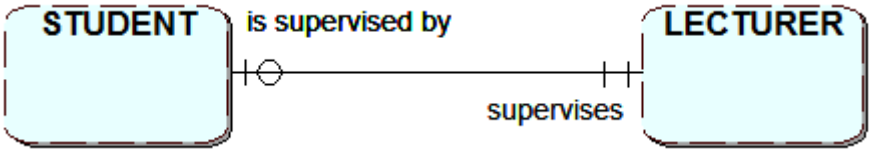
Exercise 4 feedback



2.4 Relationship optionality

When describing an entity relationship you need to record the fact on the ERD that in some cases an occurrence of an entity type may not always be present, in which case the relationship is said to be **optional**. Using the previous cardinality example, the model states that a lecturer supervises a student. However, what if some lecturers do not act as supervisors to students? In this situation an occurrence of LECTURER will not always be related to an occurrence of STUDENT so it will be an optional relationship. However, if you consider the relationship from the STUDENT perspective it is still present as all students must have a supervising LECTURER.

To denote that a relationship can be optional a small circle is included on the relationship line at the end that is optional. The following shows the optional 1:1 relationship between STUDENT and LECTURER:



A one to one (1:1) relationship with optionality.

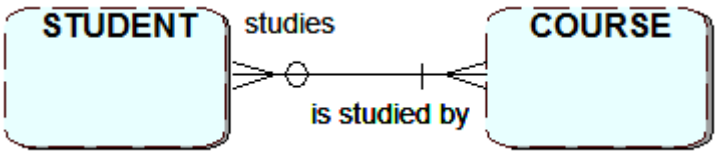
The circle might be viewed as the letter O for optional but it is best considered as a zero.

As you will recall, you always consider the relationship from the perspective of a single occurrence of each participating entity. This diagram would be read as "a student is supervised by one and only one lecturer and a lecturer supervises zero or one student". Notice that the optionality is placed at the opposite end of the relationship to the entity on which you are concentrating. From the STUDENT end of the relationship, this diagram shows that a STUDENT is always supervised by a LECTURER.

Referring to the previous M:N relationship between STUDENT and COURSE which shows a student studies one or many courses and a course is studied by one or many students, could there be optionality in this relationship? Consider this from both ends of the relationship; firstly do all students study a course? Secondly are all courses studied by students?

The answer to the first question should be obvious, as if the student was not studying a course they would not normally be classed as a student. To answer the second question you need to consider that a university may advertise a new course which will not have enrolled any students yet. Consequently, at any time, there could be at least one occurrence of COURSE that has zero STUDENTs attached to it.

This optionality can now be shown on the ERD as follows:



A many to many (M:N) relationship with optionality

The alternative to an optional relationship is a mandatory relationship. If the relationship is mandatory i.e. where there will always be at least one occurrence of the entity type, this is usually shown by a short vertical line | on the relationship. If you recall that the O can be said to represent a zero, then the | can be taken to represent the number one. The relationship as drawn above therefore shows the minimum and the maximum cardinality for each direction of the relationship.

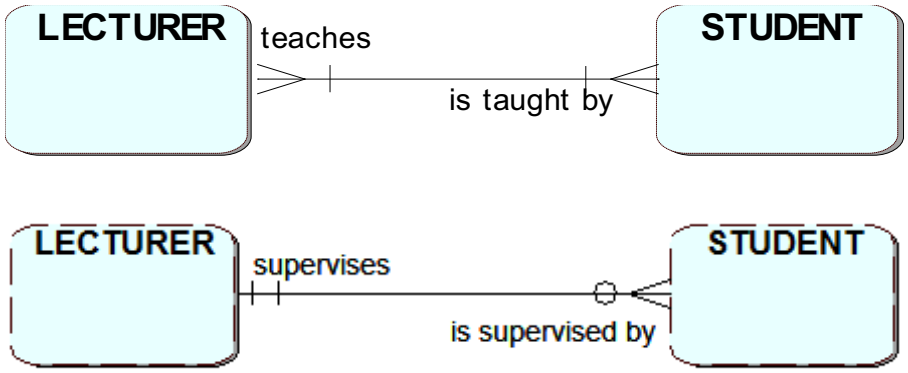
It is helpful to think of the inner relationship symbol as the minimum and the outer symbol as the maximum number of occurrences. In the example above the inner symbols used ore o and | and the crow's feet are the outer symbols.

The relationship is read as follows:

One STUDENT studies one or more COURSEs
One COURSE is studied by zero (possibly one) or many STUDENTs.

The reading of the relationship starts with "one" whatever the cardinality at that end is. Always read the relationship in **BOTH** directions to ensure you have correctly identified the cardinality and optionality for both ends of the relationship.

It is important to label relationships clearly. If you do not, readers may misinterpret the relationship and choose the wrong cardinality or optionality. For example, the following pair of entities is shown to have two different relationships, each requiring different cardinality and optionality values.



Different relationships

Exercise 5

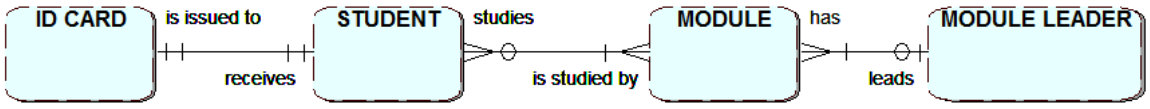
Here is a revised version of Exercise 4. The relationships have now been altered to include optionality in some areas. Produce an amended ERD to reflect this.

a) An ID CARD is issued to a STUDENT
A STUDENT receives an ID CARD

b) A STUDENT studies one or more MODULEs
A MODULE is studied by zero, one or more STUDENTs

c) A MODULE LEADER leads many MODULEs
A MODULE may have a MODULE LEADER

Exercise 5 feedback



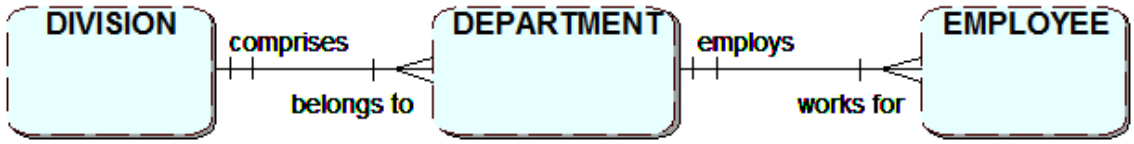
Exercise 6

Below are the exercises that you modelled in the last chapter. Using the entities you identified, draw ERDs showing all the entities and relationships for each scenario. Remember to use meaningful relationship names and determine the cardinality and optionality.

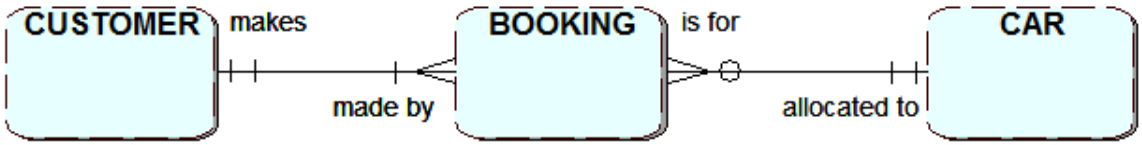
1. A large business consists of a number of divisions. Each division has a number of departments. Each employee works for a department.
2. A car hire company's customers make bookings to hire its cars. When a booking is made it is for a specific model of car and includes the collection and return dates. At the time of the booking the company may assign a particular car, however at the time of collection a different car may be provided. The company needs to keep records of which car each customer actually hired.

Exercise 6 feedback

1.



2. Note it is assumed that there may be some cars that have not received any bookings



3 Complex relationships

On completion of this chapter you should be able to:

- build a more complex entity relationship diagram
- identify relationships that have data associated with them
- model entities in which occurrences are related to each other.

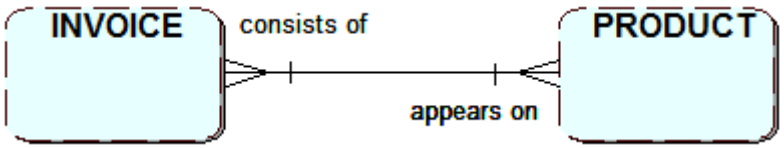
3.1 Introduction

The previous chapters have introduced you to the basic concepts of the entity relationship modelling technique. You are now going to look at some modelling situations in more detail. In particular you are going to consider a technique to enable you to deal with modelling relationships that have data associated with them. You will also be introduced to extended entity-relationship modelling techniques (sub types and exclusive relationships) that can be used in more complex situations.

3.2 Associative relationships

An ERD may contain several many-to-many relationships. If you consider these many-to-many relationships in more detail you are likely to discover that they actually hold attributes of data. To understand this, consider the following example.

In a business situation an invoice showing products purchased could be represented by an INVOICE entity type related to a PRODUCT entity type as a M:N relationship shown as follows:

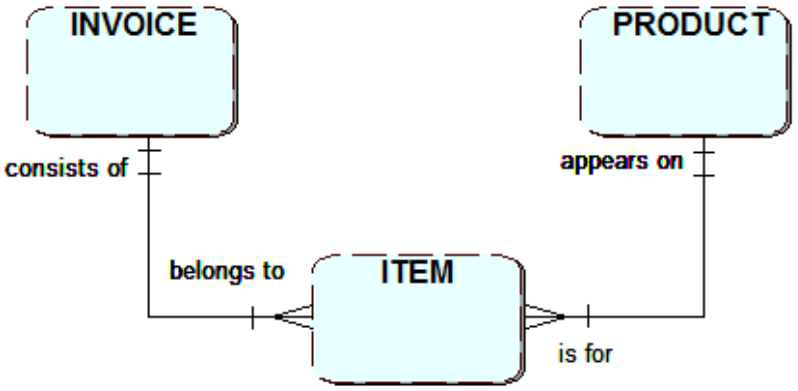


Many to many relationship between INVOICE and PRODUCT

If you consider the data attributes that might be stored for INVOICE, these might include the invoice number (unique identifier), invoice date and customer code. The PRODUCT attributes would include product code, description and unit price. However, this combined set of attributes does not actually represent the complete set of data needed to represent an actual business invoice, as you cannot identify the quantity purchased of a product for a specific invoice.

This is clearly data that needs to be recorded, but it cannot be recorded as an attribute of either of the relationship entities. In fact, it needs to be placed somewhere else, as will be explained shortly. It is desirable to show this hidden information on the ERD. This is accomplished by resolving the many-to-many relationship and capturing the data in a third entity type. This new entity links together a single occurrence of each of the other two entities.

The process used to resolve a many-to-many relationship involves adding a new entity type which is often referred to as an **associative** or **link** entity, and replacing the original relationship with **two** one-to-many relationships linking each of the existing entities to the new link entity, as shown below:



Resolved M:N

The new link entity can now be used to hold the quantity purchased of a product. It will also need other attributes which will be used as the entity identifier and also to link to the appropriate occurrence of the INVOICE and PRODUCT entities; this will be explained later. In this example the new link entity has been called ITEM as it represents an invoice item line. However, it is not always easy to think of a meaningful name for the link entity, although if the original relationship name was meaningful this may provide the answer. Alternatively, if you consider the purpose of the new entity and, in particular, what attributes will be included it should make it easier to choose one. If you cannot identify a suitable name, you could combine the two entity names e.g. INVOICE and PRODUCT.

A M:N relationship is replaced with a new entity and two 1:M relationships. The new link entity uses both original entity identifiers together to form its identifier.

Exercise 1

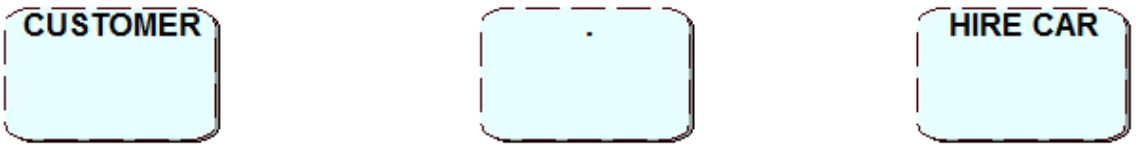
For a car hire business consider one occurrence of CUSTOMER and one occurrence of a HIRE CAR. What data might you need to record about that particular customer's use of the hire car?

Exercise 1 feedback

The main data values to be recorded would be the collection date and return date, though you may choose to store other information as well.

Exercise 2

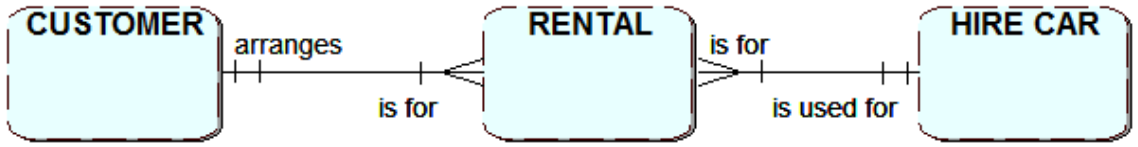
a) Resolve the many-to-many relationship between CUSTOMER and HIRE CAR. Make sure that you show the cardinality of the relationships correctly. First choose a suitable name for the new link entity. Then you can decide on sensible names for the relationships.
b) Suggest some suitable attributes for CUSTOMER and HIRE CAR.
c) Suggest some suitable attributes for the new link entity.



Exercise 2 feedback

Based on Exercise 1 you may choose to call the new entity type RENTAL or HIRE.

a)



If it is likely that a car may not be hired out, the optionality symbol could be placed on the relationship between hire car and rental at the rental end of the relationship.

b) CUSTOMER (Customer no., booking date, customer name,…..)

HIRE CAR (Car reg., model, …)

c) RENTAL (Customer no., collection date, return date, Car reg, …)

3.3 Link entity identifier

As mentioned in Chapter 1, all entities must have a unique identifying attribute, and those looked at so far have all been single attributes. However this is not appropriate for a link entity.

It is unlikely that there will be a suitable natural single attribute for identifying the new entity. The identifier for a CUSTOMER is its Customer no. and the identifier for a HIRE CAR is its Car Registration. However, neither of these identifying attributes can be used alone in the new entity as they may have repeat occurrences, as shown in the example tables below. In this situation, the identifier is usually formed by combining the identifiers from the original pair of entities.

Let us apply this to the car hire system that was modelled in Exercise 2. The unique identifier that will identify a single occurrence of RENTAL will be the combination of Customer no. and Car Registration.

Now, if you examine some occurrences of these entities, you can clearly see that a combined identifier is needed to uniquely locate a specific RENTAL of a particular HIRE CAR by a CUSTOMER.

What issue might arise relating to the identifier if a customer could rent a car more than once?

If the following sample data is used:-

CUSTOMER

Customer no.	**Name**
C101	M Jones
C102	A Khan

HIRE CAR

Car reg.	**Model**
A77 NWW	Peugeot 205Gti
F123 XWX	Subaru Imprezza
RS101	Ford Focus RS

RENTAL

Customer no.	**Car reg.**	**Collection date**	**Return date**
C101	A77 NWW	10-Jan-2015	15-Jan-2015
C102	A77 NWW	3-Mar-2015	3-Mar-2015
C101	F123 XWX	7-Apr-2015	8-apr-2015
C102	RS101	11-Jun-2015	22-Jun-2015

If customer C101, having hired car A77 NWW in January 2015, then hired it again in March, including the data in the table above would cause the combined identifier C101and A77 NWW to be duplicated, so a further attribute, collection date, would be needed as part of the identifier to uniquely identify the occurrences.

RENTAL

Customer no.	Car reg.	Collection date	Return date
C101	A77 NWW	10-Jan-2015	15-Jan-2015
C102	A77 NWW	3-Mar-2015	3-Mar-2015
C101	A77 NWW	7-Mar-2015	9-Mar-2015
C101	F123 XWX	7-Apr-2015	8-apr-2015
C102	RS101	11-Jun-2015	22-Jun-2015

Exercise 3

Model the following situation:

The small independent "Yorkshire Cinema" has two screens, Screen one and Screen two. Only one film each evening is shown on each screen. Cinemagoers can make online bookings for films currently being screened. To make a booking the person selects the screen and the screening date. A booking reference number is emailed to the purchaser. Cinemagoers sometimes enquire about current films and information about the film actors appearing in them, film length etc. is supplied.

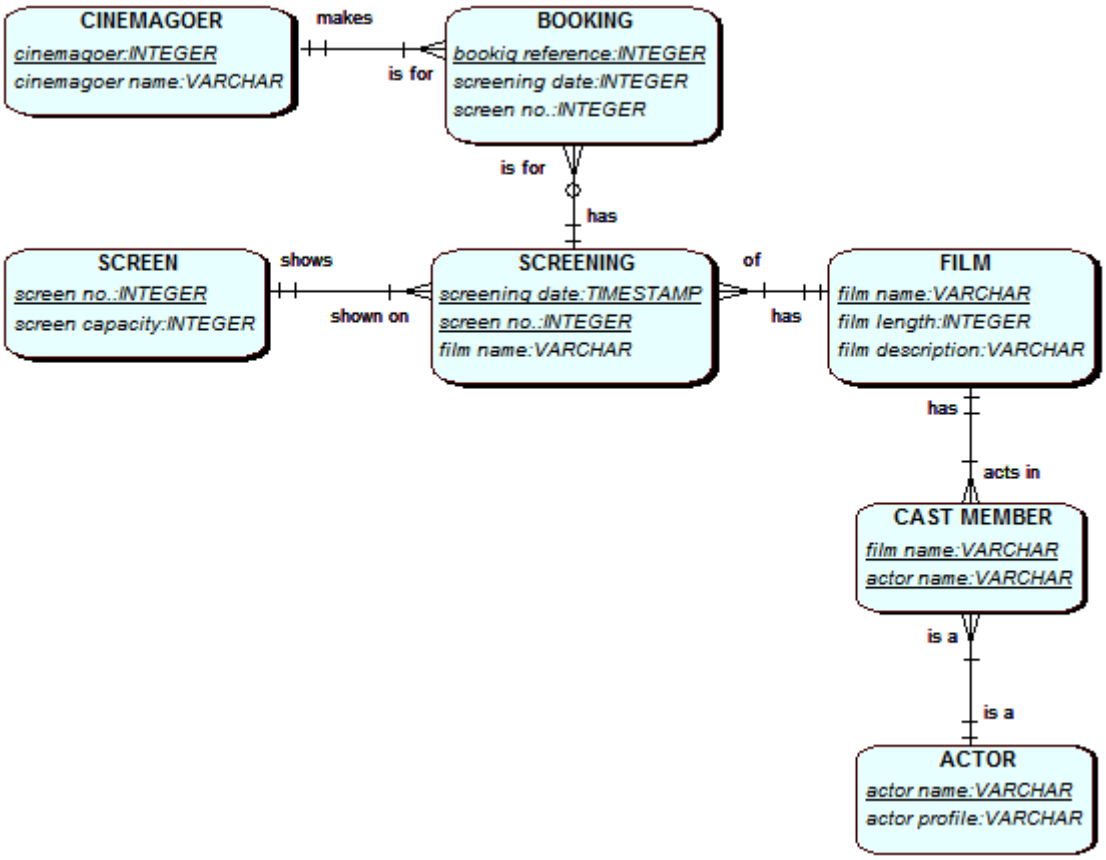
The entity types have already been identified for this system as SCREEN, CINEMAGOER, BOOKING, SCREENING, FILM and ACTOR.

a). Draw the ERD for this case study including these entity types and the relationships linking them. Remember to name the relationship from both entity types involved and also to include the cardinality and any potential optionality. There is one many-to-many relationship; resolve this by including a link entity.

b). List at least TWO possible attributes including the identifier for each entity type.

Exercise 3 feedback

This ERD shows the entity types with two attributes and their data types (data types will be discussed in Chapter 6), the identifiers are shown underlined. This is made possible as the attributes were entered into the case tool.



The many-to-many relationship between FILM and ACTOR has been resolved by including the link entity CAST MEMBER.

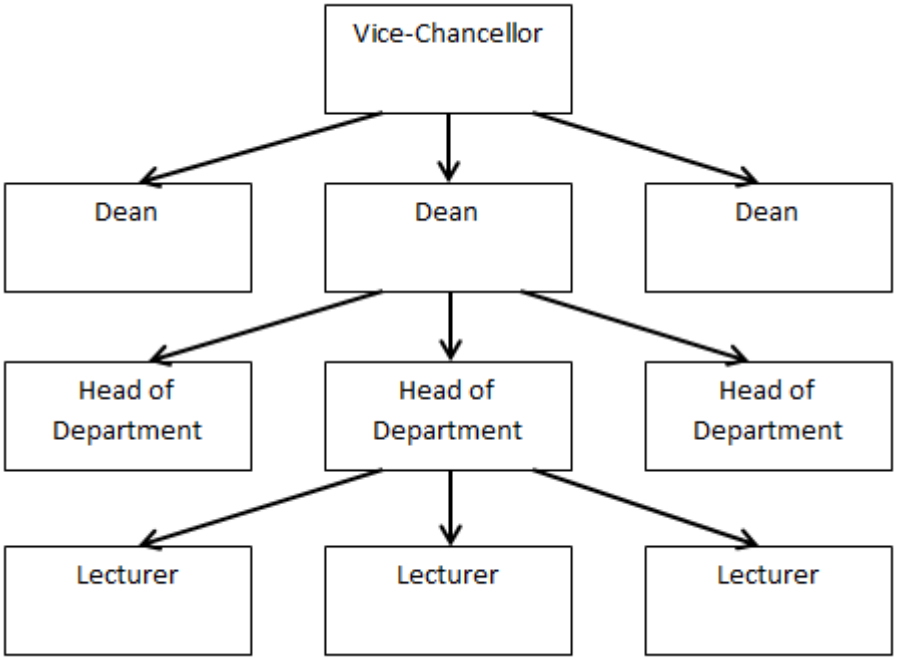
3.4 Recursive relationships

So far you have concentrated on identifying and modelling relationships between pairs of entity types. Most of these relationships will be one-to-many, a few might be many-to-many and some might be one-to-one. You have also discovered how to resolve many-to-many relationships that contain data which is of interest in the situation being modelled.

You may also encounter entities that are related to themselves. To be more specific, occurrences of the entity type are related to other occurrences of the **same** entity type. This is called a **recursive relationship.**

Consider the entity type EMPLOYEE in a university where there are approximately 500 employees, resulting in 500 occurrences of the entity. The Vice-chancellor manages the Deans of Faculty and each Dean manages several Heads of Department. The Heads of Department manage the lecturers.

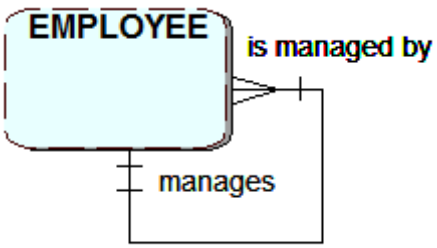
This gives rise to a hierarchical relationship within this single EMPLOYEE entity type. This can be represented graphically using a hierarchy diagram, as follows:



The "manages" relationship between occurrences of EMPLOYEE

This hierarchy diagram clearly shows that an occurrence of the entity EMPLOYEE, say Dean, manages one or more other occurrences of EMPLOYEE. Another occurrence of the entity, a Head of department, also manages one or more other occurrences of the same entity, Lecturer.

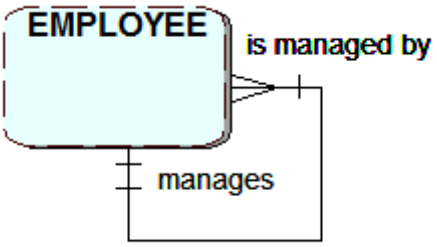
To show this 1:M recursive relationship on an ERD you draw a relationship line starting and finishing at the entity, as follows:



Modelling a recursive relationship

This relationship would be read from the 'one' end as, "an EMPLOYEE manages one or more other EMPLOYEEs". Reading the relationship from the other end, you can say an EMPLOYEE is managed by one and only one EMPLOYEE.

Following a more detailed analysis, this model does not accurately model the 'manages' relationship between occurrences of university employees. The reason for this is that there is an optional participation in the relationship. The Vice-chancellor in effect manages all employees. However, the Vice-chancellor is not managed by another member of staff. (His/her activity is monitored by a Board of Governors, but the Governors are not occurrences of the entity EMPLOYEE). Similarly, there are many individual lecturers who do not manage any other staff. Consequently, this optionality needs to be modelled on the ERD:



Recursive relationship with optionality

The ERD now accurately shows the 'manages' relationship and can be read as an EMPLOYEE manages zero, one or more other EMPLOYEEs, and an EMPLOYEE is managed by zero or one other EMPLOYEEs.

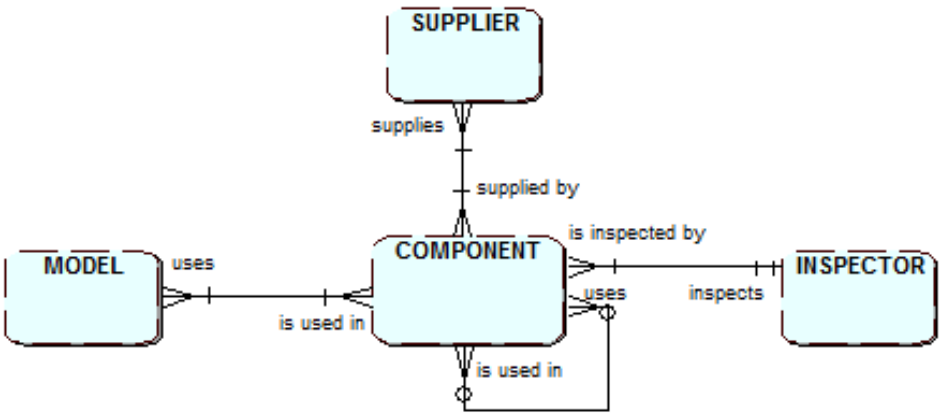
Exercise 4

Model the following situation:

A car manufacturer makes a number of different models of car, each of which comprises many components. Some components are used in many models. A number of suppliers supply the components and each supplier supplies a number of different components. In some cases components are used to build other components, e.g. pistons, crankshaft etc. are used to build an engine which is a component of a car. Each component is checked by the quality inspector before use; the inspector will be responsible for checking many different components.

Exercise 4 feedback

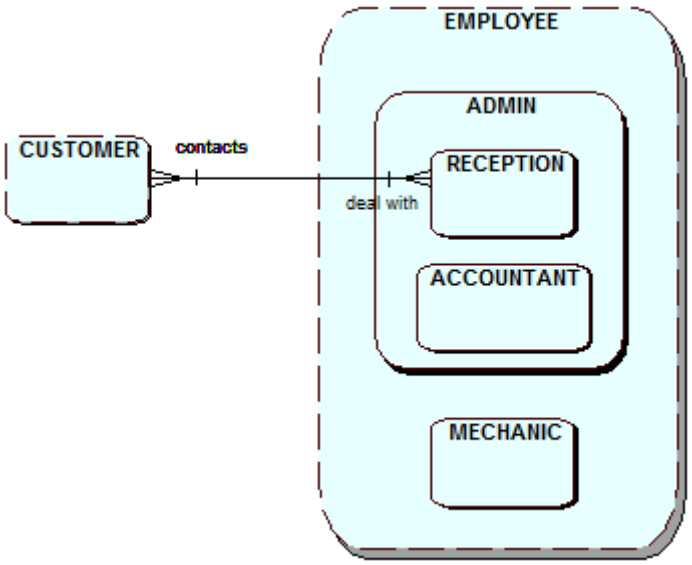
The recursive relationship is read as "a component may be used in zero, one or many components and a component may use zero one or many components".



3.5 Sub types

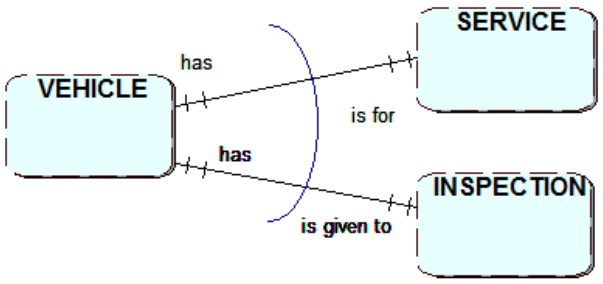
When studying some entity types it becomes apparent that they have **sub types** this is usually the case when the sub types contain different sets of data attributes. The main entity is usually referred to as the **Super type** and contains attributes that are common to all of the entity's sub types. It is possible to have sub types within a sub type, although it is not advisable to have too many sub levels. The sub type identifier is the same as its super type identifier. The use of sub types also allows you to show relationships more accurately. An example of this is given below, where a CUSTOMER only has contacts with an administrative employee. This is shown by the relationship to the sub type RECEPTION, rather than the more general super type EMPLOYEE.

E.g. An employee at a car hire company could be modelled using sub types to show the different categories of employee.



3.6 Exclusive relationships

Sometimes two or more relationships are mutually exclusive, e.g. a VEHICLE may be undergoing a SERVICE or an INSPECTION but not both at the same time. This is shown by an arc symbol pointing towards the mutually exclusive options.



3.7 Summary

Here is a summary of the main terminology relating to entities and relationships that you have now been introduced to:

Entity	A data object of interest to the system
Attribute	A property of an entity
Identifying attribute	An attribute or combination of attributes that uniquely identifies an entity occurrence
Relationship	An association between two entities
Cardinality	The number of occurrences in one entity that can have a relationship with the occurrences in another entity **one to one** — 1:1 **one to many** — 1:M or 1:m **many to one** — M:1 or m:1 **many to many** — M:N or m:n (or M:M)
Optionality	Participation in a relationship

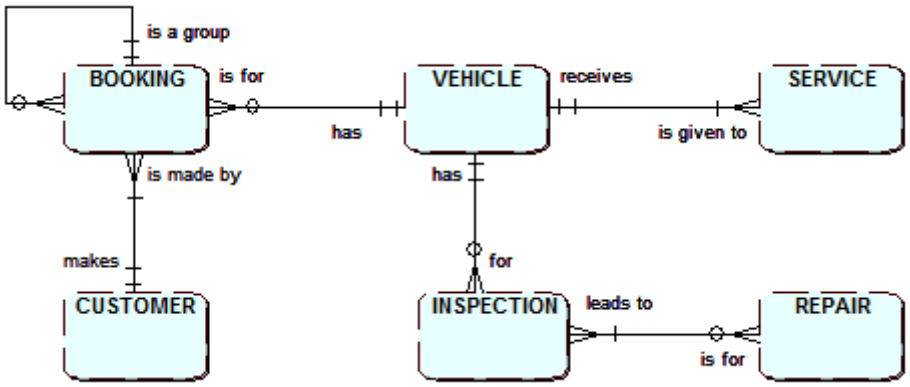
Exercise 5

1. A car hire company takes bookings from customers for hire vehicles. A booking may relate to a group of other bookings i.e. when a customer has made multiple bookings for vehicles to be collected at the same time. Vehicles receive regular services. When vehicles are returned following a hire they may be inspected. Following an inspection the vehicle may need repairs. A repair will result in further inspections.

   a) Produce a list of entity types then draw an entity relationship diagram for the system which includes appropriate relationship names, cardinality and optionality.

Exercise 5 feedback

a)



4 Logical Database Design

On completion of this chapter you should be able to:

- convert an ERD into a logical database design
- identify primary and foreign keys
- provide a logical design for relationships of different cardinalities.

4.1 Introduction

Once you have completed the conceptual design and have produced your Entity Relationship Diagram you can move on to the next stage in the database development lifecycle, namely **Logical Design**. You will produce an initial logical design for a database in this chapter, and in the next chapter you will study the process of **Normalisation**. This is used to check the structure of your database tables with a view to eliminating data redundancy and ensuring that you have produced an efficient set of tables to be implemented in the final design stage **Physical Design**.

In this chapter you will learn how to turn an Entity Relationship Diagram into a logical design comprising a set of **relations** (not to be confused with **relationships**). This design will become a set of tables suitable for implementation using a relational database management system such as Microsoft Access, MySQL or Oracle. A **relation** is a table-like structure made up of columns (attributes) and rows. Each column has a domain name and this defines the nature of the data to be held.

4.2 Relations

The **relational database** derives from the relational model which is based on mathematical concepts introduced by (E.F. Codd, 1970). A **relation** is a logical construct that is similar to a table. (Please note that a relation is **not** the same as a relationship). However, as you will see below, the term relation can be used in a slightly different context.

A relational database just stores data, and nothing more, inside tables. Any processing of the data is done by using a data manipulation language which works on the tables to output information or alter values stored within the tables. The most commonly used data manipulation language used for accessing relational databases is the **Structured Query Language (SQL)**.

Although SQL has an American National Standard (ANSI) there are some differences between the implementations used by the different database management systems.

A relation can be defined as a logical representation of an entity type with its attributes and **keys**.

Here is an example of a relation for the entity type CUSTOMER:

CUSTOMER (**Customer_no.**, name, address, date_of_birth, ……)

Customer no. is the unique identifier, but in the relation it is referred to as the **Primary Key (PK)**.

Just as all entity types have a unique identifier, all relations have a Primary Key.

The alternative definition of a relation is the relational database table itself. However, to avoid confusion in this context you would normally refer to it as a 'table'. You can now visualise the relation CUSTOMER as a table showing occurrences of entity types as follows:

Customer_no.	Name	Address	Date_of_birth	…
C4347888	Amin Khan	103, Short Street, Wakefield WYS6 2EG	13-Jan-1987	
C8365872	Sarah Jones	13, Gain Lane, Sheffield SS7 1AX	09-Dec-1980	

Although all tables consists of rows and columns, a relational database table has to satisfy the following rules.

- Each row of the table (also called a **'tuple'** – sounds like 'couple') is associated with exactly one entity occurrence so no two are identical (although column values other than the primary key could be identical)
- Rows can be in any order
- Each table column contains attribute values.

4.3 Keys

Keys play a vital role in database design and have to be identified and used correctly. The following terminology is used in association with relational database keys:

- a key uniquely identifies an entity occurrence: it is the entity identifier
- a primary key is the key 'chosen' for a given relation / table
- a candidate key is a 'possible' primary key (several candidate keys may exist for a relation)
- a compound key is a key consisting of two or more attributes.

In the following relation:

VEHICLE (vehicle_identification_no., registration_no., vehicle_make, vehicle_model, …)

both vehicle_identification_no. and registration_no are candidate keys.

(Note that from now on, spaces in the names of attributes will be replaced by underscore characters like this: An_attribute_name_of_several_words. This is because when the design is implemented the database management system will not allow column names which include spaces.)

Exercise 1

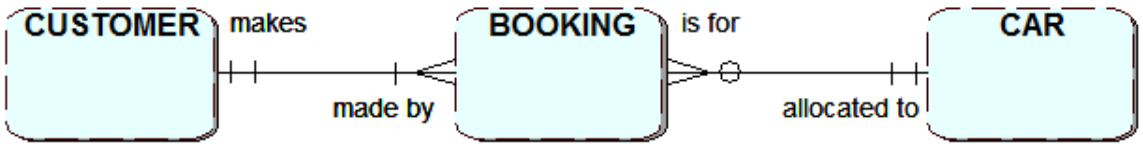
Which candidate key from the VEHICLE relation above would you choose as the primary key and why?

Exercise 1 feedback

A manufacturer-assigned vehicle_identification_no. would be chosen, as the vehicle registration_no. may change over the lifetime of the vehicle. Though depending on the use of the system, vehicle registration no. may be a better choice, i.e. in a car hire business.

Exercise 2

a) Consider the following ERD:



Choose a Primary Key for each of the entities.

b) Which of these applies to the key for BOOKING?

- a compound key
- a candidate key

Exercise 2 feedback

a) Primary key for entity type CUSTOMER would be Customer_no.
Primary key for entity type CAR would be vehicle_identification no. or registration number.
Primary key for entity type BOOKING would be Customer_no. and vehicle_identification no. together.

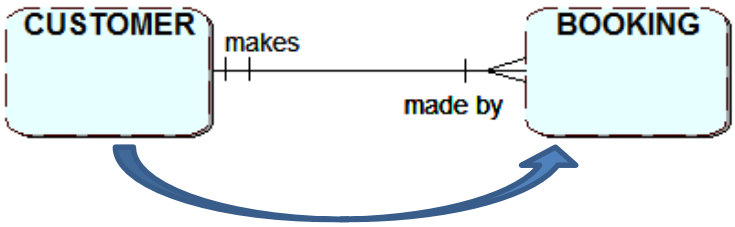
b) A compound key.

4.4 Identifying relations

From your conceptual data model you need eventually to generate a set of relations (tables) that will form the basis of the database. You will need to link these tables in order to be able to reflect the relationships that were modelled on the ERD at the conceptual stage. In order to get to the stage of producing the tables, you first need to produce a complete set of relations in which all of the keys have been identified.

In order to establish the relationships between entities you will make use of keys. In order to do this a new type of key, the **Foreign Key (FK)**, is needed.

For example, earlier you modelled the one-to-many relationship between CUSTOMER and BOOKING on the conceptual level. To achieve a logical design for the tables in your database you will need to produce two relations: CUSTOMER and BOOKING. In order to take into account the link between these two relations you must copy the primary key of CUSTOMER (which is at the "one-end" of the relationship) into the relation BOOKING (at the "many-end" of the relationship). So the Customer_no. becomes a foreign key in the relation BOOKING. The following diagram emphasises this:



This ERD would result in the following two relataions:

CUSTOMER (**Customer no.**, customer name, customer_address, …)

BOOOKING (**Booking ref.**, Booking_date, ***Customer_no, …***)

In order to identify all the relations and their keys you need to work methodically from your ERD, checking each relationship in turn in order to produce a set of **'skeleton relations'**. A **skeleton relation** shows only the name of the relation and **all** of the keys, i.e. the primary key and any foreign keys. The additional attributes can be added at a later stage to form a completed set of relationships.

A skeleton relationship for the booking from above is shown as follows:

BOOKING (**Booking ref.**, ***Customer_no***)

It is important to ensure that keys are clearly identifiable, as such the following notation is used in printed text when showing relations:

Primary key – bold or underline or both e.g. **Student_ID**

Foreign key – *italicise* e.g. ***Course_Code***

In handwritten text you can use a dotted underline for a foreign key.

In handwritten text a key that is both a primary key and a foreign key can be underlined twice, once with dotted and once with normal underline.

Also note that if underlining a compound key consisting of 2 or more attributes, ensure the underline under them is continuous e.g. **order_no, line_no**

Whichever notation you choose, you should ensure that you use it consistently.

A foreign key is an attribute (or combination of attributes) in one relation which exists as a primary key in some other relation.

Remember for a 1:M relationship with no optionality at the one end, the primary key from the one end of the relationship goes in at the many end as the foreign key.

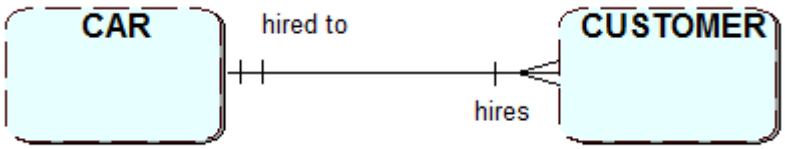
Exercise 3

Can you think of a reason why you must put the foreign key into the BOOKING table and not the other way around?

Exercise 3 feedback

As a customer may make many bookings you would need to make provision to hold all of the Booking references in customer. This is not practical, as you would need to decide the maximum number of bookings to allow for, and if this number altered it would need a change to the implemented table structure. If a large number of Booking Refs. were allowed for this could also result in many of them being left empty in the customer row. Finally, it is much more difficult to search through a repeating list of attributes, so whilst it is technically possible to store repeating attributes this is not normally acceptable so you should avoid doing this.

Exercise 4



a) Which of the following would you do to represent the above relationship between a hire CAR and CUSTOMER on the logical level?

- put Car_reg. as an attribute of CUSTOMER
- put Customer_no. as an attribute of CAR
- both
- neither

b) What would be the skeletal logical relations in this case? Choose a sensible notation for the primary and foreign keys.

Exercise 4 feedback

a) put Car_reg. as an attribute of CUSTOMER

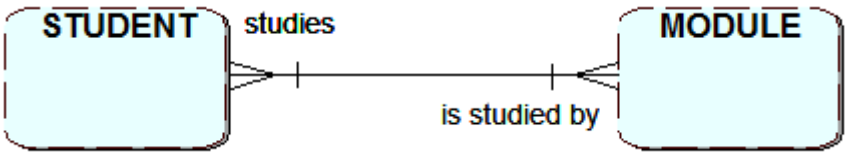
b) CAR (**Car_reg.**) Car_reg. is the primary key in CAR

CUSTOMER (**Customer_no.**, ***Car_reg.***) Car_reg becomes a foreign key in CUSTOMER.

4.5 Resolving many-to-many relationships

In Chapter 3 the M:N relationship was examined and it was shown that it has data associated with it. It was also shown that this could be resolved by inserting a link or associative entity. Although it was convenient to expand the relationship at that time, it could have been postponed until the logical design stage. In the logical design whenever you are dealing with an M:N relationship, whatever the optionality you will need to create a new relation to represent the relationship and produce a skeleton relationship for it.

For example, the following ERD will require three skeletal relations:



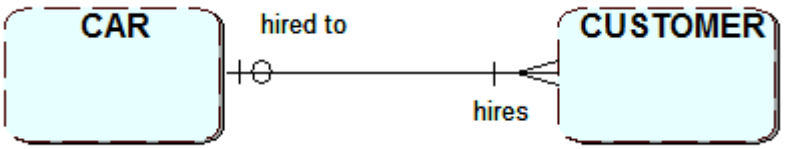
STUDENT (**Student_id**)
MODULE (**Module_Id**)
STUDY (***Student_id, Module_id***)

Note that in the last relation here, which represents the M:N relationship, both foreign keys are needed together to form a compound primary key, which will uniquely identify occurrences that will exist in the resulting relationship table STUDY.

4.6 Resolving one-to-many relationships with optionality

You have already seen how a one-to-many relationship **without optionality** is resolved to produce two relations. You now need to consider how to deal with optionality in a one-to-many relationship. If optionality is shown at the **many** end of the relationship then the previous approach can still be applied. The primary key from the one end of the relationship will be placed in the relation at the many end. If, however, there is **optionality** at the **"one"** end this will not work!

Consider the example used in Exercise 4, only now there is optionality at the one end of the relationship:



This would be interpreted as "a customer may not be allocated a car for hire immediately i.e. when they make a booking". In this situation, if you tried to assign the Car_reg. as the foreign key in CUSTOMER this would result in an empty value as there is no car reg. to enter! The value would be what is referred to as **null**; however it is **not acceptable** to allow a null value for any key. To avoid this situation, three relations are created in the logical design:

CAR (**Car_reg.**)
CUSTOMER (**Customer_no.**)
HIRED_CAR (***Customer_no.***, *Car_reg.*)

Note that the new relation contains two foreign keys ***Customer_no.*** and ***Car_reg.*** The ***Customer_no.*** was chosen as the primary key for the new relation as the customer is only allocated one hire car at a time. With this arrangement an entry is only made in the HIRED_CAR table when a car is actually allocated to a customer.

Exercise 5

What changes to the above relations would you make if you wanted to record the history of a customer who may, over time, hire a number of cars?

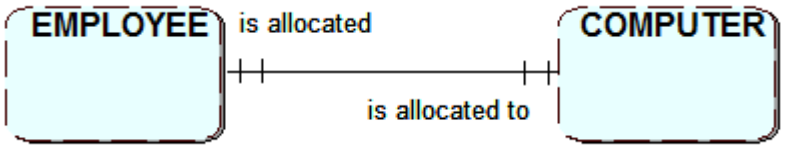
Exercise 5 feedback

You would need to add the **hire date** to the HIRED_CAR table. The ***Customer_no.*** would no longer be a suitable primary key for this table as it could duplicate, so a compound key would be needed using the Customer_no. with the hire date. Note the Car_reg. would not be a suitable alternative as the customer could over time hire the same car again.

4.7 Resolving one-to-one relationships

There are three possible combinations for one-to-one (1:1) relationships and each requires a different process.

Firstly, you may have a 1:1 relationship which is **mandatory at each end**, for example:

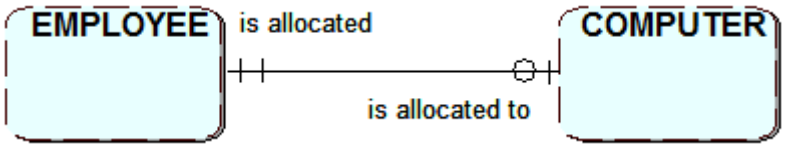


In this type of relationship there will only ever be one set of attributes from one entity type matched with one set of attributes from the other entity type, so both sets can be merged to form one database table. At the logical design stage the two entities are effectively merged into one relation with a single primary key. Either of the following skeleton relations could be used:

EMPLOYEE (**Employee_No,** Computer_ID)
or
COMPUTER (**Computer_ID,** Employee_No)

Which you choose depends on the purpose of the system.
If it is for staff management then the first alternative would be chosen.
If it is equipment maintenance then the second would be chosen.
Note that the remaining attributes would still be the same whichever is chosen.

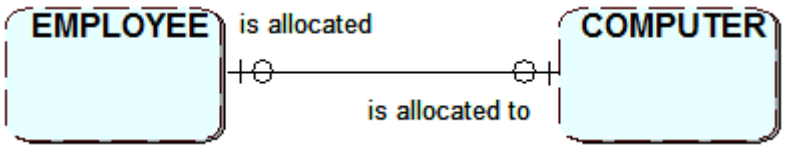
The second situation is a one-to-one relationship with **optionality at one end**, for example:



If an employee may not be allocated a computer then in this situation a relation must be created for each of the entities. The key of the entity at the non-optional end is posted into the relation for the optional entity. This approach avoids ending up with null value **Employee_No** attribute in the COMPUTER table

EMPLOYEE (**Employee_No**)
COMPUTER (**Computer_ID**, ***Employee No***)

Finally, consider the third situation, which is a one-to-one relationship with **optionality at both ends**, for example where an employee may not be allocated a computer or where a computer may not be allocated to an employee:



Three relations are needed, one for each entity and one to express the relationship between the employees and their computers:

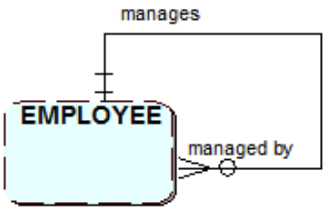
EMPLOYEE (**Employee_No**)
COMPUTER (**Computer_ID**)
COMPUTER_ALLOCATION (***Employee_No***, ***Computer_ID***)

Note that **either** of the original identifiers could have been used as the primary key for the third relation. If a history of which computers have been allocated to an employee is needed, a further attribute would need to be included in the primary key e.g. date_allocated.

4.8 Recursive relationships

When you have an entity type which can have an occurrence that relates to other occurrences of the same entity type this is called a **recursive relationship** and can be shown as follows:

For a 1:M with or without optionality at the "many" end of the relationship, include a **foreign key** attribute to hold the primary key value of the one end record, i.e. the employee's manager's Employee_No. This would be read as "an employee manages zero, one or many employees and an employee is managed by one and only one employee".

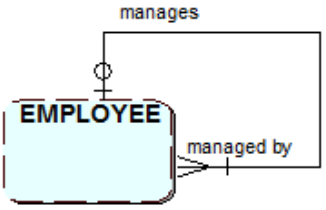


EMPLOYEE (**Employee_No**, emp_name)

becomes

EMPLOYEE (**Employee_No**, emp_name, ***mgr_emp_no***)

For a 1:M with optionality at the one end, a new relation is needed to avoid holding a null foreign key value. In this example not all employees have a manager, so a manager relation is needed which will hold the employee's Employee_No (Primary key) and their manager's emp_no (Foreign key), for each employee who has a manager.

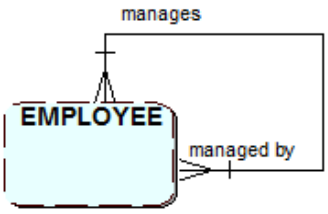


EMPLOYEE (**Employee_No**, emp_name)

and

MANAGER (***Employee_No, mgr_emp_no***)

Where there is a M:N relationship a new relation is used to hold the employee's Employer_No and their manager's emp_no. Both foreign keys are used to form a compound primary key



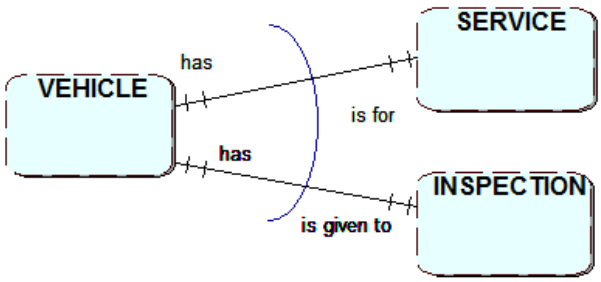
EMPLOYEE (**Employee_No**, emp_name,)

and

MANAGER (***Employee_No, mgr_emp_no***)

4.9 Exclusive relationships

In some situations you may wish to show that a relationship is mutually exclusive. This is shown using an exclusive arc facing towards what would in effect be the optional entities. In this example a vehicle may receive a service or an inspection, but not both.



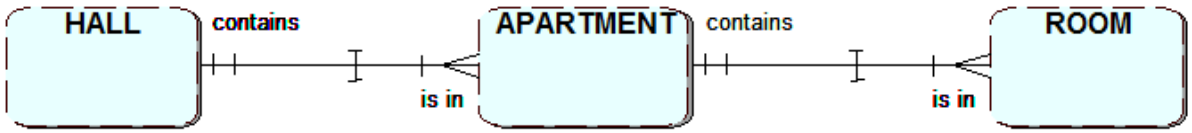
This results in the following relations, which avoids the null foreign key problem.

VEHICLE (**Registration_no**)
SERVICE (**Service_no**, *Registration_no*)
INSPECTION (**Inspection_no**, *Registration_no*)

4.10 Identification Dependency

In some situations the primary key from the non-dependent (one) end of a relationship becomes part of the key at the **dependent (weak)** end of the relationship. An entity is said to be dependent or weak if it cannot exist without the existence of a strong entity i.e. if it inherits part of its identifier from another entity.

The following shows an example for a student hall of residence where a number of halls exist, each have a number of apartments and within each apartment there are a number of student rooms. In this situation the apartment entity is dependent on the hall, as each apartment would be identified by a hall name and the apartment number together. Likewise, the room entity would also be dependent as its identifier would be made up of the primary key from the apartment entity and the room number. A large I symbol is shown on the relationship to indicate an identifying relationship as follows:



Identifying dependency relationships

These are the relations showing the keys:

HALL (**Hall_name**)
APARTMENT (***Hall_name*, Apartment_no)**
ROOM (***Hall_name, Apartment_no*, Room_no**)

An entity is referred to as **strong** if its primary key is not dependent on another entity.
An entity is referred to as **weak** if its primary key is partially or wholly made up of attributes from other entities, as in a link entity created from a M:N relationship, or where an attribute from within an entity is used with a foreign key to form a compound primary key.

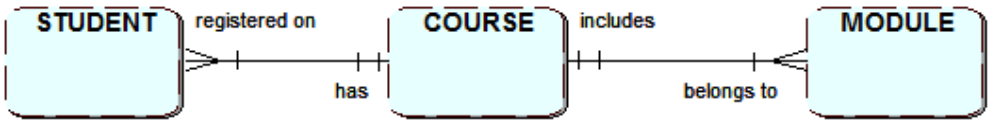
4.11 Modelling problems

Care needs to be taken when modelling systems to ensure that you avoid producing a design which, if implemented, will not allow the system to extract all of the required information needed to answer user queries. With this is mind you should check your model for the following problems:

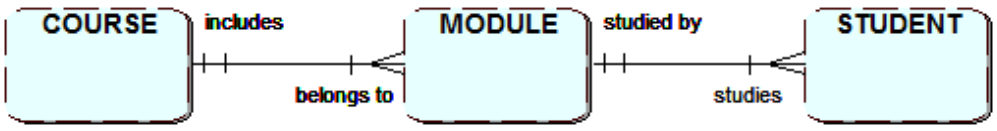
Fan trap

A fan trap is caused when it is not possible to link from one entity to another entity via a linking entity because the two 1:M relationships point away (fan out) from the linking entity.

For example, suppose you want to know if a student is studying the module Databases. The model below will not allow you to answer this query. Although the module is related to a course by the foreign key Course_ID, there is no suitable link from course to student as there is no foreign key Student_ID in course for student.



To resolve this problem a new relationship could be added, linking module directly with student, though a neater solution would be to rearrange the model as follows:-

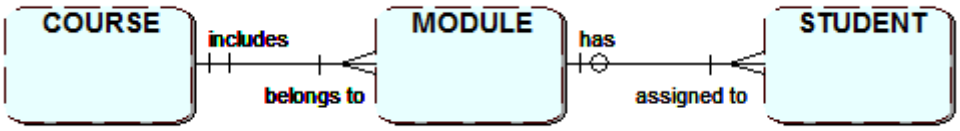


The student relation will now hold the Module_ ID as a foreign key. This would also be used to form a compound primary key with the Student_ID, as a student would normally be expected to study more than one module. It is still possible to identify which course a student is studying because the Course_ID is stored as a foreign key within the module relation and so provides a link to the course title in the course relation. As well, the student relation is linked to the module relation using the foreign key Module_ID.

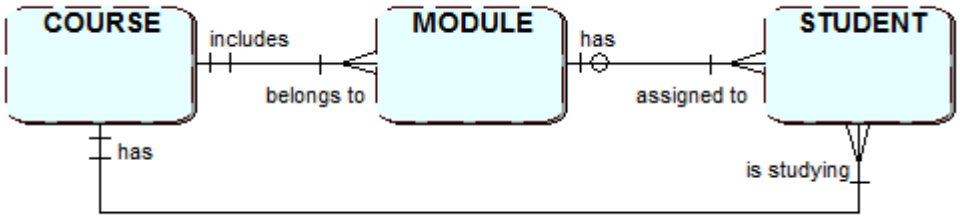
Chasm trap

A chasm trap is created when relationships between entity types indicate a route linking them, but due to optionality it is not possible to make the required connection for all occurrences.

For example, suppose you want to identify which course a student is taking. The following model will not work if the student has not been assigned to a module. Although there is a link between course and module due to the Course_ID foreign key in the module relation, there would be no link between student and module if the student was not taking a module – there would be no foreign key Module_ID in student to provide a link to the module relation.



This problem can be resolved by adding a 1:M relationship linking course to student directly.



4.12 Summary

An ERD can be converted into a logical design suitable for a relational database by defining a set of relations, some of which have been derived directly from the ERD entities, others coming from the relationships. The relations include all the keys which will be required to link the tables in the database when it is created.

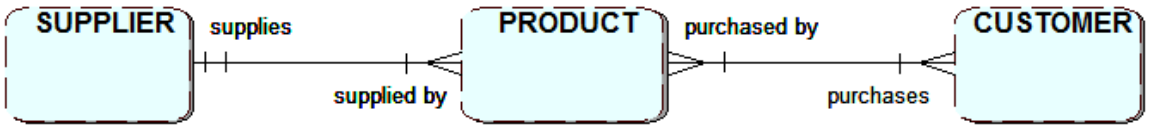
Summary of the rules for deriving the relations and their keys from the relationships:

1:1, the FK can go in either relation or combine to form one relation
1:1 with optionality at one end, the PK from the mandatory end is stored as FK at optional end
1:1 with optionality at both ends, create a new relation including a FK from either end used as PK
1:M, the PK from the one end goes in the many end as a FK
1:M optionality at the one end, create a new relation with FK from many end it becomes the PK
M:M, create a new relation with a compound PK made up of the FKs from both relations

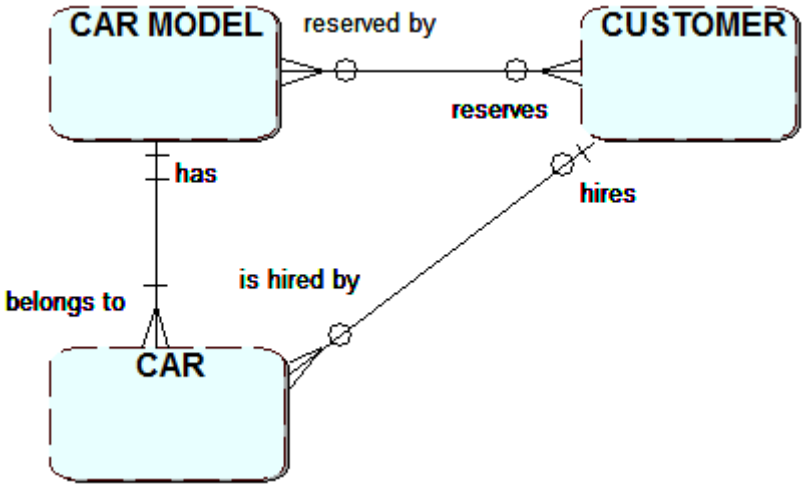
Key things to bear in mind are:
- tables should not have repeating attributes
- a foreign key should never hold a null value.

Exercise 6

1. **Produce the logical design for the following ERD.**



2. **Produce the logical design for the following ERD.**



It should be noted that this diagram has been drawn for a system that only needs to store information on cars currently hired (i.e. on loan now).

3. **Sub type task**
   Identify some sub types for the car hire model above and redraw the model.

4. **Exclusive arc task**
   Model and show the relations for the relationship "A student studies either an undergraduate course or a postgraduate course".

Exercise 6 feedback

1. The M:M relationship between PRODUCT and CUSTOMER is resolved by creating a new relation called PURCHASE giving the following relations:

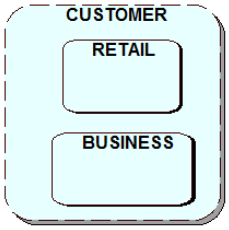
SUPPLIER (**Supplier_code,** Name, ..)
PRODUCT (**Product_ID**, Name, Price, *Supplier_code,* ..)
CUSTOMER (**Customer_code,** Name, Address, ..)
PURCHASE (**Product_ID, *Customer_code,*** quantity)

2. CAR MODEL (**Model_name**, manufacturer, size,…)
CAR (**Registration_no,** …)
CUSTOMER (**Customer_code**, Name, Address, ..)
RESERVATION (***Model_name, Customer_code***,…)
HIRE (**Registration_no**, *Customer_code,* ..)

HIRE is created to handle the relationship between CUSTOMER and CAR (as customer is optional). RESERVATION is created to handle the M:M relationship between CAR_MODEL and CUSTOMER.

Note: In reality, further information would need to be stored for HIRE and RESERVATION e.g. hire and return date etc.

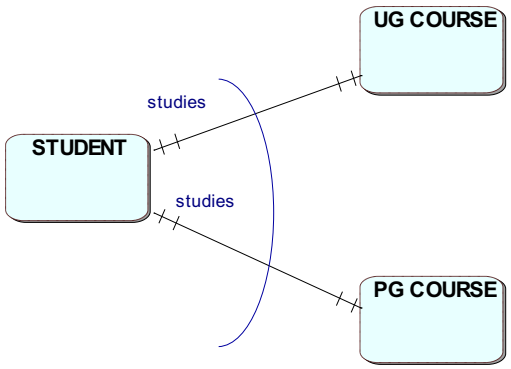
3. For CUSTOMER you may want to define the sub types RETAIL customer and COMMERCIAL borrower.



CUSTOMER (**Customer_code,** Name, Address,..)
RETAIL (***Customer_code,*** preferences,….)
BUSINESS (***Customer_code***, Company_name,….)

All sub type relations **must** have the same primary key (also a foreign key) as the super type. Additionally, you may consider sub types for CAR MODEL.

4. A Student studies either an undergraduate course (UG) or a postgraduate course (PG).



The UG COURSE and PG COURSE entities could also be shown as sub types of COURSE

STUDENT (**student_id**, …..)
UG_COURSE (**ug_course_id,** *Student_id*)
PG_COURSE (**pg_course_id,** *Student_id)*

5 Normalisation

On completion of this chapter you should be able to:

- understand the differences between normalised and un-normalised data
- convert un-normalised data to third normal form.

5.1 Introduction

You have now seen how a system can be modelled from a **"top-down"** perspective, starting with entities, then producing an entity relationship diagram and finally converting this into a set of relations. Although the top-down approach is the initial approach adopted, another modelling approach called **"bottom-up"** can be used to ensure that all the relations have been identified correctly. The bottom-up approach is based on looking at the attributes which exist in the system being analysed and grouping them into logical relations.

In this chapter the 'bottom-up' approach will be introduced to enable you to become familiar with the important general idea of "normal forms" of data and the process of **normalisation**.

Normalisation is a process undertaken to minimise data redundancy and produce efficient table structures. It is a formal technique for analysing **individual** relations and was introduced by (E.F. Codd, 1970). A set of steps is followed to ensure that the relation is transformed through a number of states into a form which is generally referred to as **'normalised'**.

At the logical design stage relations can be in one of four states, depending on whether certain conditions are satisfied.

These states are:

- **Un-normalised (UNF),**
- **1st Normal Form (1NF),**
- **2nd Normal Form (2NF),**
- **3rd Normal Form (3NF).**

Each state imposes further conditions on those required by the previous one.

The process of normalisation is used to produce a logical design which will lead to an efficient and effective database.

In particular, the aim is to avoid in the database:

- unnecessary wastage of storage space
- data redundancy (or data duplication – data (attributes) held in more than one place).

It is important to avoid data redundancy – storing the same data in more than one place has the potential to produce anomalies that can arise when data is inserted, amended or deleted. This is to be avoided in order not to compromise the integrity of the system. For example, if a customer's details were stored in many different places and they were not all updated at the same time this could cause inconsistencies which would be likely to result in out of date information being used.

If the ERD has been produced correctly, the resulting logical design will already be normalised to some extent. In this case, the normalisation process is used to check the definitions of the relations and, if necessary, refine them.

If the database is designed using a bottom-up approach where existing system documents, e.g. forms and reports have been considered, the initial logical design is unlikely to be normalised and applying the normalisation process is likely to cause changes to the original set of relations.

You are now going to be introduced to the process of normalisation and its normal forms.

5.2 Un-normalised form (UNF)

An un-normalised table can be defined as having any of the following issues:

- there are repeating groups (of attribute names)
- the attribute values are not atomic (single)
- there are "embedded tables"

To start the process you have to identify whether the data is in un-normalised form. The following is a representation of a typical business invoice. The invoice, which has a number of invoice item lines shown on it, will be used to illustrate the process.

Yorkshire Computer Supplies
1 Long Road, Leeds, LS3 3QS, West Yorkshire, UK. Tel: 0113 2832700

INVOICE

Invoice No: 1034
Invoice Date: 31/1/2015
Customer No: C101

Invoice Name / Address
H. Jones
9 The Avenue
Harrogate
HG2 7LR

ITEM ID	DESCRIPTION	QTY	PRICE	AMOUNT
PC1	Computer	3	500.00	1500.00
MN2	Monitor	3	200.00	600.00
LP1	Printer	1	156.00	156.00
		SUBTOTAL		2256.00
		TAX @ 20%		451.20
		DELIVERY		35.00
		TOTAL		2742.20

The first step is to identify a list of attributes including an identifying attribute from the invoice that you want to hold within the database system:

- **Invoice_No**
- Invoice_Date
- Customer_No
- Invoice_Name
- Invoice_Address
  (Item_ID
- Description
- Qty
- Price
- Amount)
- Subtotal
- Tax
- Delivery
- Total

The attributes shown between the (…) form a repeating group.

Note that the company name, address and telephone number for Yorkshire Computer Supplies are not included on the list as the system is only being used by them, so this information does not need to be stored in the database.

Although it is possible your lists of attributes may already be in a normalised form you should always apply the rules of normalisation to ensure that all your relations have been normalised before progressing to the physical design stage.

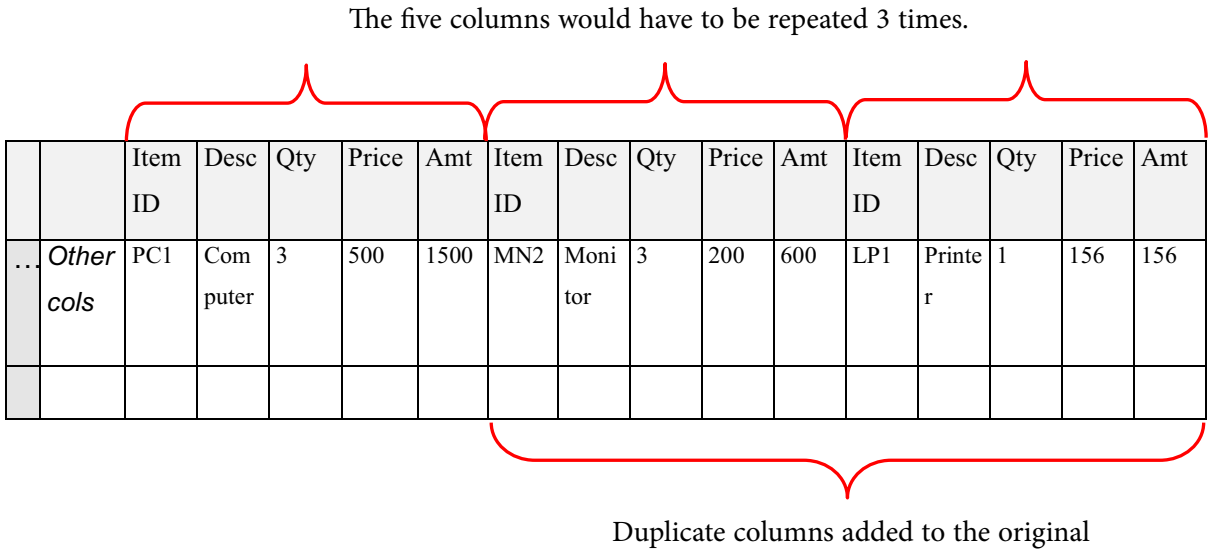
The problems of trying to build a single table with all of the un-normalised data attributes are explained below.

The INVOICE table created from the attributes identified shows that for the following columns Item_ID, Desc, Qty, Price and Amount the data is not atomic – the cells contain more than one attribute value.

Inv No	Date	Cust No	Inv Name	Inv Addr	Item ID	Desc	Qty	Price	Amt	Sub Tot	T a x	D e l	Total
1034	31/1/15	C101	H Jones	9 The Avenue Harrogate HG2 7LR	PC1 MN2 LP1	Computer Monitor Printer	3 3 1	500 200 156	1500 600 156	2256	…	…	2742.20
1035	…	…	…	…	…	…	…	…	…	…	..	..	…

You can also see that, just as the original invoice document included a table of values for Item_ID, Description, Qty, Price and Amount showing three rows of values, this table still appears to be embedded in the INVOICE table. These issues mean that it would not be appropriate to implement this table within the database as it would prove difficult to extract the individual invoice items from it.

In order to remove the embedded table it would be possible to redesign the table with repeating groups of attributes. The data could be rearranged by moving all the groups of invoice item data together in one long row, as illustrated by the following:



If this approach was adopted you would need to decide on the maximum number of invoice items per invoice. However, if a large number was chosen the cells would often be left empty because not all invoices would have the maximum number of items, thus wasting storage space. This also makes searching for data more complicated and if there was a need in future to increase the number of repeating groups, the table structure would need altering. Having considered the issues for this example you can now see that it is in an un-normalised state and so will need normalising to avoid these issues.

Exercise 1

a) In the invoice table below can you identify the embedded table?

Invoice No	Invoice Name	Invoice Address	Item ID	Description	Qty	Unit Price	Amount	Del
1034	H. Jones	9 The Avenue Harrogate HG2 7LR	PC1	Computer	3	500.00	1500.00	35.00
			MN2	Monitor	3	200.00	600.00	
			LP1	Laser Printer	1	156.00	156.00	
1044		Etc.						

b) It might be considered that a way round the problem would be to design the table in the following way:

Invoice No	Date	Invoice Name	Invoice Address	Item ID	etc.	Del
1034	31.01.15	H. Jones	9 The Avenue Harrogate HG2 7LR	PC1		35.00
1034	31.01.15	H. Jones	9 The Avenue Harrogate HG2 7LR	MN2		35.00
1034	31.01.15	H. Jones	9 The Avenue Harrogate HG2 7LR	LP1		35.00
1044	tc.					

Can you see why this should not be allowed?

Exercise 1 feedback

a) **The embedded table**

Item ID	Description	Qty	Unit Price	Amount
PC1	Computer	3	500.00	1500.00
MN2	Monitor	3	200.00	600.00
LP1	Laser Printer	1	156.00	156.00

b) The amended table should not be allowed as all the non-item columns Invoice_no, date, Invoice_Address, Sub total, Tax, Delivery and Total would be repeated for each item, a very wasteful situation.

5.3 First Normal Form (1NF)

First Normal Form

A relation is in First Normal Form if all attributes are functionally dependent on the primary key. In other words, for each value of the primary key there is only one value for each attribute in the relation. Or, put simply, if you entered a Student_ID to search the database you would expect it to return just one set of student attributes.

To convert a table to First Normal Form, multiple values or repeating groups of attributes must be removed to form another relation. To do this the following steps are carried out:

1. Identify the primary key for the relation (you may have to invent one)
2. Identify the repeating group or groups
3. Remove the repeating group completely from the original relation and place it in a newly- created relation
4. Ensure the relations are linked by putting the primary key of the original relation into the new one as a foreign key
5. Define a primary key for the new relation. This usually consists of two attributes, one of which is the Foreign Key, but there could be more than two.

To help with the conversion process you can use the Normalisation Template available in Appendix D.

To convert to First Normal Form, start with the original un-normalised list of attributes and identify a unique identifier for the whole list of attributes. This becomes the primary key; in this example it is **Invoice_No.**

The next step is to look for any repeating groups of attributes. In this example each group of 'invoice items' will be repeated for each item being invoiced. The repeating group consists of Item_ID, Description, Qty, Price and Amount and is shown in ( ) in the list of attributes. The repeating group is removed from the original group and placed in another relation.

In some situations there may be no obvious unique identifier attribute(s). In these cases you can invent an artificial identifier to act as the primary key, e.g. for a list of student details you could choose Student_ID.

If you find more than one repeating group of attributes, each group must be placed separately in its own relation.

You can now see how the original list of attributes has been broken down into two separate lists, therefore a foreign key needs to be identified for the new relation; this will be the primary key of the original relation Invoice_No. A foreign key is needed otherwise there will be no way of linking the new relation with the original one

Un-normalised form	First normal form
Invoice_No	**Invoice_No**
Date	Date
Customer_No	Customer_No
Invoice_Name	Invoice_Name
Invoice_Address	Invoice_Address
(Item_ID	Subtotal
Description	Tax
Qty	Delivery
Price	Total
Amount)	
Subtotal	***Invoice_No***
Tax	**Item_ID**
Delivery	Description
Total	Qty
	Price
	Amount

Finally, you need to ensure that the new relation is assigned a primary key. This will be a **compound key** which will use the foreign key and another attribute(s) which is chosen from the new relation in order to provide a unique identifier. In this example **Invoice_No** and **Item_ID** are used together.

The resulting attribute lists are in effect relations, though they need to be given meaningful names. The original relation is now named as INVOICE and the new relation INVOICE_ITEM.

The following show how these 1NF database tables based on these relations would look.
You could create an INVOICE table:

Invoice No	Date	Customer No	Invoice Name	Invoice Address	Sub Total	Tax	Delivery	Total
1034	31.01.15	C101	H. Jones	9 The Avenue Harrogate HG2 7LR	2256.95	451.20	35.00	2742.20

and an INVOICE_ITEM table:

Invoice No	Item ID	Description	Qty	Unit Price	Amount
1034	PC1	Computer	3	500.00	1500.00
1034	MN2	Monitor	3	200.00	600.00
1034	LP1	Laser Printer	1	156.00	156.00

You can see that for order number 1034 there is one row in the INVOICE table and three in the INVOICE_ITEM table. There are no repeating columns and no empty cells.

Exercise 2

A manufacturer obtains materials from a number of different suppliers which are used to make up products. The information about the suppliers and what they can supply is shown below.

Material No	Description	Unit Of Ordering	Supplier No	Supplier Name	Supplier Address	Minimum Order Quantity	Delivery Time	Price
B301	1cm Steel Bar	Metre	S455	J. Jones plc	12 May St Bootle	1000	1 weeks	4.45
			S126	Fabric	2 Green St Leeds	1500	2 weeks	4.50
			S312	F. Dunn Ltd	Oddy Way Grimsby	500	2 weeks	4.67
B377	50mm Bolts	Each	S007	Metalco	Axo Works Back Lane Sheffield	3000	3 days	0.01
			S402	GST plc	31 Bridge Rd Leeds	2000	1 week	0.01

a) Mark the column range for the embedded table(s) in the MATERIAL table above.
b) List all the attributes in tabular form (UNF).
c) Convert the data from UNF to 1NF
d) Draw the conceptual data model (ERD) for the two resulting relations.

Exercise 2 Feedback

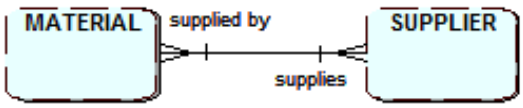
a)

Supplier No	Supplier Name	Supplier Address	Minimum Order Quantity	Delivery Time	Price

b) & c)

UNF	1NF
Material no.	**Material no.**
Description	Description
Unit of Ordering	Unit of Ordering
Supplier No.	
Supplier Name	***Material no.***
Supplier Address	**Supplier No.**
Minimum Order Quantity	Supplier Name
Delivery Time	Supplier Address
Price	Minimum Order Quantity
	Delivery Time
	Price

d)



5.4 Second Normal Form (2NF)

Second Normal Form
A relation is in Second Normal Form if –
- It is already in 1NF and
- All non-key attributes are fully functionally dependent on the whole key.

Conversion from 1NF to 2NF

This conversion **only** applies to relations that have a **compound key**. i.e. any relation with a simple (single attribute) key is already in 2NF, if it is already in 1NF.

The process involves checking whether or not there are attributes that depend on only **one part** of the compound key. To do this, follow the steps below:

1. Identify attributes with partial key dependencies
2. Remove the attributes with partial key dependencies into a new relation
3. Make the part key they are dependent on the primary key for the new relation

Do not forget to ensure that the original relation retains its compound key.

Consider the INVOICE_ITEM relation example which is in 1NF but is not in 2NF.

You are **not** interested in the INVOICE relation because its primary key is a single attribute, so in effect is already in 2NF. So, that just leaves the INVOICE_ITEM relation. The compound primary key for this relation consists of the **Invoice_No** and **Item_ID**. So you need to determine which, if any, of the **non-key** attributes are dependent on only **part** of this key.

The following table shows the dependencies:

Non-key attributes	Invoice_No + Item_ID	Invoice_No	Item_ID
Qty	✔		
Description			✔
Price			✔
Amount	✔		

You can see that the Description and Price attributes only depend on the Item_ID part of the compound key and so need to be removed to a separate relation.

Continuing the conversion using the template will produce the following results:

Un-normalised form	First Normal Form	Second Normal Form
Invoice_No	**Invoice_No**	**Invoice_No**
Date	Date	Date
Customer_No	Customer_No	Customer_No
Invoice_Name	Invoice_Name	Invoice_Name
Invoice_Address	Invoice_Address	Invoice_Address
(Item_ID	Subtotal	Subtotal
Description	Tax	Tax
Qty	Delivery	Delivery
Price	Total	Total
Amount)		
Subtotal	***Invoice_No***	***Invoice_No***
Tax	**Item_ID**	***Item_ID***
Delivery	Description	Qty
Total	Qty	Amount
	Price	
	Amount	**Item_ID**
		Description
		Price

Note that there are now two foreign keys defined (*Invoice_No* and *Item_ID*) and they would be used together to form a compound primary key in the new INVOICE_ITEM relation. If the 2NF relations were implemented as tables, the INVOICE table would remain unchanged and the INVOICE_ITEM table would now have fewer columns than before:

Invoice No	Item ID	Qty	Amount
1034	PC1	3	1500.00
1034	MN2	3	600.00
1034	LP1	1	156.00
1044	Etc…		

There would also be a new table which would be named PRODUCT:

Item ID	Description	Price
PC1	Computer	500.00
MN	Monitor	200.00
LP1	Laser Printer	156.00
Etc…		

The advantage of having data in 2NF is that redundancy has been removed. In 1NF for instance, you would have had to provide the value for the attribute "Description" in the INVOICE_ITEM table every time a new invoice was produced which had that particular product on it. In 2NF the value is provided once only in the PRODUCT table.

Exercise 3

a) Check the two relations resulting from the conversion to 1NF in Exercise 2.

b) Does one or both need converting?

c) Convert the data into 2NF by following the appropriate steps.

Exercise 3 Feedback

a), b) and c) Only the following 1NF table needs to be checked for 2NF as it has a compound primary key. This results in two tables at 2NF.

1NF	2NF
Material no.	***Material no.***
Supplier No.	***Supplier No.***
Supplier Name	
Supplier Address	**Supplier No.**
Minimum Order Quantity	Supplier Name
Delivery Time	Supplier Address
Price	Minimum Order Quantity
	Delivery Time
	Price

Notice that in 2NF the original 1NF Material relation only consists of the compound primary key, and the Supplier No. is now acting as a foreign key to the new SUPPLIER details relation.

5.5 Third Normal Form (3NF)

Third Normal Form

A relation is in Third Normal Form if -

- it is already in 2NF
- there are no functional dependencies between any pair of non-key attributes (i.e. there are no transitive dependencies)

If an attribute is functionally dependent on another attribute, that one is referred to as the **determinant**. For example:

Is Invoice No. a determinant of Invoice Name? Yes
Is Invoice Name a determinant of Invoice_No.? No

If an attribute is the determinant of a second attribute and that attribute is the determinant of a third attribute then the third attribute is **transitively** dependent on the first attribute.

Converting the data tables to the third normal form will further improve the logical design of the database.

Conversion to 3NF

This is a very similar process to the one for 2NF. To do this, follow the steps below:

1. Identify any attributes that are determined by another non-key attribute
2. Remove these attributes to a new relation
3. Set the non-key attribute to be the Primary key in the new relation
4. Convert the non-key attribute to a foreign key in the original relation.

The Third Normal Form is very similar to the Second Normal Form. However, instead of considering whether some attributes in the table are dependent on only part of a compound key, a check is made to see whether they are dependent on attributes which are **not** part of the key. In a very similar way to 2NF, you remove the attributes which depend on this (non-key) attribute into a new table.

As an example of a table being in 2NF but not in 3NF, consider the INVOICE relation:-

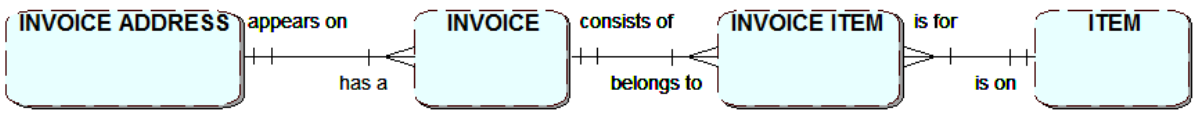
Invoice No	Date	Customer No	Invoice Name	Invoice Address	Sub Total	Tax	Delivery	Total
1034	31.01.15	C101	H. Jones	9 The Avenue Harrogate HG2 7LR	2256.95	451.20	35.00	2742.20

If you look carefully at this relation you will see that there is a functional dependency between the non-key attributes Invoice Name and Invoice Address. Invoice address is actually dependent on (or determined by) the Invoice Name, not by the Invoice No. The Invoice Address is said to be transitively dependent on the Invoice No.

This is not an ideal situation as every time an invoice for a customer was produced, the customer's address would be duplicated. By converting to 3NF the Invoice Name and Address attributes are removed to a new relation, and the Invoice Name would become the primary key of the new relation. The Invoice Name would remain in the existing table but would now be acting as a foreign key. The relations are now in 3NF and so you can say they have been 'normalised', thus providing you with a set of efficient table structures which can now be considered for use in the physical design stage. The completed template below shows all the relations and their names.

Un-Normalised Form UNF	First Normal Form 1NF	Second Normal Form 2NF	Third Normal Form 3NF	Relation Name
Invoice_No	**Invoice_No**	**Invoice_No**	**Invoice_No**	**INVOICE**
Date	Date	Date	Date	
Customer_No	Customer_No	Customer_No	Customer_No	
Invoice_Name	Invoice_Name	Invoice_Name	***Invoice_Name***	
Invoice_Address	Invoice_Address	Invoice_Address	Subtotal	
(Item_ID	Subtotal	Subtotal	Tax	
Qty	Tax	Tax	Delivery	
Description	Delivery	Delivery	Total	
Price	Total	Total		
Amount)			**Invoice_Name**	**INVOICE ADDRESS**
Subtotal	***Invoice_No***	***Invoice_No***	Invoice_Address	
Tax	**Item_ID**	***Item_ID***		
Delivery	Qty	Qty	***Invoice_No***	**INVOICE ITEM**
Total	Description	Amount	***Item_ID***	
	Price		Qty	
	Amount	**Item_ID**	Amount	
		Description		
		Price	**Item_ID**	**ITEM**
			Description	
			Price	

In order to see how the bottom-up approach compares with the top-down approach here is the corresponding ERD for the Invoice:



5.6 Denormalisation

Although you should always aim to implement a normalised set of tables where possible, there are occasions when you might consider denormalising, usually for performance reasons. In the following tables, if the Invoice Item's tax amount was a calculated value, the item's tax code would be used to search the Tax table for the appropriate row, and the corresponding tax % value would then be used to calculate the invoice item tax amount.

Item ID	Description	Quantity	Price	Total Price	Tax Code	Tax Amount
A101	Laptop	1	450	450	01	90

Invoice Item Table

Tax Code	Tax %
01	20

Tax Table

Denormalisation could be applied to speed up the calculation. The tax % amount could be stored within the Invoice Item table, thus saving a Tax table read, however the tax % value would now have to be maintained in both tables.

In this example there is another reason to consider denormalisation. Storing the actual tax % amount in the Invoice Item table would ensure that if the calculation had to be performed again in the future, when the tax rate in the Tax table may have changed, the result would still produce the same value which would have been calculated at the time the invoice item was initially created as it should.

5.7 Checking the model

Before moving on to the development phase it is important to check that the normalised set of relations will support the business transactions that are required for the database system. This is usually achieved by taking each transaction or search query and tracing the pathways from entity to entity on the composite ERD in order to ensure that it is possible to link to the relevant entities and extract the relevant attributes to satisfy the query.

5.8 Summary

The relations of the logical design can be improved in terms of reduction of wasted space and elimination of redundancy by the process of normalisation. Relations derived from ERDs may well be already normalised, but ones derived from other sources, e.g. forms or reports, will probably not be.

Normalisation means transforming the relations by stages into:

a) First Normal Form – to ensure that an implemented table would have only a single value for each attribute (there are no repeating groups).
To achieve 1NF: Remove the repeating group completely into a new table and place a link (Foreign Key) in the NEW table. Remember to identify the compound Primary Key in the new table.

b) Second Normal Form – the non-key attributes depend on the whole (compound) key, not just part of it.
To achieve 2NF: Remove the attributes that are dependent on only one part of the key with that part key into a new table, leaving a link (Foreign Key) behind in the ORIGINAL table.

c) Third Normal Form – all the attributes are dependent on the key.
To achieve 3NF: Remove any attributes which are dependent on another non-key attribute and place them in a new table leaving a link (Foreign Key) behind in the ORIGINAL table.

Do not normalise all the attributes as one data set, normalise each logical data set separately.

To help you remember the 3 normal forms, the attributes in a relation should depend on:

1NF the key
2NF the whole key
3NF nothing but the key

When you have normalised all the individual data sets, you should check to see if any relations identified are not shown as entities on the original ERD. If they are not, it is advisable to add them to form a complete (composite) ERD showing all of the data requirements.

Exercise 4

For each of the following statements decide which of the three normal forms would be applicable.

1. The non-key attributes depend on the whole (compound) key, not just part of it.
2. For each row of the table, there is a single entry in each column.
3. In a table in this form all non-key attributes are fully functionally dependent on the key.

Exercise 4 feedback

1)	The non-key attributes depend on the whole (compound) key, not just part of it	Second Normal Form
2)	For each row of the table, there is a single entry in each column	First Normal Form
3)	In a table in this form all non-key attributes are fully functionally dependent on the key	Third Normal Form

Exercise 5

A cycling organisation wishes to set up a database to record results for cycle races. The cyclists' ID, name, address and phone no. are to be held. In addition, their cycling club ID and name and its address need to be recorded. The database will also need to hold all details of all the races that the cyclist takes part in. Each race has a unique number. Each cyclist can only belong to one club though clubs can have many cyclists.

An initial un-normalised relation is:

CYCLIST (**Cyclist#**, Surname, Forename, Cyclist address, Cyclist_phone#, Race#, Race name, Race type, Race result, Club#, Club name, Club address)

Convert this to 3NF using the normalisation template. A starting point is given in the table below.

Un-normalised form	First normal form	Second normal form	Third normal form
Cyclist_ID			
Surname			
Forename			
Cyclist_address			
Cyclist_phone#			
Race#			
Race_name			
Race_type			
Race result			
Club#			
Club_name			
Club_address			

Exercise 5 feedback

Un-normalised form	First normal form	Second normal form	Third normal form
Cyclist ID	**Cyclist ID**	**Cyclist ID**	**Cyclist ID**
Surname	Surname	Surname	Surname
Forename	Forename	Forename	Forename
Cyclist_address	Cyclist_address	Cyclist_address	Cyclist_address
Cyclist_phone#	Cyclist_phone#	Cyclist_phone#	Cyclist_phone#
Race#	Club#	Club#	*Club#*
Race_name	Club_name	Club_name	
Race_type	Club_address	Club_address	**Club#**
Race result			Club_name
Club#	***Cyclist ID***	***Cyclist ID***	Club_address
Club_name	**Race#**	***Race#***	
Club_address	Race_name	Race result	***Cyclist ID***
	Race_type		***Race#***
	Race result	**Race#**	Race result
		Race_name	
		Race_type	**Race#**
			Race_name
			Race_type

The repeating group race is broken out at 1NF.
The part-key dependent race details are broken out at 2NF.
The non-key dependent club details are broken out at 3NF.

This would give the following set of relations:
CYCLIST, CLUB, RACE_RESULT, RACE,

Exercise 6

Normalise the following:

1. EMPLOYEE (Employee Name, Address, Age, Department, Division)

2. An airline wishes to keep the following information about each of the flights made:

   Fight Reference Number, Departure Date, Pilot No, Arrival Date, Aircraft ID, Flight Destination, Name of Pilot, Aircraft Name, Aircraft capacity, Aircraft Type, Aircraft max speed.

3. TEACHER (Teacher#, Teacher_Name, School_Ref, School_Name)

4. RACE(Race_ID, Competitor_ID, Competitor_Name, Position_Achieved, Race_Distance,)

5. CAR_RALLY_RESULT(Driver_ID, Driver_Name, Co_Driver_ID, Co_Driver_Name, Driver_Ranking, Rally_Name, {Stage_No, Stage_Time}

   Note: Rally Cars compete in motor rallies. Each rally car has a driver and a co- driver.
   Each rally comprises a number of timed stages { }.

6. Derive a set of normalised relations (to 3NF) for the following data that is to form the basis of a database used to hold computer repair records.

Customer Number, Customer Name, Customer Address, Repair Date, PC ID, Make, Model, Technician Name, Technician Grade, Repair Cost.

Assume that each PC repair is assigned to one technician whose grade determines the rate to be charged for the work done. The database must be able to record a series of repairs carried out on the same PC over a period of time. The company identifies each PC with a single owner (the Customer).

Exercise 6 feedback

1. EMPLOYEE (**Employee_Name**, Address, Age, ***Department***)
   DEPARTMENT (**Department**, Division)

2. PILOT **(Pilot#**, Pilot_Name)
   AIRCRAFT(**Aircraft_ID**, Aircraft_Name, Aircraft_Type, Aircraft_Capacity, Aircraft_max _Speed)
   FLIGHT(**Flight_Reference_No.**, Flight_Destination, Departure_Date, Arrival_Date, ***Aircraft_ID, Pilot#)***

3. TEACHER**(Teacher#**, Teacher_Name, ***School_Ref***)
   SCHOOL(**School_Ref**, School_Name)

4. RACE **(Race_ID**, Race_Distance)
   COMPETITOR (**Competitor_ID**,Competitor_Name)
   RESULT (***Race_ID, Competitor_ID,*** Position_Achieved)

5. DRIVER(Driver_ID, Driver_Name)
   CO_DRIVER(Co_Driver_ID, Co_Driver_Name)
   RALLY(***Rally_Name, Driver_ID***, ***Co_Driver_ID***)
   CAR_RALLY_RESULT (***Rally_Name, Driver_ID*** , **Stage_No**, Stage_Time)

6. PC_REPAIR ***(PC_ID,Repair_Date***, ***Customer#, Technician_Name,*** Repair_Cost)
   CUSTOMER (**Customer#**, Customer_Name, Customer_Address)
   PC (**PC_ID,** Make, Model)
   TECHNICIAN (**Technician_Name**, Technician_Grade)

6 Introduction to Oracle SQL

On completion of this chapter you should be able to:

- sign in and out of Oracle
- use a script file
- create tables
- insert data.

Introduction

You are now ready to implement physically a logical design, using a database management system (DBMS). There are a number of Relational Database Management Systems (RDBMS) in use; popular ones include Oracle, MySQL, Microsoft SQL Server and Microsoft Access.

In order to use a DBMS you need to be familiar with a data manipulation language. The most widely used one is the Structured Query Language, commonly referred to as **SQL**.

Although SQL has become a standard and is used by many DBMSs there are slight differences between them so some SQL code may not work on all systems without changes

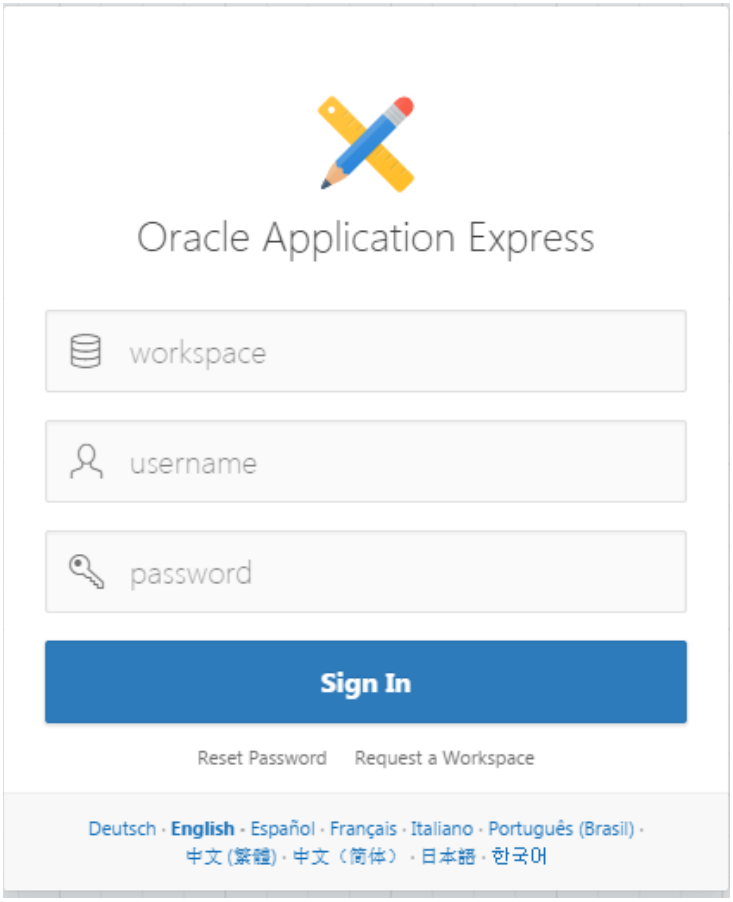
This book uses the Oracle database with the Oracle Application Express (APEX) version 5 toolset to execute the presented SQL commands.

A free Oracle account can be obtained from apex.oracle.com. Alternatively, a copy of Oracle Database Express Edition 11g and APEX can be downloaded from apex.oracle.com for installation on your own PC. A web browser is used to access APEX and the database.

The following chapters will introduce you to the main SQL commands; a full Oracle SQL language reference can be found at **docs.oracle.com.**

Signing In

To sign in to the database, open your web browser and go to **apex.oracle.com** or your system's own login page. You will need to enter your database **workspace name, username and password**. These will be provided to you when you open an Oracle account or will be given to you by your system administrator. The password can be reset by clicking on the Reset password option on the sign-in screen if required. Your workspace within the Oracle database is where your data tables and SQL commands will be stored.



Oracle APEX Sign In screen

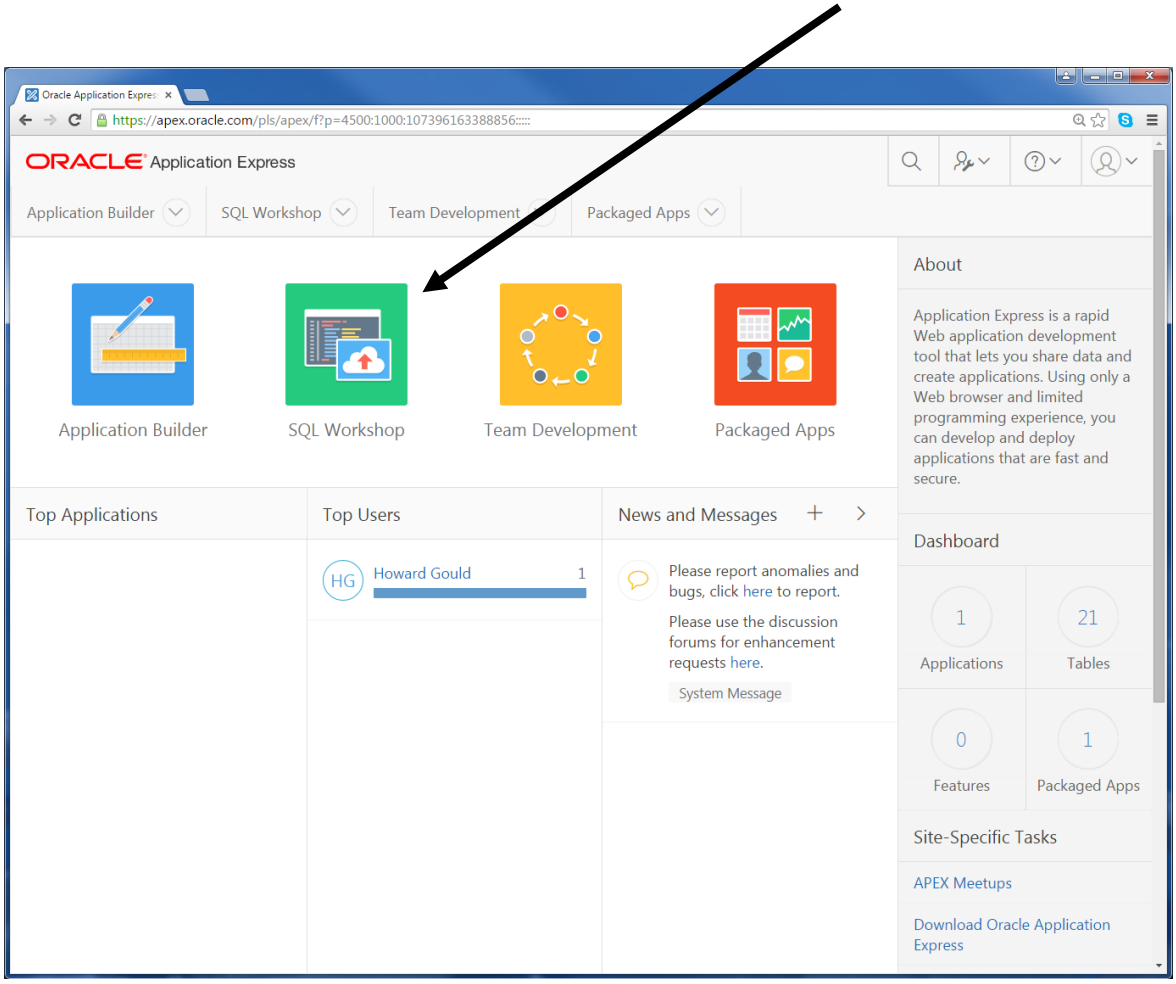
Script files

Although you can enter SQL commands directly using the APEX SQL Command Tool, when developing applications it is usual practice to create script files to store regularly used SQL commands along with any supporting comments, e.g. commands for creating database tables.

A script file allows you to quickly retrieve, edit and rerun SQL commands. It is basically a simple text file and can, if required, be created outside of APEX using any text editor such as Microsoft Notepad.

> Do not confuse script files with database tables, even though they can both have the same name.
> Script files should be saved with a suffix of **.sql** to avoid confusion with other file types.

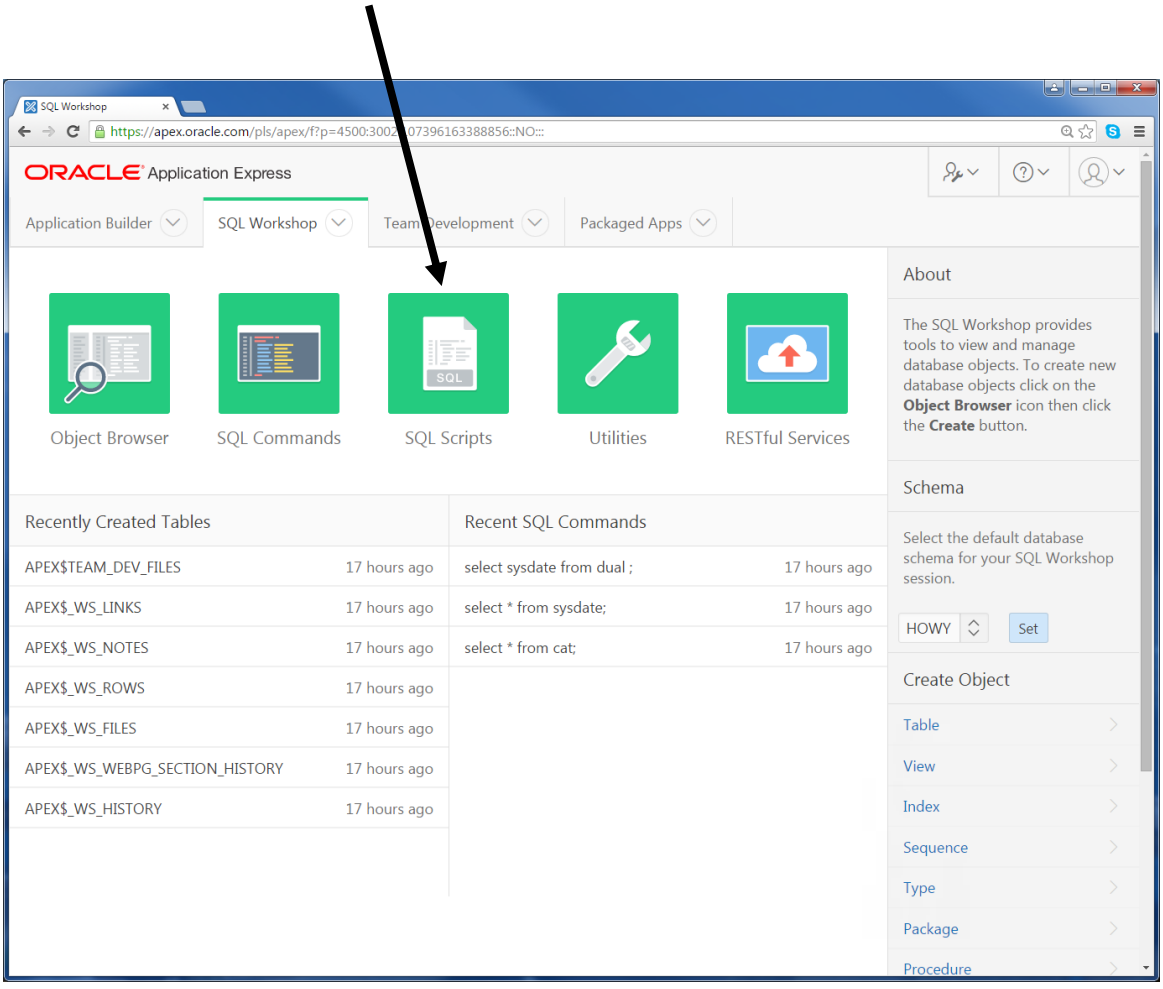
1. Once signed in to APEX, on the Home page click on the SQL Workshop icon.



Apex Home Page

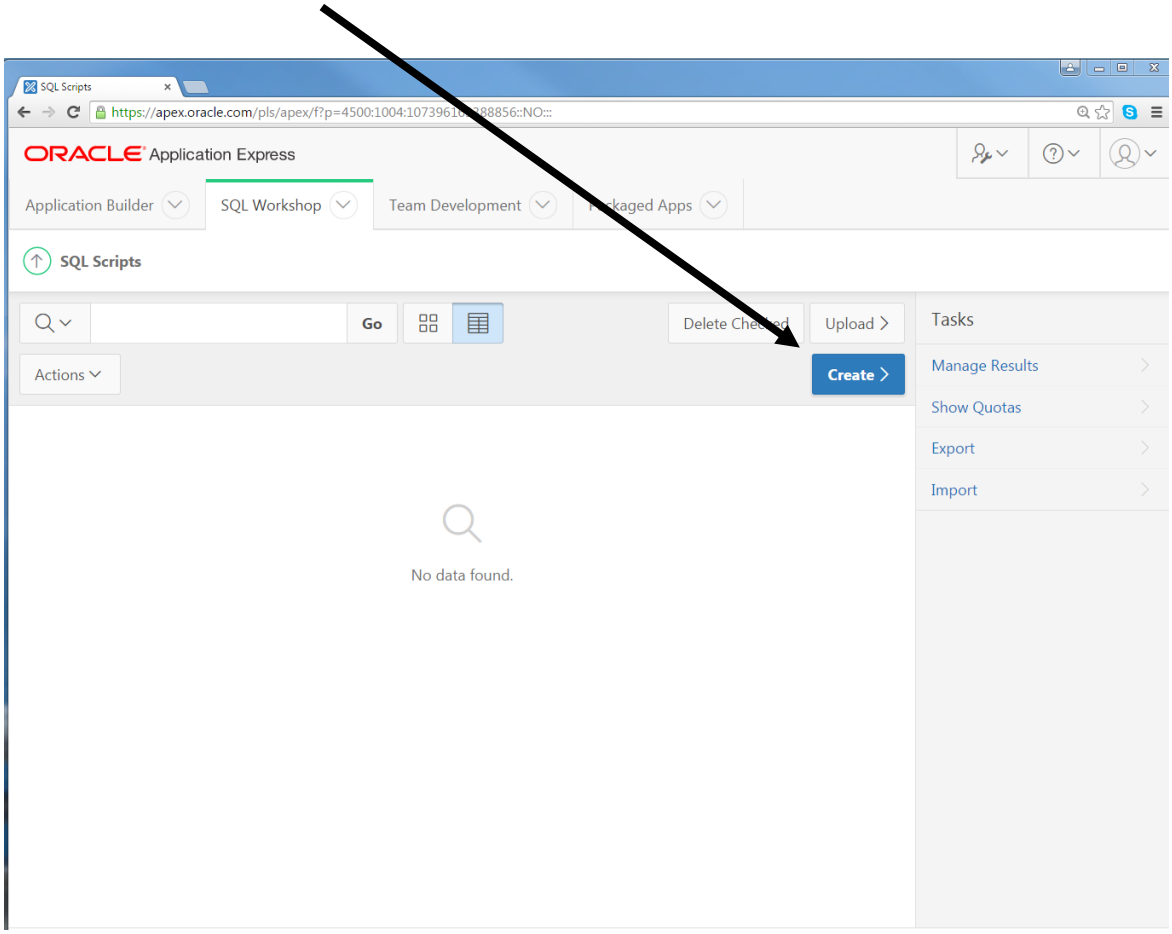
The following screens will show you how to create and run a script file called music.sql which contains SQL commands that will create four tables in your database workspace relating to a music system (see Appendix B). The system will hold data about music companies that have produced music CDs, details of each CD, the tracks (songs) on each CD and music categories which tracks belong to.

2. Click on **SQL Scripts**.



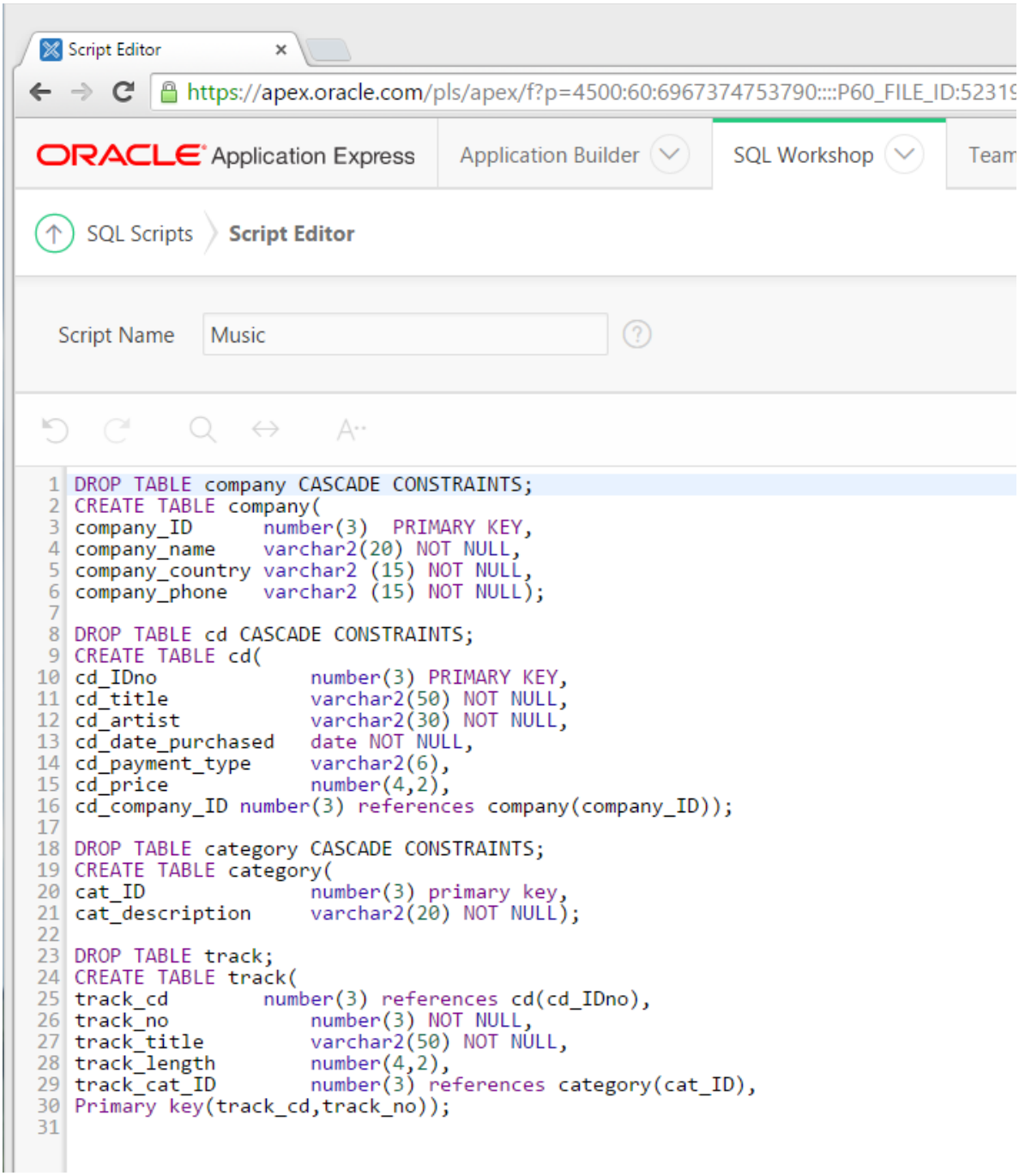
SQL Workshop Page

3. Then click on **Create** >.



SQL Scripts Page

4. Enter **music** in the Script Name box and then position the cursor in the editor area and enter all the SQL commands from the **music.sql** script file in Appendix B. Take care when entering these commands to ensure that they are all entered exactly as shown below.

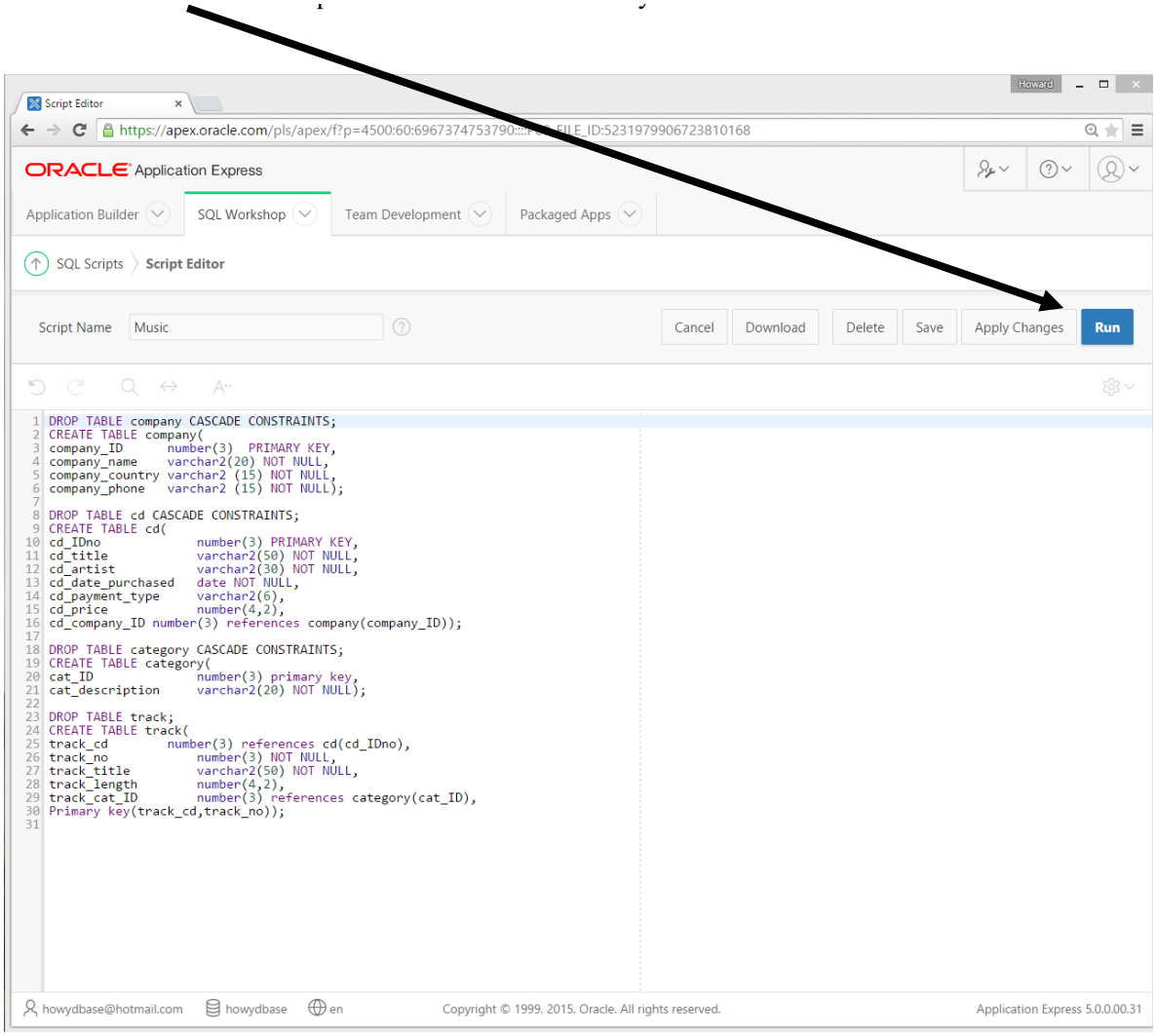


Script Editor

Alternatively, you can create a script file outside of Oracle using a simple text editor such as Microsoft Notepad. Enter your SQL commands using the editor and save the file with a **.sql** suffix. To upload the script into APEX go to the **SQL Scripts** tool and click on the **Upload** button, then choose the file name, enter a script name and click on the **Upload** button.

Comments can be included in script files by preceding them with --

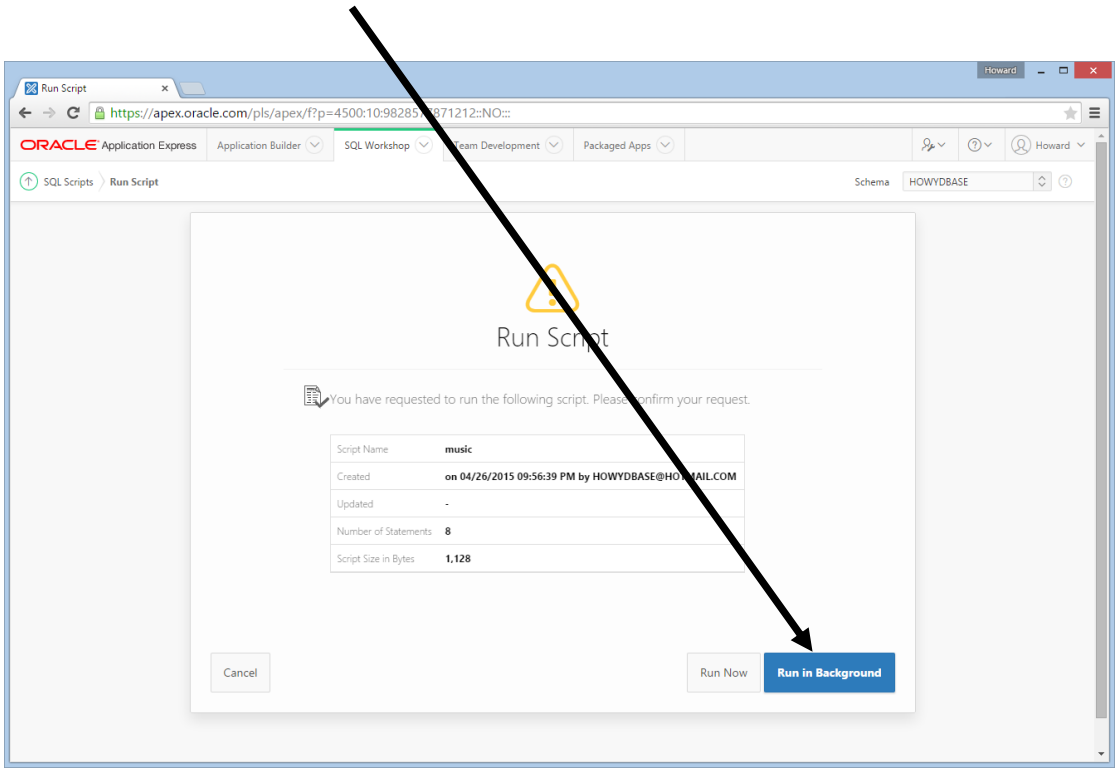
5. You are now ready to run the script file to execute its SQL commands; click on the Run button. The script will be saved automatically.



Running a Script

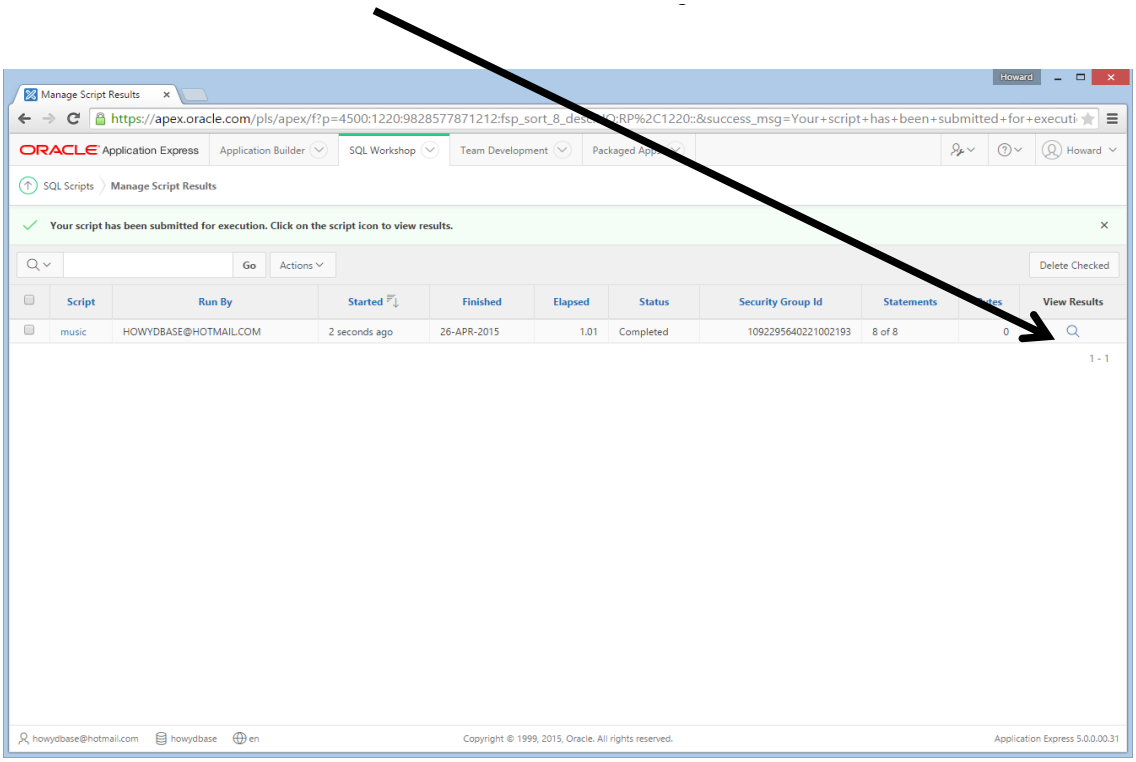
To delete a script on the SQL Scripts page just tick the small box to the left of the script detail and click on the **Delete Checked** button, then click OK when prompted to perform the delete action.

6. Click on the **Run in Background** button (or Run Now)



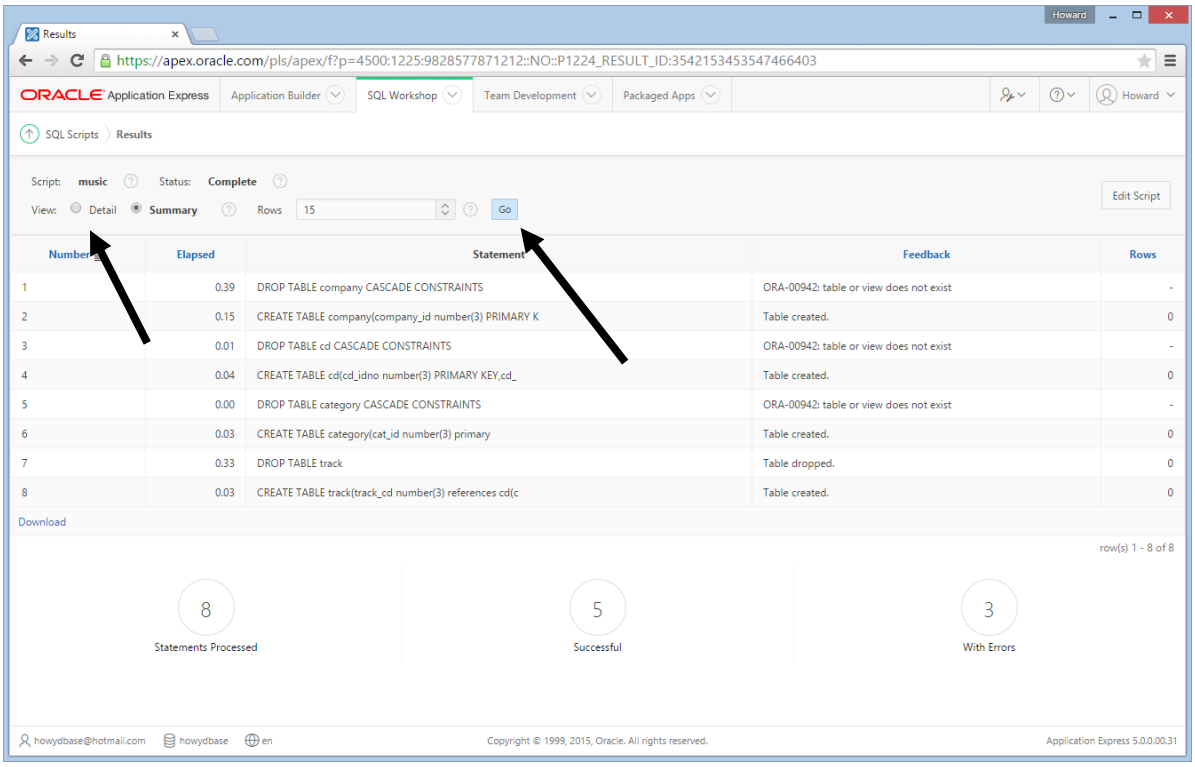
Run Script

7. Click on the View Results icon for the music script



Manage Script Results

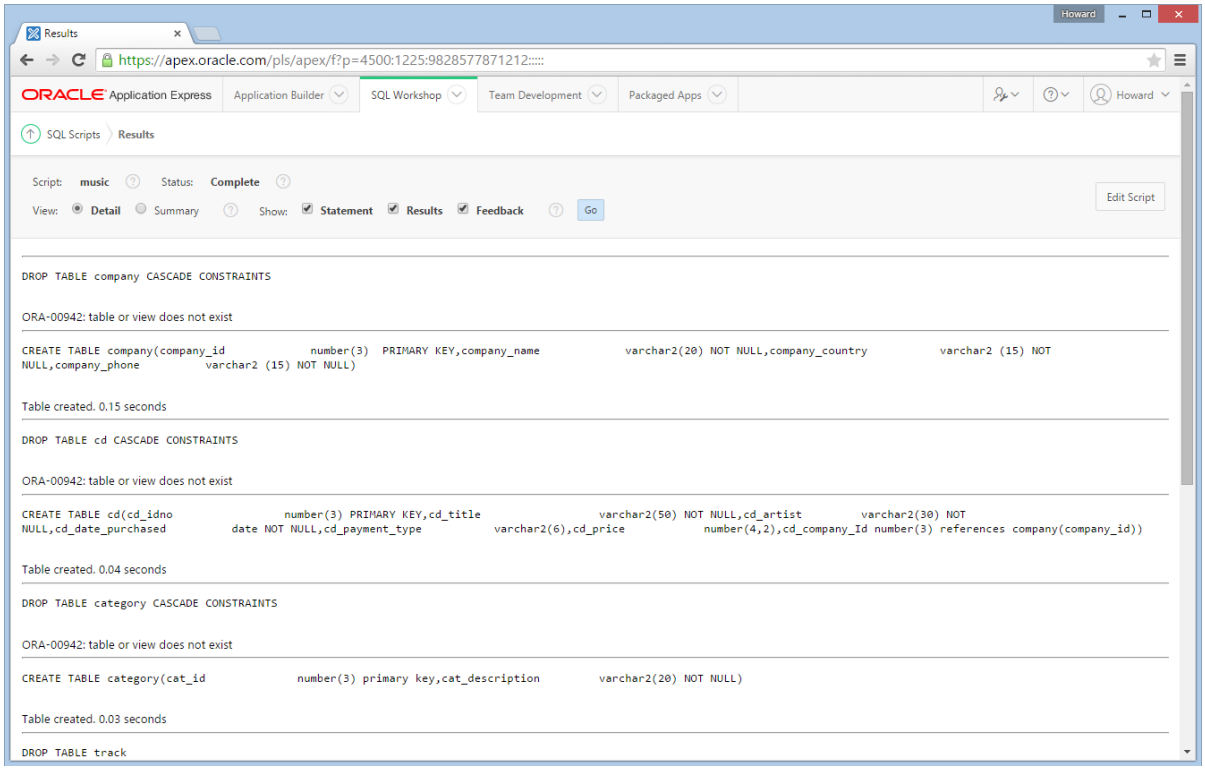
A brief summary of the SQL commands with the execution feedback is displayed



Summary Script Results

8. To see more details click on the **View: Detail** button and click on the **Go** button

The **music** script commands appear along with the Oracle run time messages.



Detail Results View

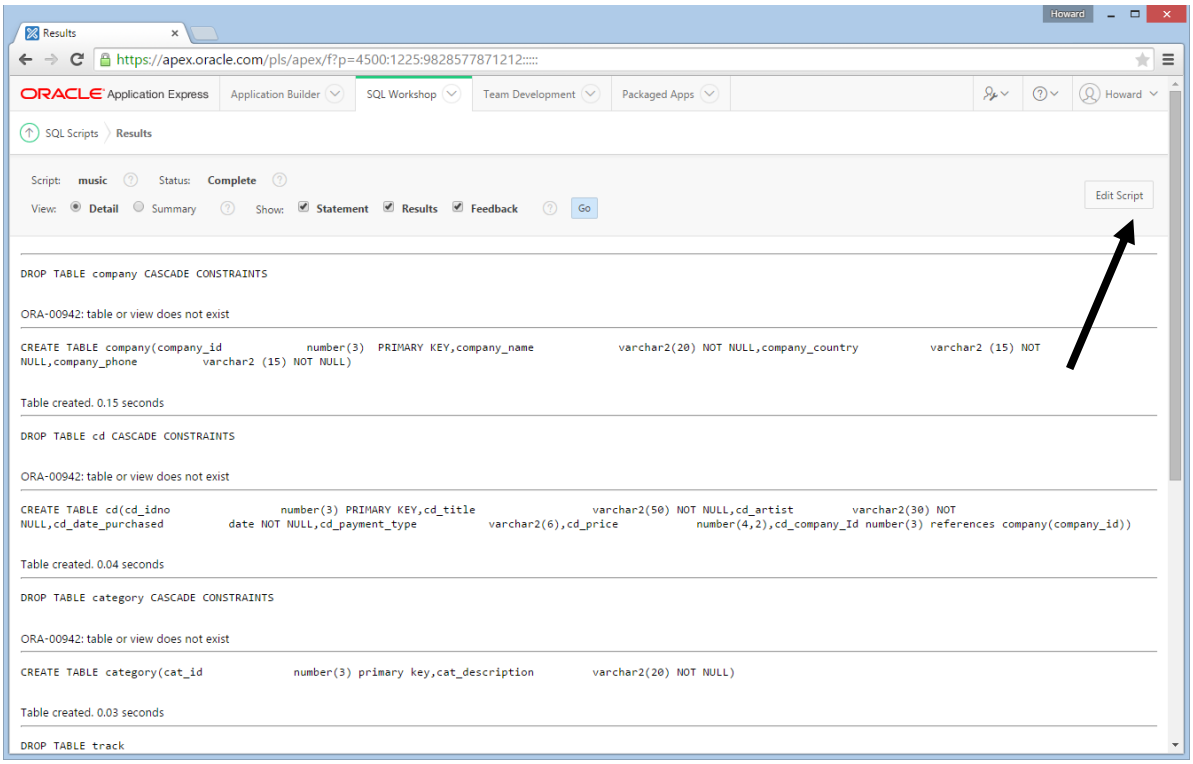
On examining the SQL run time messages you will see that four tables have been created. Each table has a number of named columns (attributes) and each has a particular datatype. There are a number of different datatypes available but these tables just use three. The VARCHAR2 datatype is used to hold variable length character strings; the number inside the ( ) specifies the maximum number of characters. The NUMBER datatype can be used to hold an integer value (a number with no decimal places) or a real number (includes decimal places). The DATE datatype holds dates.

The first time you run this script the following error message will appear for each table – "ORA-00942: table or view does not exist." This is not a problem as these errors are generated by the DROP TABLE commands because Oracle is trying to drop (remove) the tables from the database, but as they do not yet exist it cannot do this. However, it is usual practice to include a DROP TABLE command before a CREATE TABLE command as you cannot overwrite an existing table with the CREATE TABLE command.

Therefore you should always DROP a TABLE before you try and CREATE it, in case there is one already there.

If you need to amend the **music** script, click on the **Edit Script** button.
Make any amendments you require to the SQL and then click on the **Save** button.

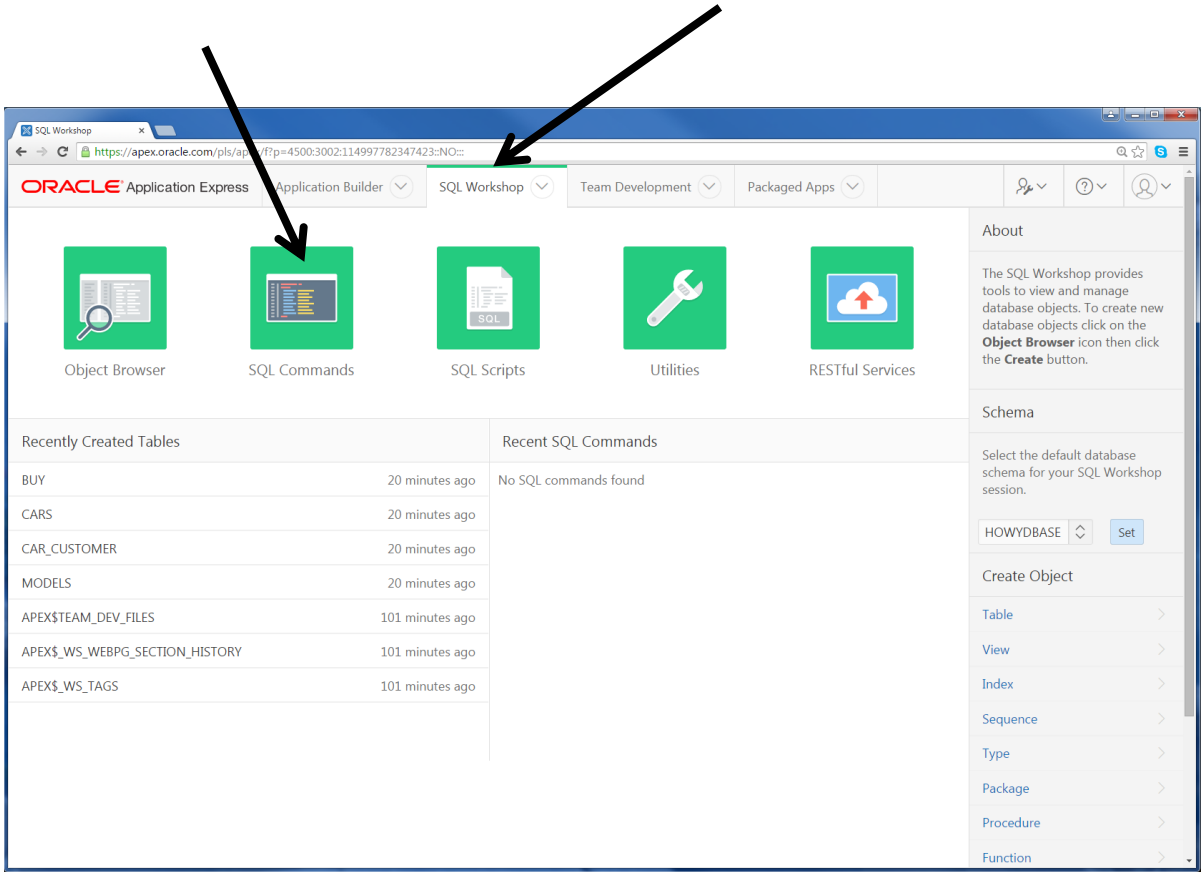
To run the script again, click on the **Run** button followed by the **Run in Background** button as in 5 above.



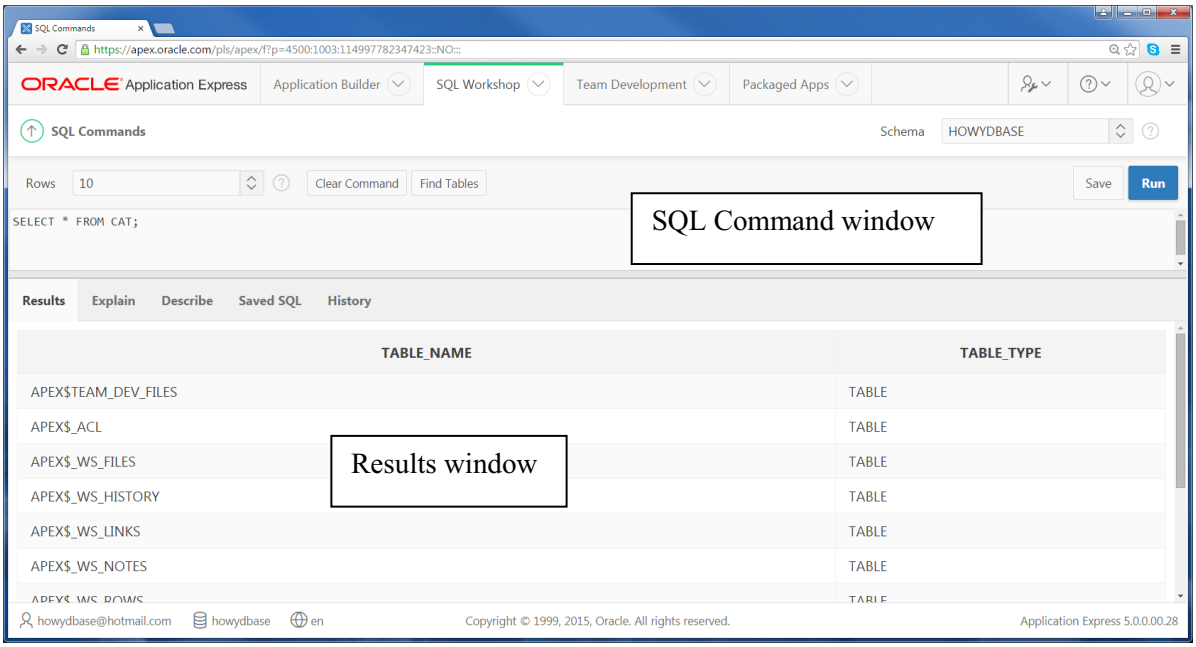
Script Results

Always remember to check the APEX output results and feedback messages to ensure that your SQL commands have run successfully.

To enter and run individual SQL commands directly, rather than using a script file, you can use the **SQL Commands Tool**. To access this, click on the **SQL Workshop** tab and then click on the **SQL Commands** button.



SQL Workshop



SQL Command and Results windows

APEX by default displays 10 Rows of output in the results window. This is not usually sufficient to see all of your output results, so alter the number of Rows displayed by clicking on the Rows box above the SQL commands window and choose a larger number, e.g. 50.

Creating tables

The SQL CREATE TABLE command is used to create a database table and defines its data columns. It has the following format:

CREATE TABLE *tablename* (*column_name* datatype [*constraint_name* constraint type], …..);

[ ] represents and optional clause

Table and column names must follow these rules:

- the table name must be unique
- it can be up to 30 characters long
- it must start with a letter
- it can have letters, digits and the symbols $, # and _
- the name cannot be a reserved word (e.g. an Oracle SQL command)

Columns must have a datatype and usually a size defined e.g.:

Fixed length character string	**CHAR**
Variable length character string	**VARCHAR2**
Numbers	**NUMBER**
Date	**DATE**
Timestamp	**TIMESTAMP**

Example column sizes for **CHAR, VARCHAR2** and **NUMBER** columns:

CHAR(3)	Holds strings of 3 characters (right filled with spaces)
VARCHAR2(25)	Holds strings up to 25 characters long (variable length no padding)
NUMBER(3)	Holds integers up to three digits long e.g. -999 to 999
NUMBER(5,2)	Holds real numbers e.g. -999.99 to 999.99 The total number of digits (5) is referred to as the precision and the decimal places (2) are referred to as the **scale**.

Constraints

Tables can have **table** and **row** level constraints. Once set, these are automatically applied to restrict the data held, in order to maintain the integrity of the database by applying relational database rules. The constraints are usually defined when the table is created, but can be applied later.

There are two categories of constraints, **integrity** and **value**.

An **integrity** constraint ensures that you can define a **Primary key** to ensure duplicate rows cannot be added to a table. A **Foreign key** can be defined which ensures a row must reference a primary key in the referenced table.

A primary key can be defined at column level as in the company table:

```
Company_ID  number(3) PRIMARY KEY,
```

Or at table level, usually when there is a compound key, for example in the track table:

```
PRIMARY KEY(track_cd,track_no));
```

Foreign keys can similarly be defined and will be discussed later.

The following **value** constraint types can be used to ensure either that a column does not have an empty (NULL) value or that it contains a specific value:

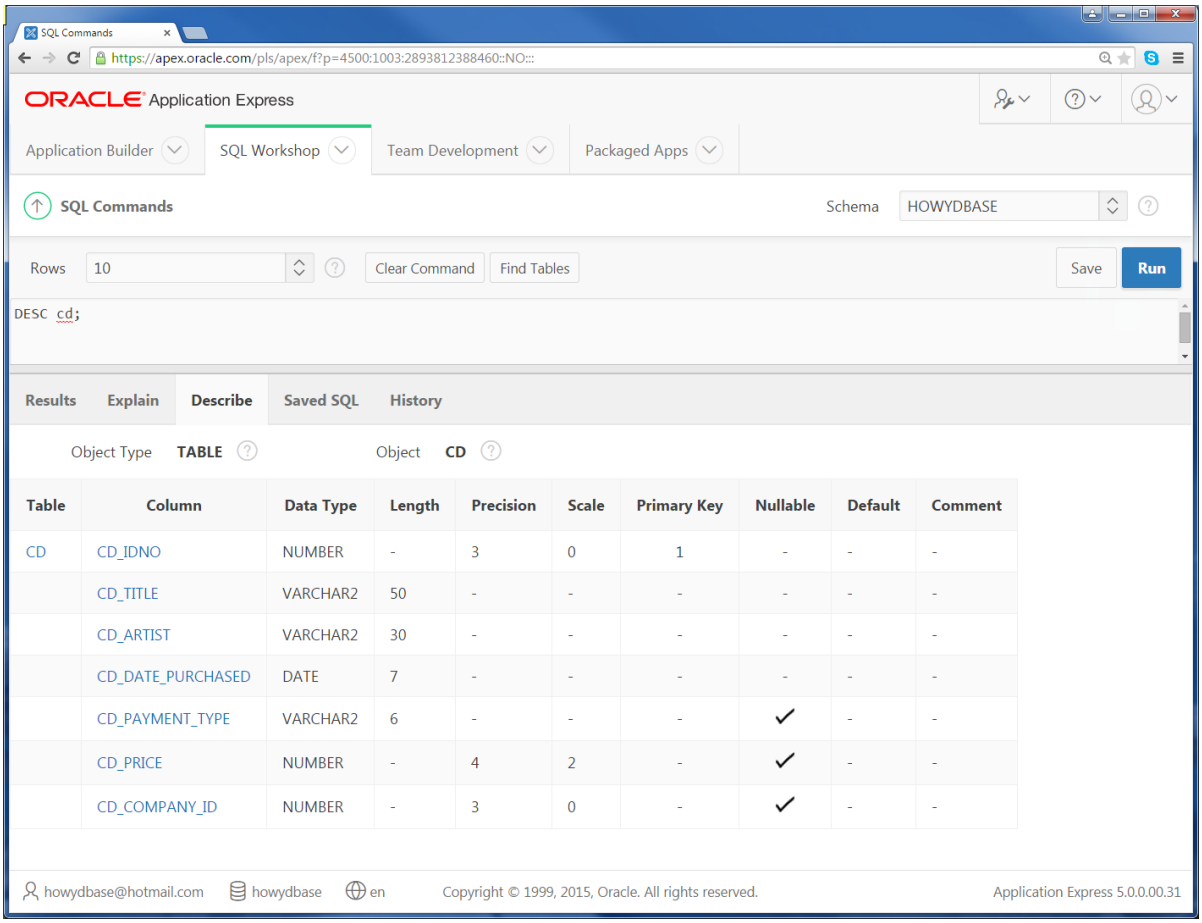
NOT NULL	ensures that a value must be entered (cannot be left empty)
CHECK	ensures that a column has a value within a set range e.g. CHECK (price > 10) and (price < 20)
UNIQUE	ensures that a single column has a unique value or that a group of columns is unique

If a column is defined as a PRIMARY or FOREIGN key then the column will not permit a NULL or empty value, so you should not specify the NOT NULL constraint as well.

Constraints are given names by the system, although you can assign your own.

9. Consider the **music.sql** script to see which constraints have been used and why.

10. To see the structure of a table you can also use the SQL **DESC** command in the **SQL Commands** tool e.g. **DESC cd;**



DESC cd

Inserting data

Now that you have created the music system tables you can enter rows of data into your tables. To do this you will need to use the SQL **INSERT** command. The general format is:

INSERT INTO *table_name* (*column_name, …*) VALUES (column_value, …);

or

INSERT INTO *table_name* VALUES (*column_value, …*);

The first version lets you enter values for named columns, the second requires a data value for **all** the table columns **in the order they were defined**.

e.g. INSERT INTO category(cat_id, cat_description) VALUES (10, 'Bhangra');

or

INSERT INTO category VALUES (10, 'Bhangra');

Ensure that the data values you are trying to enter match the type and fit the size of their defined data columns and remember character strings and dates should be enclosed between a pair of **single** 'quotation marks' **not** double **"marks"**. Note the single quotation marks will appear like this ' ' when viewed in Oracle APEX.

To enter a null or empty string into a column, just use two **single quotation marks** together with no space in between them.

11. Create a new script file called **music_data** and enter all the INSERT commands as shown in the **music_data.sql** script in Appendix B.

12. Save and run the **music_data** script. Check the results carefully. If you have been successful the comment **'1 row(s) inserted'** will appear for each row inserted into a table. If you have any errors, edit the script file to make the corrections, save it, then re-run the script.

If you successfully insert rows and then run the script again you will see an Oracle error message **'ORA-00001: unique constraint (…) violated'** for each of those rows.

It is important to understand what this error means. The message is telling you that you have violated the unique constraint, i.e. that you are trying to enter a row with a primary key value which already exists. Do not worry. In this case no action is required as the rows have already been inserted.

Remember – duplicate primary key values in a table are not permitted.

You are now going to add some data of your own.

13. Edit the **music_data** script file and enter an INSERT command to add a new company of your choice to the company table and a new music category to the category table. Refer to Appendix B for the table specifications.

    Take care to ensure that you assign unique primary key values or you will get errors.
    Also remember that any **NOT NULL** columns must have a value.

    Save the script and run it. You should see the comment **"1 row(s) inserted"** for each new row that was inserted correctly.

Exercise 1

It is important to be able to check if a table exists and what its structure is.
How can you check what the table structure is?

Exercise 1 feedback

DESC company;
DESC cd;
DESC category;
DESC track;

Deleting data

If you need to delete a row or rows of data you can use the DELETE command which has the following format

```
DELETE FROM tablename
WHERE condition(s);
```

e.g.

```
DELETE FROM company
WHERE company_ID = 10;
```

In the above example only the row where the company ID is 10 will be deleted from the company table. The "WHERE" condition allows you to specify the criteria for selecting the row(s) to be deleted. Chapter 8 explains the "WHERE" conditions in more detail.

Warning: DELETE FROM *tablename* without a WHERE condition will delete all the rows.

Updating data

To change a column value in a row or rows the UPDATE command can be used. This has the following format

```
UPDATE tablename
SET condition
WHERE condition
```

e.g.

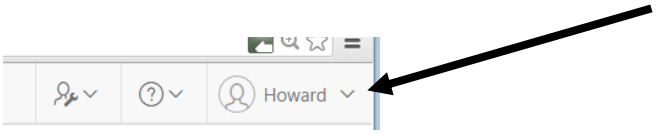
```
UPDATE company
SET company_name = 'BMB'
WHERE company_ID = 10;
```

The above example would change the company name to BMB for the row which had a company_ID value of 10.

> Take care, if you leave off the WHERE condition all rows in the table will be updated.

Signing out

To sign out of Oracle at any time, click on the account menu tab and then click on the **Sign out** button.



> You do not have to worry about saving your data before signing out as it will be maintained in the database until you run SQL commands to delete it.

> It is important to be able to understand oracle error messages and be able to identify and correct mistakes in your SQL code, so for a summary of **Oracle Errors** see www.docs.oracle.com/
>
> There are a number of support resources available for Oracle SQL and APEX including the Oracle Technology Network www.oracle.com/technetwork/index.html
>
> Other resources include www.w3schools.com/sql which offers SQL tutorials and quizzes. Remember, there may be differences in SQL commands depending on which database product you are using.

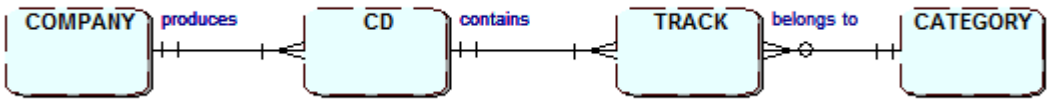
7 Using Foreign Keys

On completion of this chapter you should be able to:

- create tables with foreign keys
- drop tables referenced by foreign keys
- insert data into tables with foreign keys.

In order to implement all the relationships from the logical design you will need to implement the foreign keys that are used to link the tables. The value of the foreign key column in each row of the child table (relationship many end) must match the value of the primary key column in a row of the parent table (relationship one end). In Oracle, primary and foreign keys are referred to as **integrity constraints** and a foreign key is a **referential integrity** constraint.

The ERD for the music system is shown below:



This shows that there are relationships which require foreign keys.

There is a **1:M** relationship between **company** and **cd**; each **cd** is produced by one **company** and each **company** can produce one or more **cds** so the foreign key **cd_company_Id** is needed in the **cd** table.

The foreign key constraint for the **cd** table as it appears in the music script file is as follows:

cd_company_ID number(3) references company(company_Id));

There is also a **1:m** relationship between **cd** and **track** and there is a **1:m** relationship between **category** and **track**. So the **track** table will need two foreign keys, the cd ID no from the **cd** table and a foreign key, the cat_ID from the **category** table.

The foreign key constraints for the **track** table as they appear in the music script file are as follows:

track_cd number(3) references cd(cd_IDno),

and

track_cat_ID number(3) references category(cat_ID),

Remember the track_cd value must reference an existing primary key value, cd_IDno in the cd table, and likewise the track_cat_ID must reference an existing music category, cat_ID in the category table.

You are now going to edit the **music_data** script to enter two more rows to the track table. These rows should be for tracks 3 and 4 for cd_IDno 4, "The Rising".

Note this table uses a **compound primary key** consisting of the **cd ID number** and **track number** columns defined at the table level as:

Primary key(track_cd,track_no));

Ensure that the compound key value is unique i.e. the track_cd value 4 will be repeated but the track_no should be unique for the cd.

Rerun the script and correct any errors with the new rows.

If defining a compound foreign key use the following table level definition:
FOREIGN KEY(*column_name1, column_name2, …*)
REFERENCES table_name (*column_name1, column_name2, …*));

Cascade constraints

If you need to drop a table which has a primary key and it is referenced by a foreign key, then you need to include the CASCADE CONSTRAINTS clause in the DROP TABLE command, e.g. DROP TABLE *table_name* CASCADE CONSTRAINTS. If you omit this clause, and such a referential integrity constraint exists, then the database will return an error and will not drop the table.

In the music system this means that the company, cd and category tables will need the cascade constraints adding to their drop table commands, e.g.

```
DROP TABLE cd CASCADE CONSTRAINTS.
```

It is important to understand that if a table is referenced by a foreign key constraint it must be created before any tables that reference it are created. In the music system, the company table is created before the cd table because it is referenced by the cd table foreign key, and the cd and category tables have to be created before the track table as this table references both of them.

It is not always possible to do this where there are many table dependencies. In this situation the ALTER TABLE command can be used to add constraints after the tables have been created. The format is

```
ALTER TABLE tablename
ADD [CONSTRAINT constraint_name] constraint_type (column,…);
```

For example,

```
ALTER TABLE CD
ADD FOREIGN KEY (cd_company_ID )
REFERENCES COMPANY (company_ID);
```

The following are typical errors that you may encounter if the above guidance is not followed

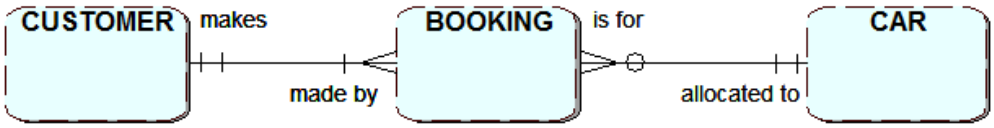
The error **"ORA-00001: unique constraint … violated"** means that a row has already been inserted into the table with a matching value in the constrained column i.e. a primary key value that already exists.

The error **"ORA-02291: integrity constraint … violated – parent key not found"** means the foreign key is not valid, either because you are trying to reference the primary key in a table that does not exist yet or the value does not match with an existing row primary key value.

Now that you have created some tables containing data you can move on to see how the data can be extracted by means of SQL queries.

Exercise 1

You are now going to create the following set of tables:



CUSTOMER (**Customer_No**, Customer_Name, Customer_Address, Customer_Tel_No)
BOOKING (**Booking_No**, Booking_Date. ***Customer_No, Car_Reg***)
CAR (**Car_Reg**, Car_Make, Car_Model)

1. Create a new script file called **Bookings** and create the three tables based on the relations above. Ensure you choose suitable column data types and sizes and do not forget to specify a primary key for each table and the two foreign keys ***Customer_No, Car_Reg*** in the booking table. Consider the order the tables will be created, as the table at the one end of a relationship must be created before the many end one – you cannot reference a table that does not exist.

2. Run the script file Bookings to create the tables.

3. Edit the script Bookings, and at the end of the script file write insert commands to insert data into all three tables. Do not forget that the data at the **one** end of a relationship must exist before the **many** end, so the booking data must be inserted last.

4. Run the Bookings script file to insert the data. Edit the file to correct any errors.

8 Selecting data from a table

On completion of this chapter you should be able to:

- select all the data from a table
- select some columns from a table
- select some rows from a table
- select some rows and some columns from a table
- order the output from a select statement.

In order to query the database you will need to select data from your tables using the SQL **SELECT** command. This is a versatile command which has a number of optional clauses [..] some of which are shown below. The | symbol is used to separate alternatives:

SELECT [DISTINCT] *column_name,….*
FROM *table_name* [,….]
[WHERE *condition(s)*]
[ORDER BY *column_name* [ASC|DESC]];

The DISTINCT clause can be used to eliminate duplicate values being displayed
The WHERE condition allows you to specify selection criteria to restrict the data retrieved
The ORDER BY clause allows you to sort the output into Ascending (default) or Descending order before it is displayed.

The SELECT command can be spread over a number of lines to aid readability, just remember to terminate the command with a semi colon;

Selecting all the data from a table.

To show all the columns for each row in the category table use the following:

SELECT cat_ID, cat_description FROM category;

Alternatively, to view **all** the columns of data for a table without listing them by name, use the * symbol as follows:

SELECT * FROM category;

Selecting some rows of data from a table

If you need to select just some of the rows of data from a table you will need to use the **WHERE** clause and a condition. A **condition** is the selection criteria that must be satisfied to select a row's details.

A **simple condition** uses a column name, a **comparison operator** and either another column name or a value. The following are the available comparison operators:

Comparison Operator	Description
=	Equal to
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to
<>	Not equal to
!=	Not equal to
BETWEEN *val1* AND *val2*	Value range
IN*(val1,val2, val3 …)*	In set of values

Here is a simple condition to show all the columns for UK companies and ordered by company name:

```
SELECT * FROM COMPANY
WHERE company_country = 'UK'
ORDER BY company_name;
```

Compound conditions can be formed by combining two or more simple conditions with the **logical** operators **AND, OR** and **NOT**

The **AND** operator is used when all the simple conditions must be true.
The **OR** operator is used when any simple condition can be true.
The **NOT** operator is used when a simple condition should be reversed i.e. not true.

Some examples of compound conditions:

```
SELECT * FROM cd
WHERE cd_payment_type = 'cash' AND cd_price > 10;

SELECT cat_description FROM category
WHERE cat_description = 'pop' OR 'rock';

SELECT company_name FROM company
WHERE NOT (company_country = 'UK');
```

Computed columns

It is also possible to include computed columns (which are not stored within the table) in SELECT commands. These contain values that are calculated using data columns and arithmetic operators. The following operators can be used:

Arithmetic Operator	Description
+	Addition
-	Subtraction
*	Multiplication
/	Division

An example which calculates the 20% tax exclusive element of a CD price:

```
SELECT cd_title, cd_price, (cd_price / 1.2) AS ExTax FROM cd;
```

Before commencing with the following practical activities you must ensure that the music system tables have sufficient data in order to run the queries. Ensure that you have inserted all the data as per the **music_data** script (see Appendix B). If necessary re-run the script.

In order to construct your SELECT commands you will need to refer to the table structures defined in the music system specification (Appendix B).

The SELECT commands in the following exercises can be run directly from within the **SQL command tool** on an individual basis or alternatively run from within a script file.

Exercise 1

Selecting some columns from a table.

Decide which table columns are needed and which order they should appear. If the rows need to be ordered, use the *order* by clause.

Using the SQL Workshop, SQL Commands tool, enter select statements for the following:

a) Select all artists

b) Select all the company names

c) Select the title and price of the cds

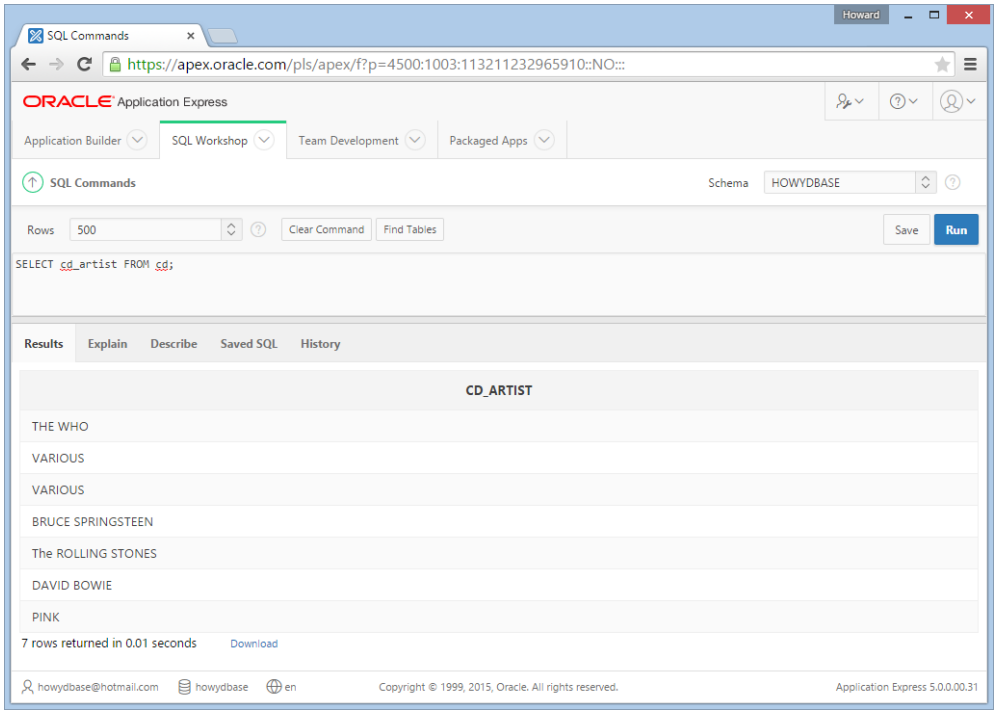
d) Select the track titles and length order by length

e) Select the category descriptions and output them in ascending order

Carefully examine the output to ensure that the command has worked correctly.

Exercise 1 feedback

a) Select all artists
SELECT cd_artist FROM cd;



SQL Command output

b) Select all the company names
SELECT company_name FROM company;

c) Select the title and price of the cds
SELECT cd_title, cd_price FROM cd;

d) Select the track titles and length order by length
SELECT track_title, track_length FROM track;

e) Select the category descriptions and output them in ascending (A-Z) order
SELECT cat_description FROM category ORDER BY cat_description;

If you do not want to see a column's output duplicated you can use the optional DISTINCT clause before the column name e.g.

```
SELECT DISTINCT cd_artist FROM cd;
```

Exercise 2

Selecting some attributes and some rows from a table

Using the SQL Workshop, SQL Commands tool, enter select statements for the following:

a) Select the names of the companies which are based in the USA.
Ensure you use upper case letters for USA or none will be selected.

b) Select the cd title and artist for the cd "THE RISING"

c) Select the track titles and track length where the length is more than 1, order by length

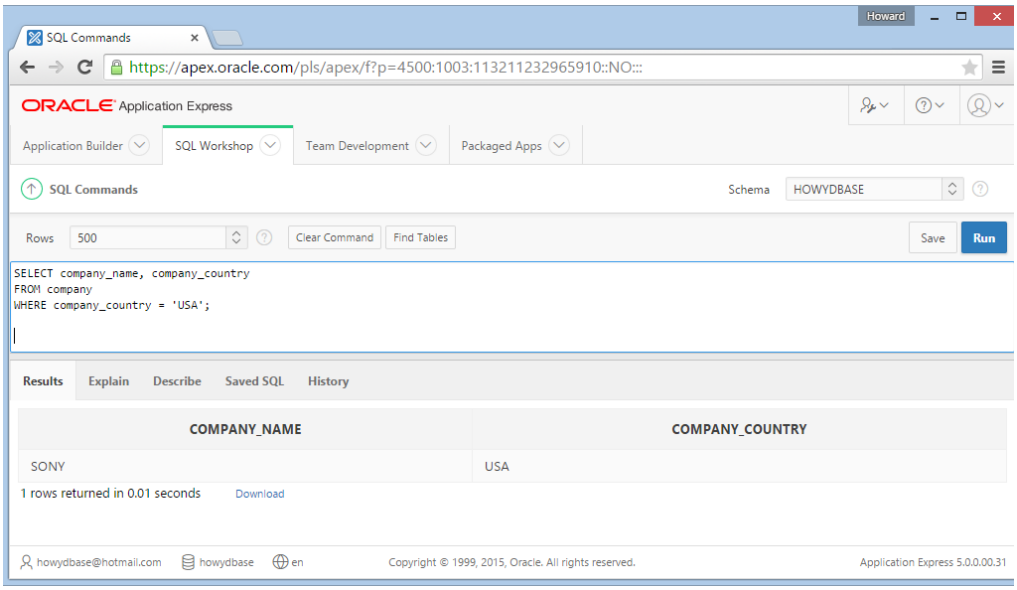
d) Select the cd title for cds produced by company number 1

e) Select the title, artist and price for cds where the price is greater than 6 but less than 13.

Exercise 2 feedback

a) Select the names of the companies which are based in the USA.

```
SELECT company_name, company_country
FROM company
WHERE company_country = 'USA';
```



b) Select the cd title and artist for the cd THE RISING

```
SELECT cd_ title, cd_artist FROM cd
WHERE cd_ title = 'THE RISING';
```

c) Select the track titles and track length where the length is more than 1, order by length

```
SELECT track_title, track_length FROM track
WHERE track_length > 1
ORDER BY track_length;
```

d) Select the cd title for cds produced by company number

```
1 SELECT cd_title FROM cd
WHERE cd_company_ID = 1;
```

e) Select the title, artist and price for cds where the price is greater than 6 but less than 13

```
SELECT cd_title, cd_artist, cd_price FROM cd
WHERE cd_price > 6 and cd_price < 13;
```

9 Selecting data from multiple tables

On completion of this chapter you should be able to:

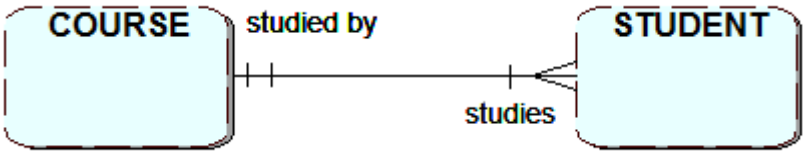
- select data from more than one table
- select data from a table using an alias.

In many cases, to satisfy a query you will need to obtain data from more than one table. If you are extracting data from two or more tables the tables will need to be **joined** to ensure that the data from one table matches correctly with the data in another table.

The join is established as a simple condition which usually includes the primary key of one table being matched with an appropriate foreign key in another table. This type of join is sometimes referred to as an **equijoin** or **inner join**.

Finally, include any other conditions which are needed to restrict the rows to those required and include ORDER BY if needed.

As an example, suppose you want to obtain the course details for a given student.




```
CREATE TABLE course
(course_code          CHAR(5) PRIMARY KEY,
course_name           VARCHAR2(20)NOT NULL,
course_start_date     DATE);

CREATE TABLE student
(student_ID           VARCHAR2(6) PRIMARY KEY,
student_name          VARCHAR2(50) NOT NULL,
student_address       VARCHAR2(50),
course_code           CHAR(5) REFERENCES course(course_code));
```

To do this you will need to construct a query that joins the course and student tables. This will be done by matching the primary key column course_code of table course with the foreign key column course_code from the student table as follows:

```
SELECT course_code, course_name, student_name,
FROM course, student
WHERE course.course_code = student.course_code
AND student.course_code = 'COMP';
```

The above query selects the course code and name from the course table and the student name from the student table only for students studying the course with code 'COMP'.

Note that the two columns being joined have the same name, so in order to avoid ambiguity each column must be referred to by its full name i.e. *table_name.column_name*. This approach would also be used if a column to be selected had the same name as a column in a different table that was used in the query.

If you **leave out** the correct join when trying to match two tables you will obtain output which is based on every row in the first table being combined with every row in the second table. This is called a **Cartesian product.**

E.g. Compare the correct output from:

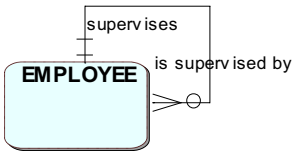
```
SELECT cd_title, track_title
FROM cd,track
WHERE cd_IDno = track_cd;
```

With the incorrect output from:

```
SELECT cd_title, track_title
FROM cd,track;
```

Self joins

It is sometimes the case that you will need to join a table to itself in order to extract the required information. This is referred to as a **recursive** or **self join**. In order to do this the table needs to be given an **alias** so that it can be referenced simultaneously from two different row positions. In this type of join the data from the primary key appears as data in another column, the foreign key. As an example, suppose you have a situation where employees have a supervisor and some employees act as a supervisor. In this case the employee details could be represented by the ERD and the single table and data below.




```
CREATE TABLE employee (
        employee_no  VARCHAR2(6) PRIMARY KEY,
        name         VARCHAR2(20),
        supervisor   VARCHAR2(6) REFERENCES employee(employee_no));
```

employee_no,	name,	*supervisor*
701	B Boss	
702	U Ling	701
703	A Lee	701

If you want to extract the name of the supervisor (B Boss) for employee no. 702 (U Ling) then you would need to access the employee table from the perspective of the employee being supervised (alias **e**) using the foreign key column *supervisor* and matching this with the supervisor's perspective (alias **s**) primary key column *employee_no*. The Select command needed is as follows:

```
SELECT s.name
FROM employee e, employee s
WHERE e.supervisor = s.employee_no
AND e.employee_no = '702';
```

Outer joins

In some cases you may wish to extract data using two tables which do not have shared column values and so cannot be joined as in previous examples. Suppose you wanted to see tracks and their music categories, including those that have not been used in the track table. If you joined the track table with the category table you would not obtain all the music categories; however, if you use an outer join this would allow you to see the missing music categories.

The track table is regarded as a deficient table as it does not contain all the category table category IDs. To specify an outer join, you use the + symbol in the selection condition on the deficient table side. This can be on either side of the condition but not both.

The following format is used:

```
SELECT table1name.columnname, [tablen2ame.columnname]
FROM table1name, table2name
WHERE table1name.columnname(+) = table2name.columnname;
```

For the example described above this would be the following:

```
SELECT track_title, track_cat_ID, cat_ID, cat_description
FROM track, category
WHERE track.track_cat_ID(+)  = category.cat_ID;
```

Note that there is no track with a cat_ID of 4 but the description "Classical" is still displayed.

Using built-in functions

In some situations there is a need to use Oracle built-in functions which can be incorporated into SELECT commands in order to extract the required data. For example, it is quite common to retrieve rows that have a column which matches a particular date or date range. In order to compare a date with a column date it is necessary to convert the required date character string into a valid format before the comparison can take place. To do this you can use the **TO_DATE** function as follows:

```
SELECT cd_title, cd_date_purchased
FROM cd
WHERE cd_date_purchased = TO_DATE('28-JAN-2002', 'DD-MON-YYYY');
```

Several different date formats can be used, so take care to ensure that the date format used in the TO_DATE function matches that used by the stored data.

Similarly, there are some character functions that can be used to compare a CHAR or VARCHAR2 column value with a character string. Care must be taken to ensure that the values are in the same case or they will not be treated as a match.

Oracle provides the built-in functions **UPPER, LOWER** and **INITCAP** which can be used for the purpose of converting a value into either **UPPER**, **lower** or upper case just for the first character of each separate group of characters within the string. These functions can be used to convert the character strings used in the WHERE clause.

If using the UPPER OR LOWER function on a column, the conversion is only temporary in order to make the comparison, it does not change the data value stored in the column.

Here are some examples that would allow you to match the data held in the cd_title column:

```
SELECT *
FROM cd
WHERE cd_title = UPPER('Best of Bowie');
```

or

```
SELECT *
FROM cd
WHERE LOWER(cd_title) = 'best of bowie' ;
```

or

```
SELECT *
FROM cd
WHERE INITCAP(cd_title) = 'Best Of Bowie' ;
```

For more information on built-in functions visit
https://docs.oracle.com/javadb/10.6.2.1/ref/rrefsqlj29026.html

You can now try some queries which use two tables to extract data from the music system (Appendix B).

Remember you must include a correct join or your output will be a Cartesian product.

Exercise 1

Using the SQL Workshop, SQL Commands tool enter select statements for the following:

a) Select the cd title and the name of its company for all the cds.

b) Select the cd titles and their track titles.

c) Select the track titles and their music category descriptions.

d) Select the track title for music category "Dance" tracks.

e) Select all cd titles which have track lengths > 3 and have a price > 10.

f) Select the company name and cd for each cd that was purchased after 1st September 2005.

Exercise 1 feedback

a) Select the cd title and the name of its company for all the cds.

```
SELECT cd_title, company_name
FROM cd,company
WHERE cd_company_ID = company_ID;
```

b) Select the cd titles and their track titles.

```
SELECT cd_title, track_title
FROM cd,track
WHERE cd_IDno = track_cd;
```

c) Select the track titles and their music category descriptions.

```
SELECT track_title, cat_description
FROM track, category
WHERE track_cat_ID = cat_ID;
```

d) Select the track title for music category Dance tracks.

```
SELECT track_title
FROM track, category
WHERE track_cat_ID = cat_ID AND cat_description = 'Dance';
```

e) Select all cd titles which have track lengths > 3 and have a price greater than 10.

```
SELECT DISTINCT cd_title
FROM cd, track
WHERE cd_IDno = track_cd AND track_length > 3 AND cd_price > 10;
```

f) Select the company name and cd for each cd that was purchased after 1st September 2005.

```
SELECT company_name, cd_title, cd_date_purchased
FROM company, cd
WHERE company_ID = cd_company_ID
        AND cd_date_purchased > TO_DATE('01-SEP-2005');
```

The following queries will require joining three or four tables.

Exercise 2

Using the SQL Workshop, SQL Commands tool enter select statements for the following:

a) Select the company_name, cd title, track title and music category description.

b) Select the names of the companies which produced cds with "Dance" tracks.

c) Select all cd titles for cds which include "Pop" or "Dance" category tracks.

Exercise 2 feedback

a) Select the company_name, cd title, track title and music category description.

```
SELECT company_name, cd_title, track_title, cat_description
FROM company, cd, track, category
WHERE company_ID = cd_company_ID AND cd_IDno = track_cd
        AND track_cat_ID = cat_ID;
```

b) Select the names of the companies which produced cds with Dance tracks.

```
SELECT DISTINCT company_name
FROM company, cd, track, category
WHERE company_ID = cd_company_ID AND cd_IDno = track_cd
        AND track_cat_ID = cat_ID AND cat_description = 'Dance';
```

c) Select all cd titles for cds which include Pop or Dance category tracks.

```
SELECT DISTINCT cd_title
FROM cd, track, category
WHERE cd_IDno = track_cd AND track_cat_ID = cat_ID
        AND cat_description IN ('Pop','Dance');
```

10 Subqueries and group functions

On completion of this chapter you should be able to:

- use a sub query
- apply group functions.

When trying to extract data which involves using a number of tables it is sometimes appropriate to use a subquery rather than a select command with lots of joins. Subqueries are also referred to as nested queries. A subquery contains at least one nested query. The innermost query is executed first and the value returned is then used at the next level up.

There are two types of sub query

- **Single Row** – returns only one row of data
- **Multiple Row** – returns more than one row of data.

Single-row subquery

The general format of a subquery is:

SELECT *column_name,….*
FROM *table_name*
WHERE *column_name*
relational operator (SELECT *column_name,…*
FROM *table_name*
WHERE **condition)**;

The subquery must be enclosed within ( )
The ORDER BY clause cannot be used.
The relational operators =, >, <, >=, <= ,<>,!= can be used.

An error will occur if more than one row is returned from a subquery, and if no rows are returned the value will be a NULL.

A subquery may be used on a single table. For example suppose you wanted to select the track name and its music category where the track music category description was Pop. The following can be used:

```
SELECT track_title, track_cat_ID
FROM track
WHERE track_cat_ID =
    (SELECT cat_ID
    FROM category
    WHERE cat_description = 'Pop');
```

Multiple-row subquery

A multiple row subquery returns more than one row therefore you need to use the following operators, not the ones used for the single row queries:

Operator	Use
IN	Matches any value in a list
ALL	Compares the given value with every returned subquery value
ANY or SOME	Compares the given value with each returned subquery value

Using the Order System specification in Appendix C let us look at an example. Suppose you wanted to select the product descriptions for items on order Z01. The item table product numbers for the chosen order would be returned and then used to match with the product table product numbers in order to select the product descriptions.

```
SELECT Prod_desc
FROM product
WHERE product.Prod_no
        IN (SELECT item.Prod_no
        FROM item
        WHERE Order_no = 'Z01');
```

Exercise 1

1. Set up the Order System tables and insert the sample data as shown in Appendix C.
2. Answer the following using **sub queries:**

    a) Name the products that "Asda" has ordered.
    b) Name the customers who have placed an order in 2012.
    c) Name the customers who have ordered "Chocolate".

Exercise 1 feedback
2. a)

```
SELECT Prod_desc
FROM product
WHERE Prod_no
        IN(SELECT item.Prod_no
        FROM item
        WHERE item.Order_no
                IN(SELECT Order_no
                FROM porder
                WHERE porder.Cust_no
                        IN(SELECT Cust_no
                        FROM customer
                        WHERE Cust_name = 'Asda')));
```

b)

```
SELECT Cust_name
FROM customer
WHERE Cust_no
        IN(SELECT porder.Cust_no
        FROM porder
        WHERE Order_date > TO_DATE('31-DEC-2011','DD-MON-YYYY')
                AND Order_date < TO_DATE('01-JAN-2013','DD-MON-YYYY'));
```

c)

```
SELECT Cust_name
FROM customer
WHERE customer.Cust_no
        IN(SELECT porder.Cust_no
        FROM porder
        WHERE porder.Order_no
                IN(SELECT item.Order_no
                FROM item
                WHERE item.Prod_no
                        IN(SELECT Prod_no
                        FROM product
                        WHERE Prod_desc = 'Chocolate')));
```

Group functions

A group function operates on a group of rows and returns a single result. Suppose you wanted to find out which cd cost the most, or what the total value of all your cds is, these queries can be answered using the group functions.

Function	Description
SUM(*column_name*)	Calculates the total value of a column, null values are ignored.
AVG(*column_name*)	Finds the average of all the values in a column, null values are ignored.
MAX(*column_name* | *expression*)	Finds the maximum value in a column or expression, null values are ignored.
MIN(*column_name* | *expression*)	Finds the minimum value in a column or expression, null values are ignored.
COUNT(* | *column_name* | *expression*)	* Counts the number of rows, including nulls If a column or expression is used counts non null values.

Note: The | separates the optional values

Here is an example showing the average, maximum, minimum and total for the cd prices:

```
SELECT AVG(cd_price), MAX(cd_price), MIN(cd_price), SUM(cd_price)
FROM cd;
```

Sometimes it is necessary to apply group functions to separate groups of rows rather than all the rows. The GROUP BY clause can be used to group the rows. The format is as follows:

```
SELECT column_name, group function (column_name)
FROM table_name
[WHERE condition(s)]
[GROUP BY column_name | expression]
[ORDER BY column_name | expression [ASC | DESC]];
```

When using the GROUP BY clause the columns in SELECT must also appear in the GROUP BY. The WHERE clause cannot be used to restrict groups, but it can be used to restrict the data before the grouping. By default GROUP BY when used with a column will output results in ascending order.

Here is an example of a GROUP BY which will tell you how many tracks you have for each music category:

```
SELECT track_cat_ID, COUNT(*) "no of tracks"
FROM track
GROUP BY track_cat_ID;
```

In order to restrict groups, the HAVING clause can be used, so suppose you only wanted to see the number of tracks for music categories that had more than 16 tracks, the following could be used:

```
SELECT track_cat_ID, COUNT(*) "no of tracks"
FROM track
GROUP BY track_cat_ID
HAVING COUNT(*) > 16;
```

Exercise 2

Write SELECT statements to answer the following:

a) Count the total number of tracks.

b) Show the maximum and minimum track lengths.

c) Display the average cd price by company for any companies with an average price < 10.

Exercise 2 feedback

a) Count the total number of tracks.

```
SELECT COUNT(*)
FROM track;
```

b) Show the maximum and minimum track lengths.

```
SELECT MAX(track_length), MIN(track_length)
FROM track;
```

c) Display the average cd price by company for any companies with an average price < 10.

```
SELECT AVG(cd_price)
FROM cd
GROUP BY cd_company_ID
HAVING AVG(cd_price) < 10;
```

You have now been introduced to the main ORACLE SQL commands and some of the built-in functions, so you should now be able to search for and try some of the other SQL commands and functions.

There are many SQL resources available on the web which can assist you to develop your skills including:-

An Oracle database SQL language reference can be found here:
https://docs.oracle.com/cd/E11882_01/server.112/e41084/toc.htm

For more help with Oracle APEX try here:
https://community.oracle.com/community/database/developer-tools/application_express

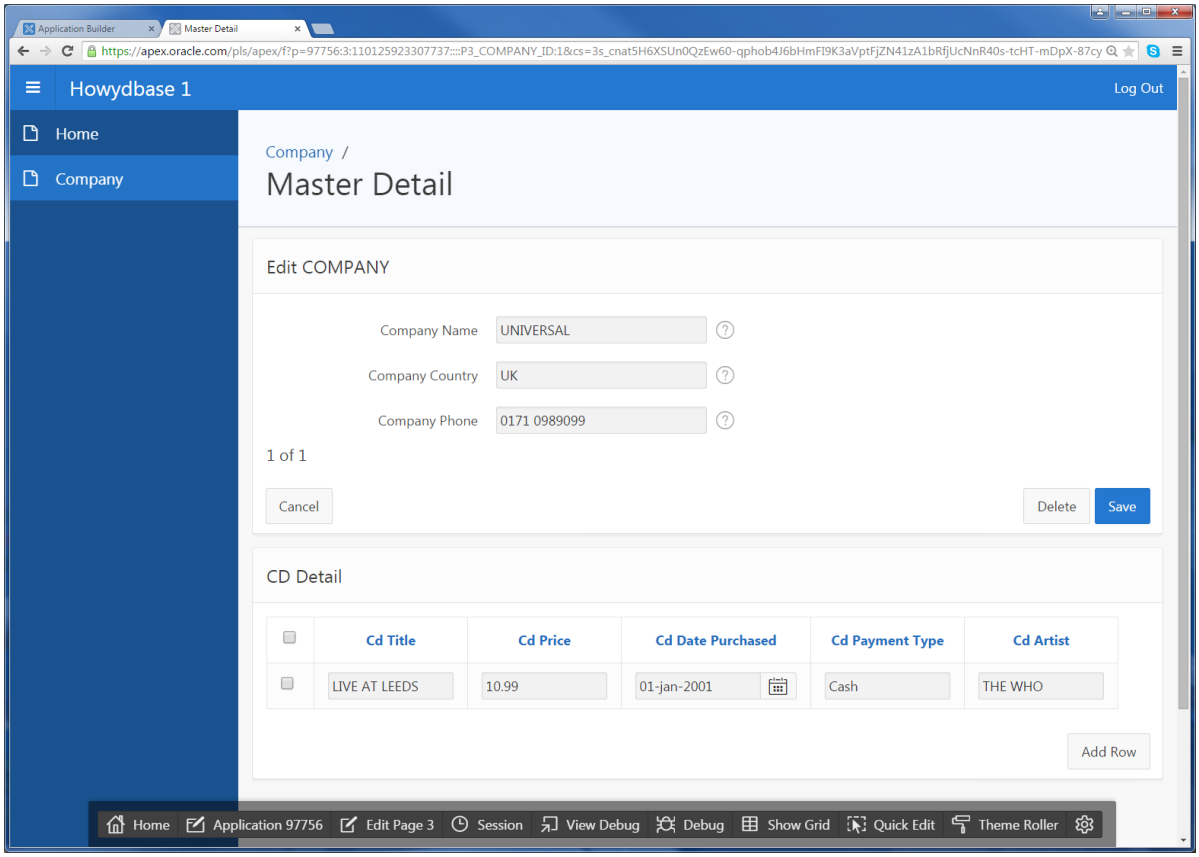
11 Creating pages & reports

On completion of this chapter you should be able to:

- create a simple master detail form page
- create a simple report.

Introduction

Many users of databases do not directly access them using SQL but instead use applications consisting of easy to use web Pages (sometimes called Forms) and reports, which allow controlled access to the data. The following is a basic introduction to using APEX for building a simple page and report. You will be using the **company** and **cd** tables to build a master (company) detail (cd) page which will represent the 1:M relationship. This can also be referred to as a parent (one) child (many) relationship between company and cd. The page will show all the cds produced for a selected company.



A Master Detail Page

Follow the following steps carefully or you may not be able to build the pages correctly.

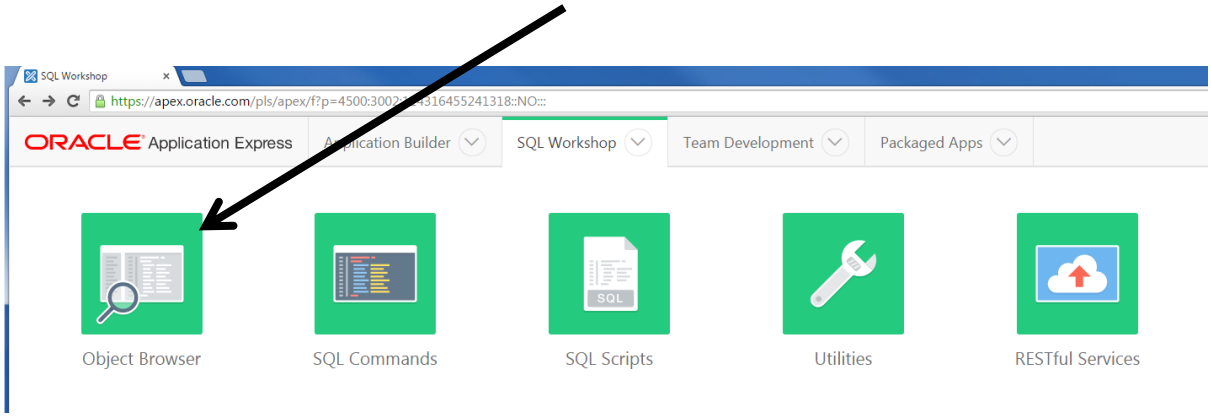
Before you can build the master detail page you need to ensure that you have set up **sequences** for the tables being used. Database sequences are database objects that act like counters and are used to automatically generate integer values. The sequence value is used to automatically populate the primary key column when a row is added to the table.

In order to create the company and cd tables with sequences for their primary keys you are going to drop the existing company and cd tables and create new versions, this time using the SQL Workshop object browser.

1. In the SQL Commands tool drop the company and cd tables using cascade constraints:
   DROP TABLE company CASCADE CONSTRAINTS;
   DROP TABLE cd CASCADE CONSTRAINTS;

You will need to set up the company table before the cd table as there is a foreign key cd_company_ID in the cd table that references the company table company_ID.

2. In the SQL Workshop click on the Object Browser icon.



SQL Workshop

3. Choose the **Tables** objects.

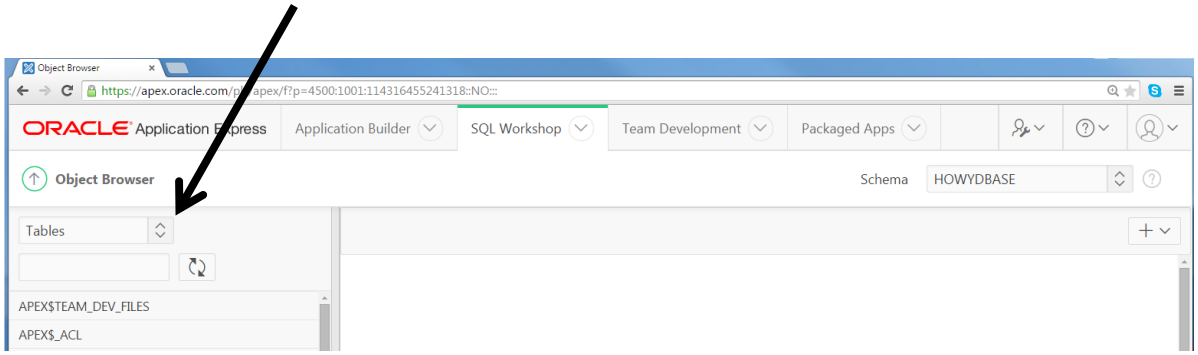
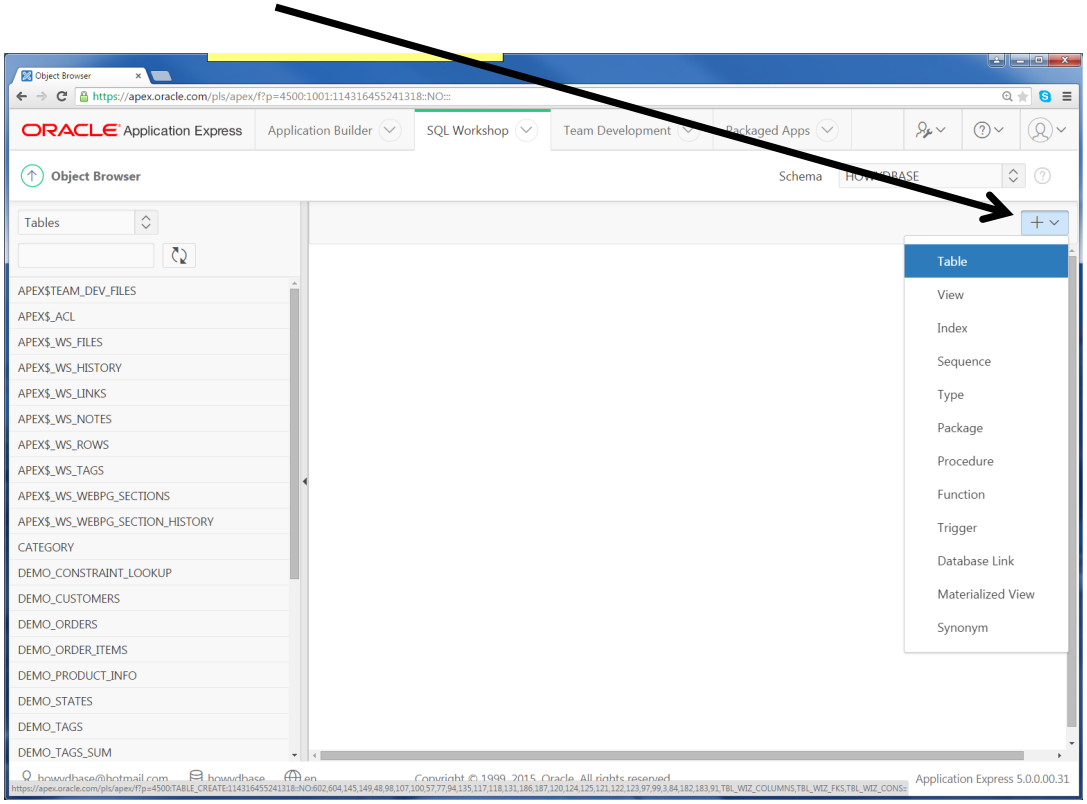


Table Objects

4. Click on the + to add a Table.



Add table

5. Enter the table name **company** and the column names, data types and sizes (use scale) as per the specification in Appendix B. To select Not Null for a column just tick the Not Null check box. Then click the **Next** > button.

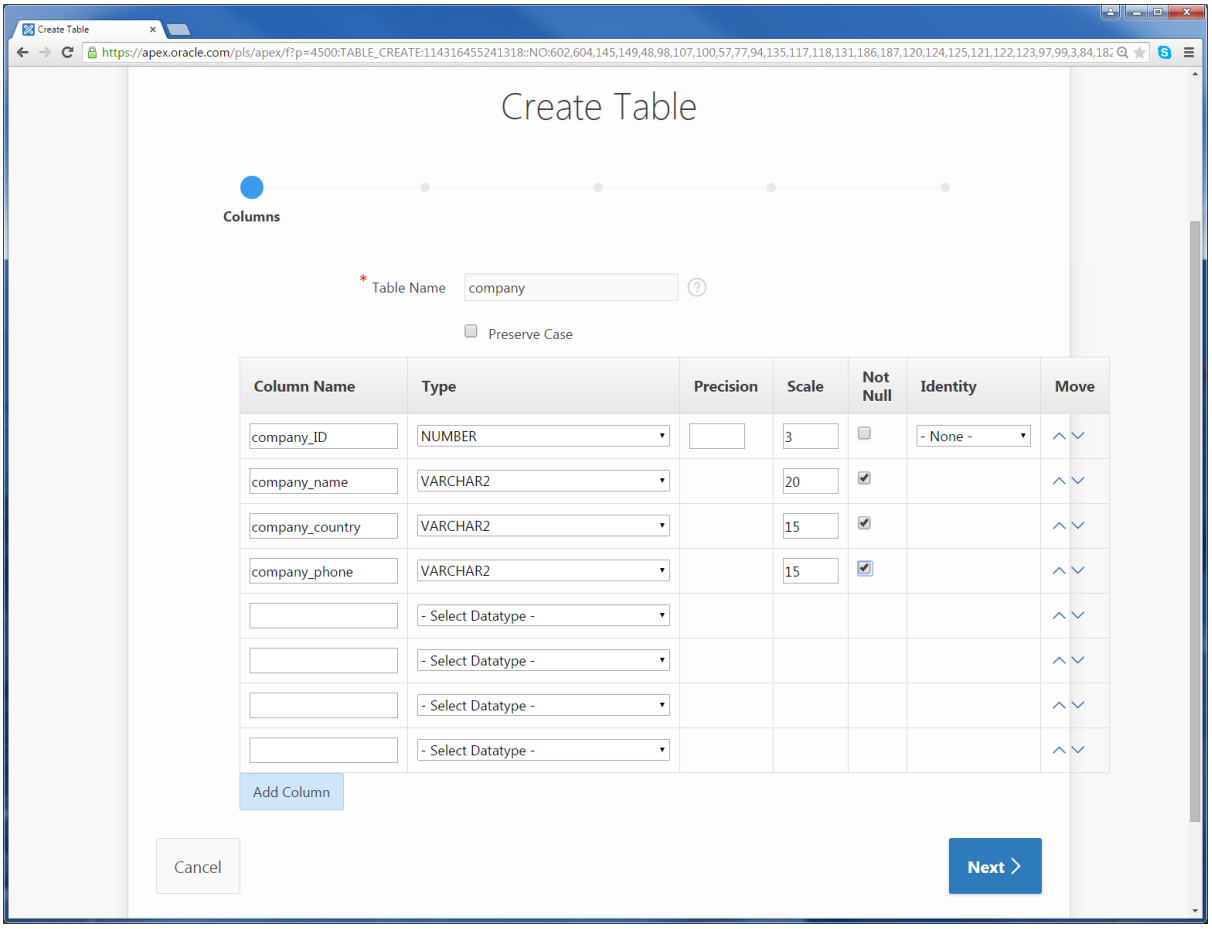
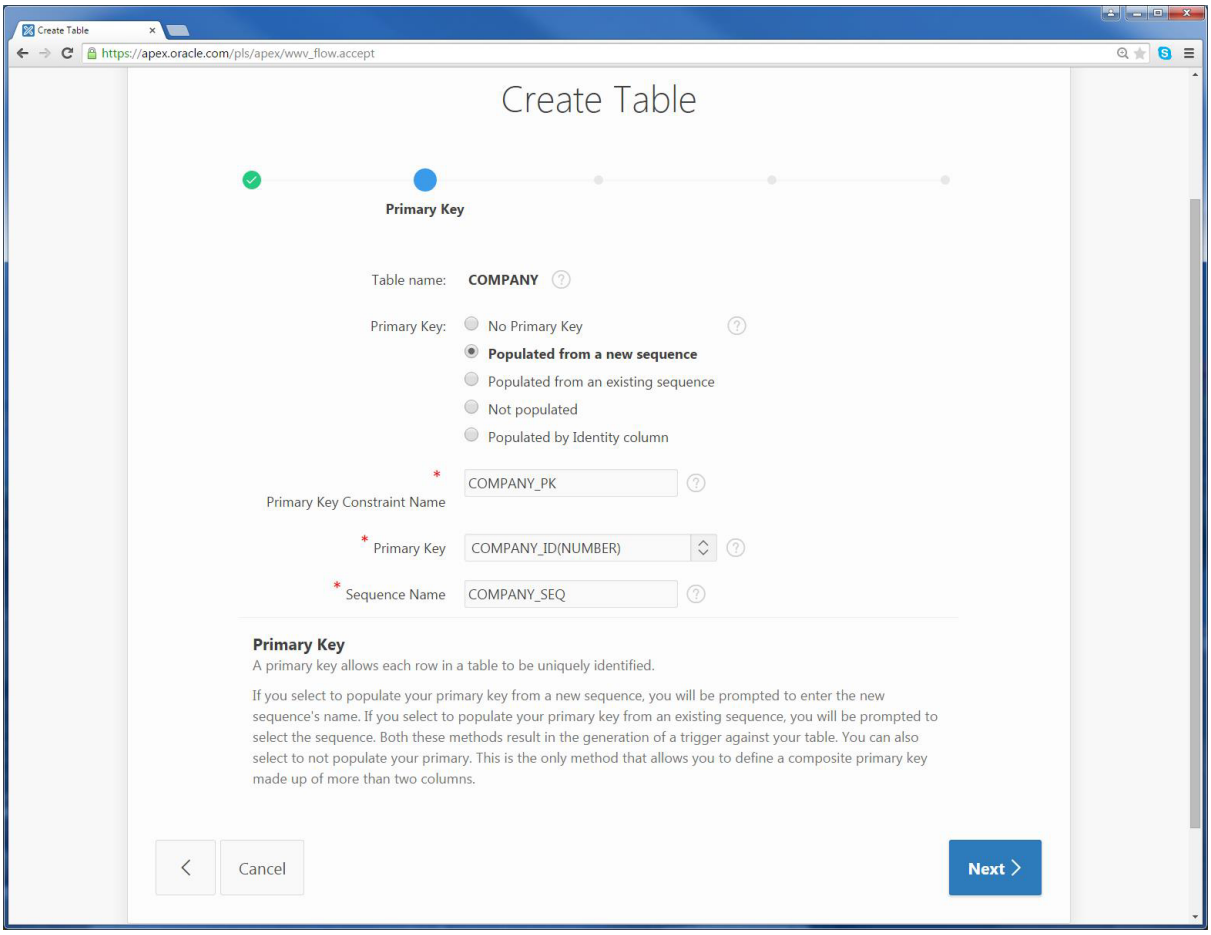


Table Columns Page

6. Click on the **Populated from a new sequence** button then click inside the Primary Key box and select the COMPANY_ID(NUMBER) column from the list of columns. The sequence will automatically be given a constraint name and a sequence name. Click **Next** >.



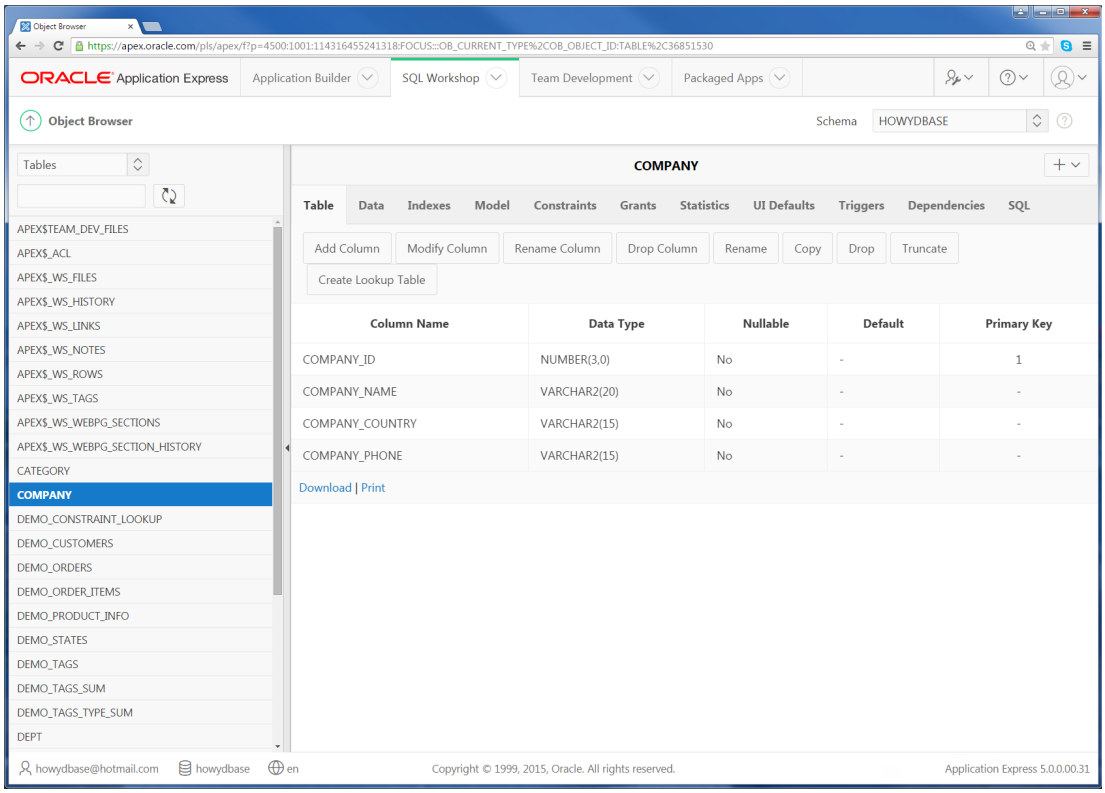
Primary Key

7. When the **Foreign Key** page appears click **Next** > as there are no foreign keys in this table.

8. When the **Constraints** page appears click **Next**> as you are not adding any further constraints to the table.

9. When the **Create table** page appears click on the **Create Table** button.

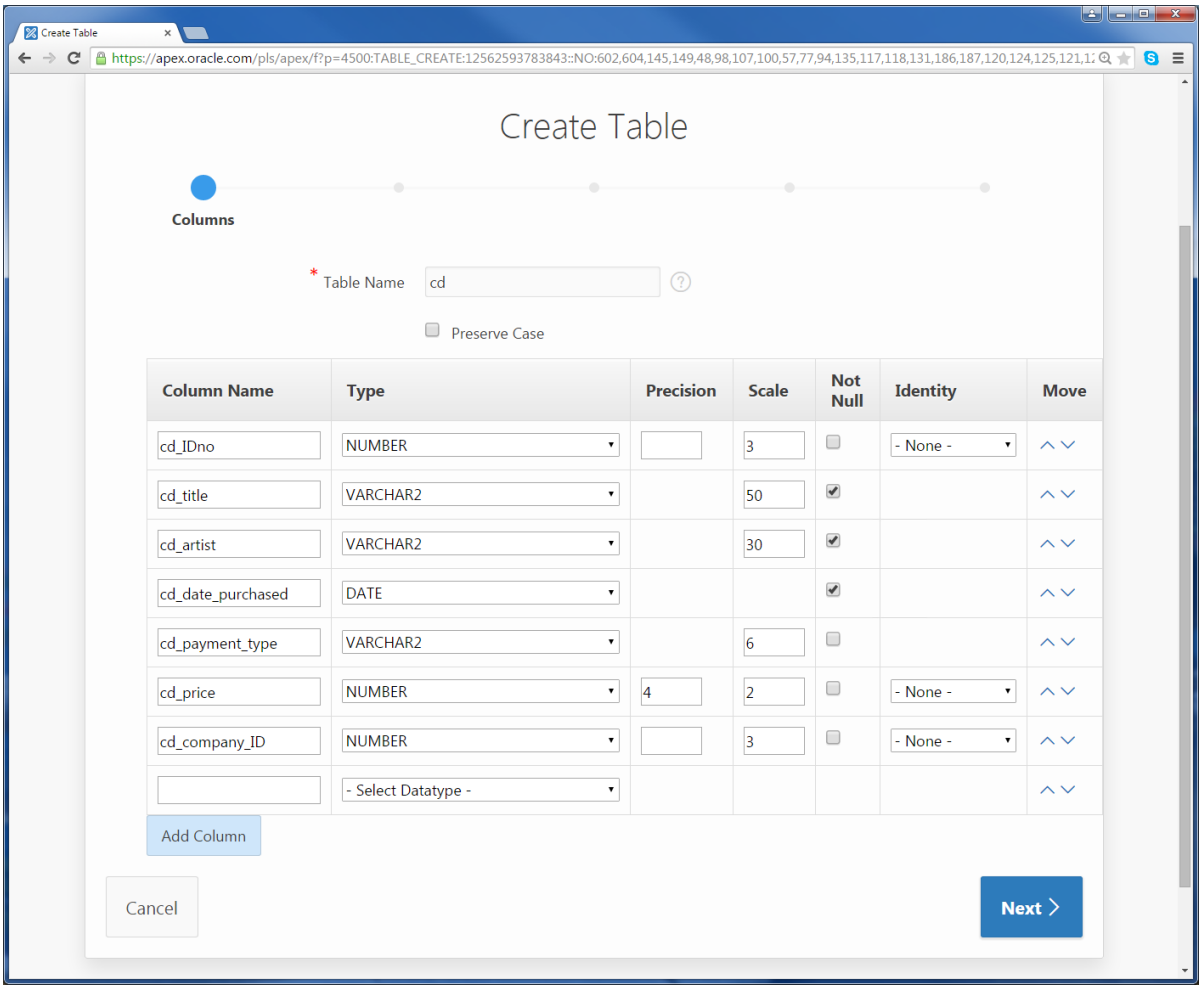
10. The confirmation page will appear in the Object Browser when the table has been created.



Company Table

You are now going to create the cd table and its sequence.

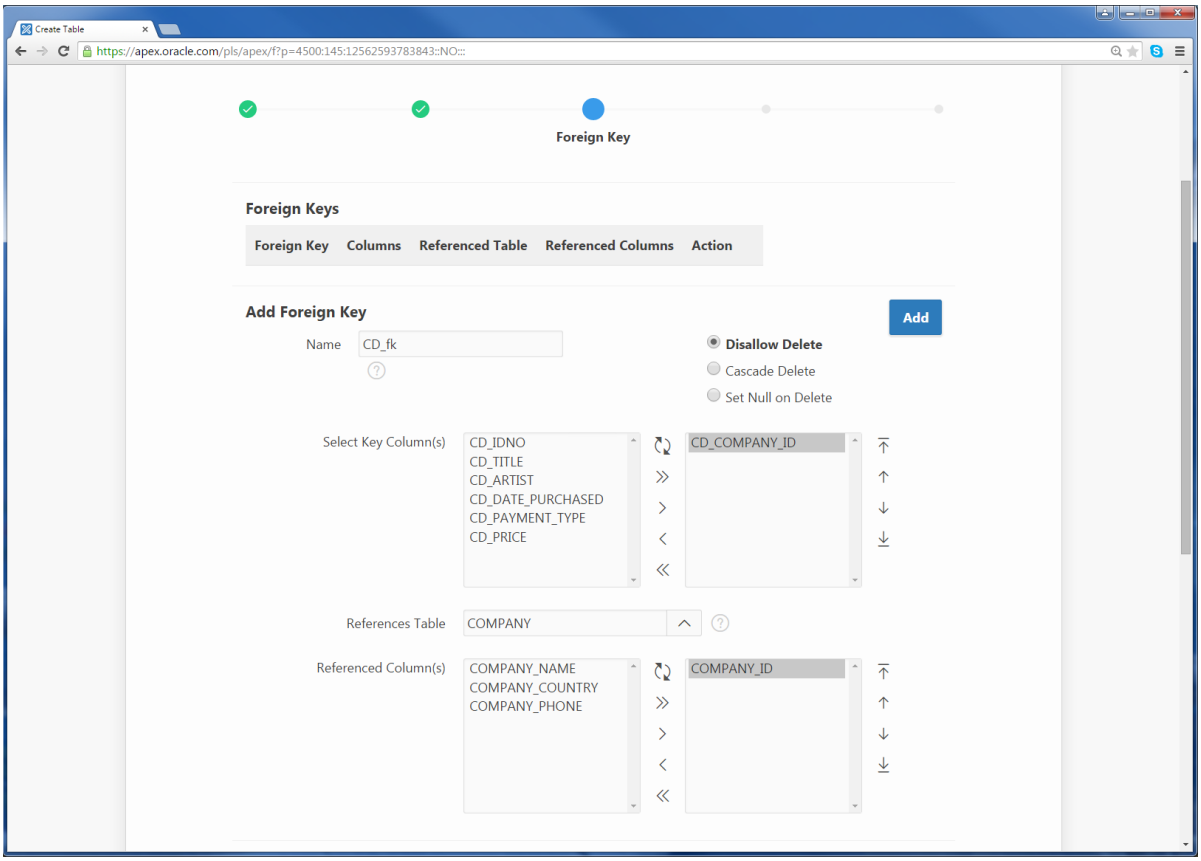
11. Repeat steps 1 to 3 above then enter the cd table name and columns as below. Remember to set the Not Null where appropriate. Take care when entering the **cd_price** column details, enter 4 in the Precision box and 2 in the Scale box. **Click Next >**.



Create cd table

12. On the **Primary Key** page select **Populated from a new sequence** then select **CD_IDNO(NUMBER)** for the **Primary Key**, click **Next >**.

13. On the **Foreign Key** page select the foreign key column **CD_COMPANY_ID** and click >, then choose the table being referenced by clicking the ^ in the references table box, then select the **COMPANY** table from the list. The company table columns will appear as referenced columns; select the primary key column **COMPANY_ID** and click >. Click the **Add** button to save the key. On the next page, as you are not adding any further foreign keys to this table, click the **Next >** button at the foot of the page.



Adding a Foreign Key

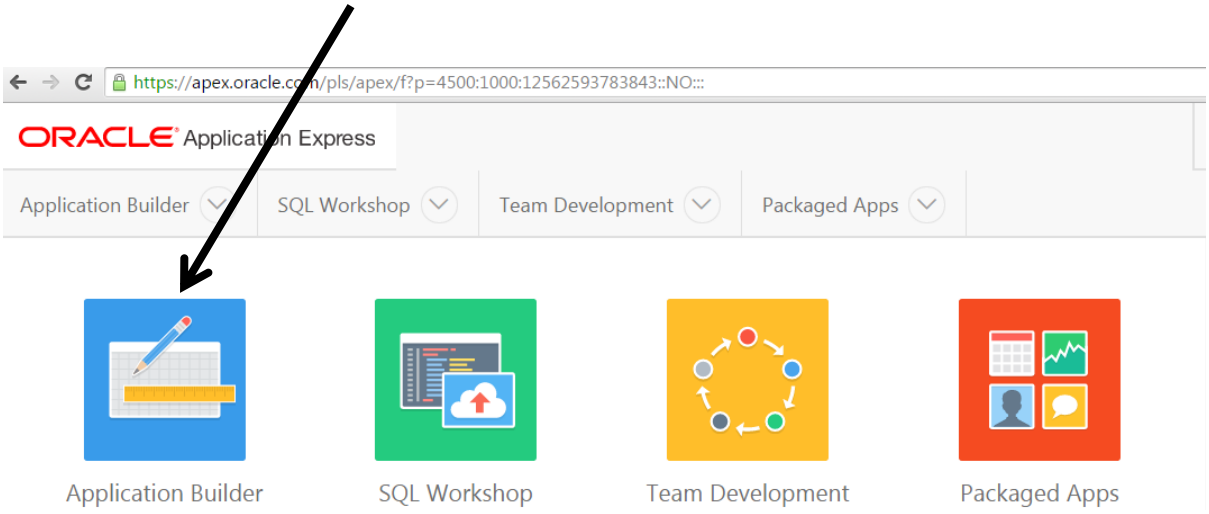
14. On the **Constraints** page click **Next >**.

15. On the **Create Table** page click the **Create Table** button. The CD confirmation page should then appear.

If you DROP either of these tables the sequences will not be automatically dropped. To remove the sequences you can use the SQL **DROP SEQUENCE** command e.g. DROP SEQUENCE CD_SQ; or alternatively use the Object Browser and select Sequences then choose the sequence and click on the Drop button.

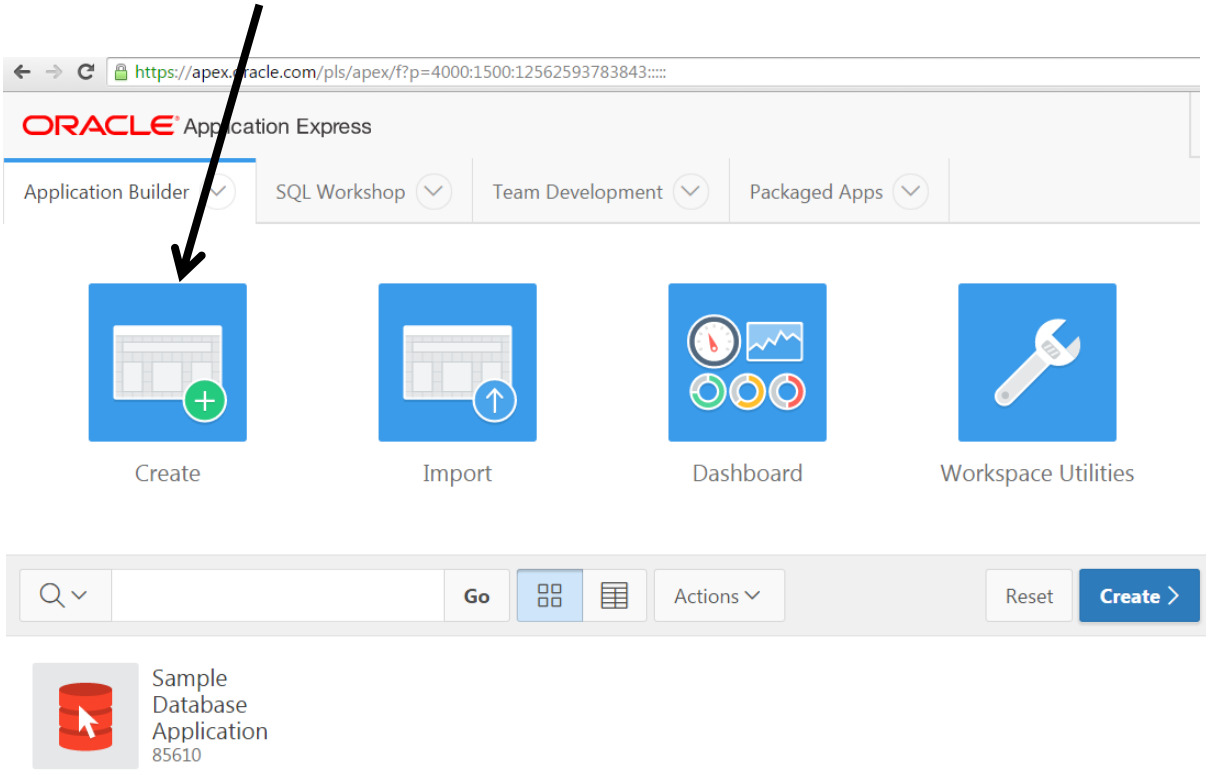
Oracle APEX applications consist of a number of pages of varying types including forms, reports and charts. Now that you have re-created the company and cd tables with sequences you can create a desktop application which will include a **Master Detail** page that will use them.

16. Click on the **Application Builder** tab on the APEX Home page.



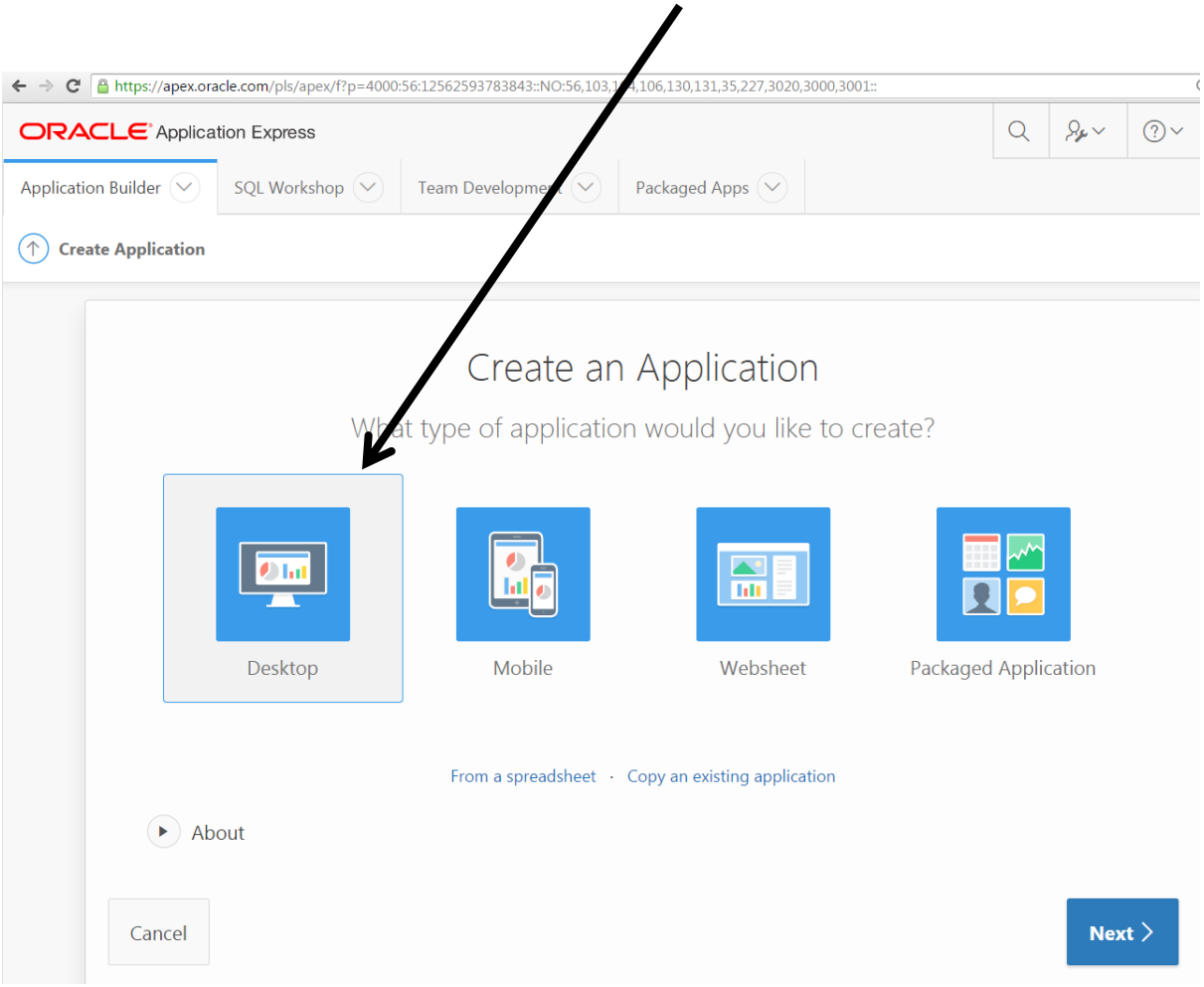
Application Builder

17. Click on the **Create** icon.



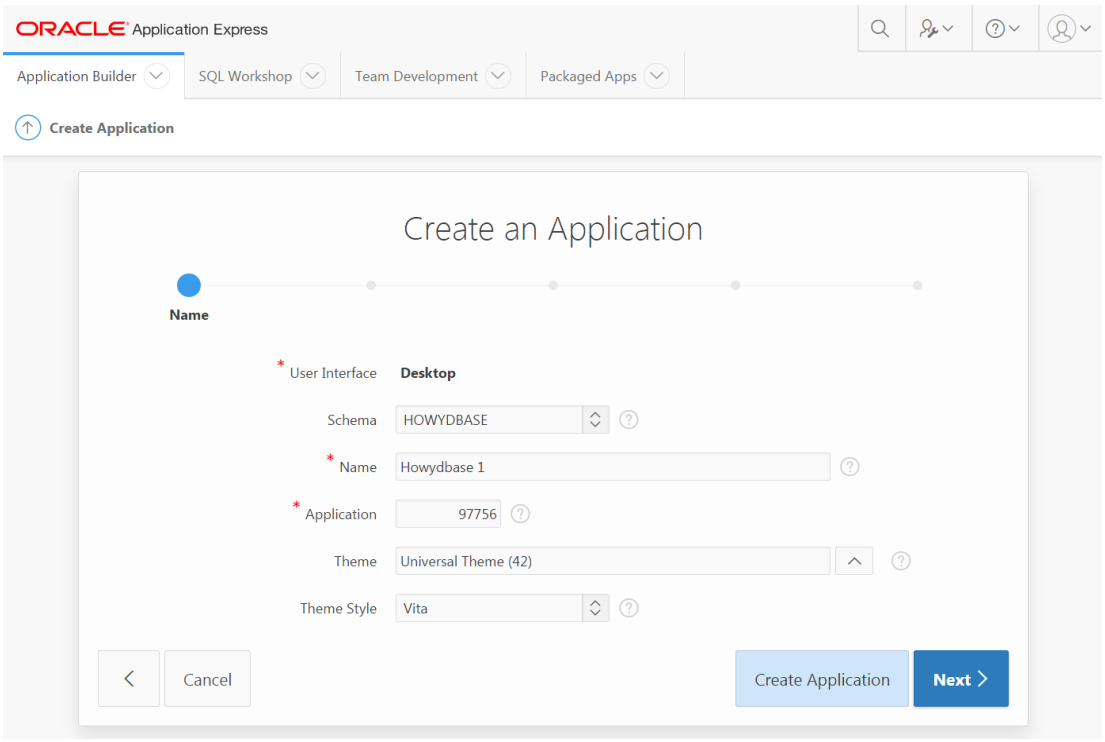
Create Application

18. On the **Create an Application** page click on the **Desktop** icon.



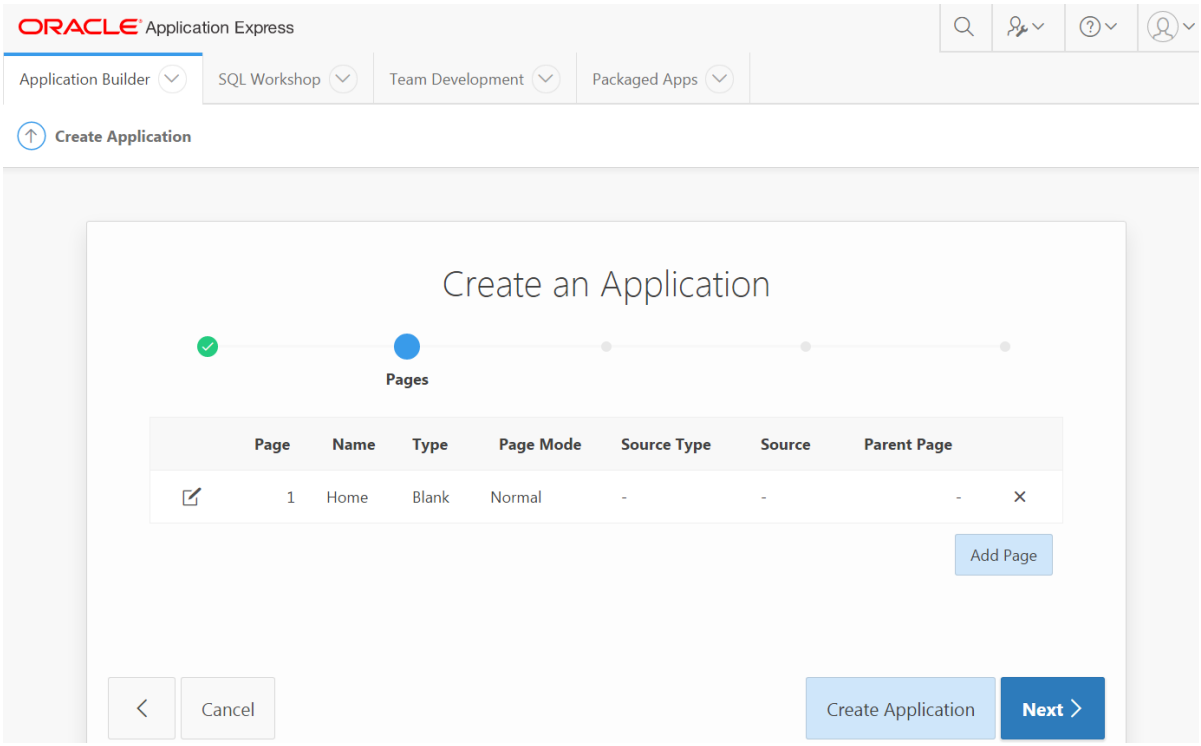
Desktop application

19. The application will be given a Name and number along with a style theme that will be used for the pages, these are alterable but for now just click on **Next >**.



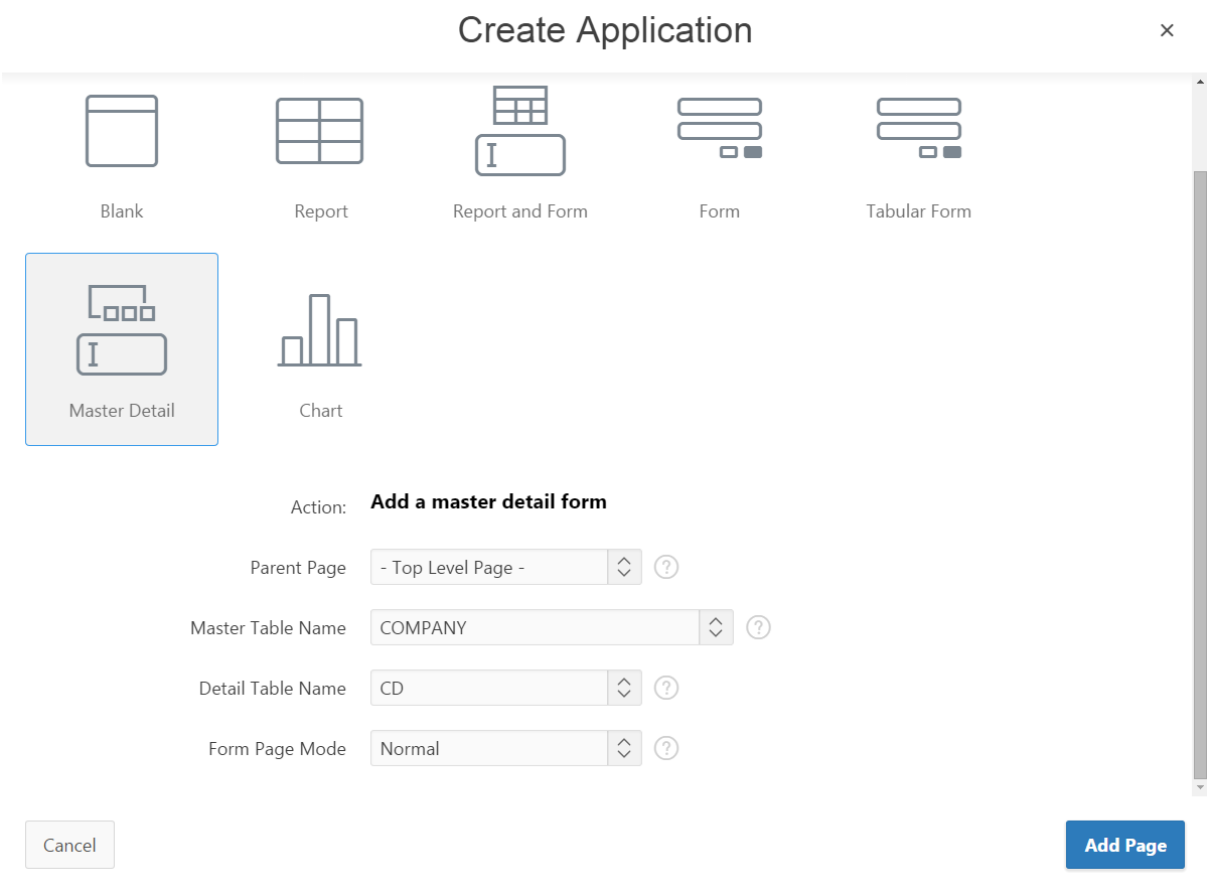
Application details

20. A home page will automatically be created for your application. Click on the **Add Page** button.



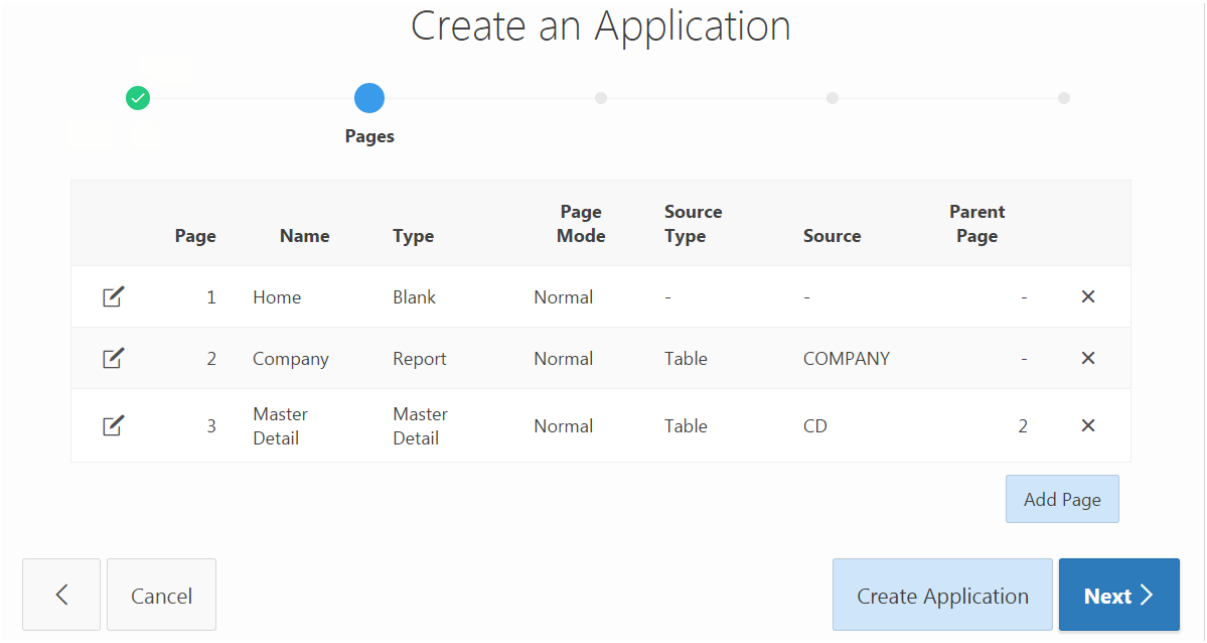
Home Page

21. On the **Page type** selection page click on the **Master Detail** icon then click in the Master Table Name box and select COMPANY. Then click in the Detail Table Name box and select CD, then click on the **Add Page** button.



Add Master Detail form

22. The create application page will now show that three separate pages have been created – Home page, Company page and a Master Detail page. Click **Next >**.

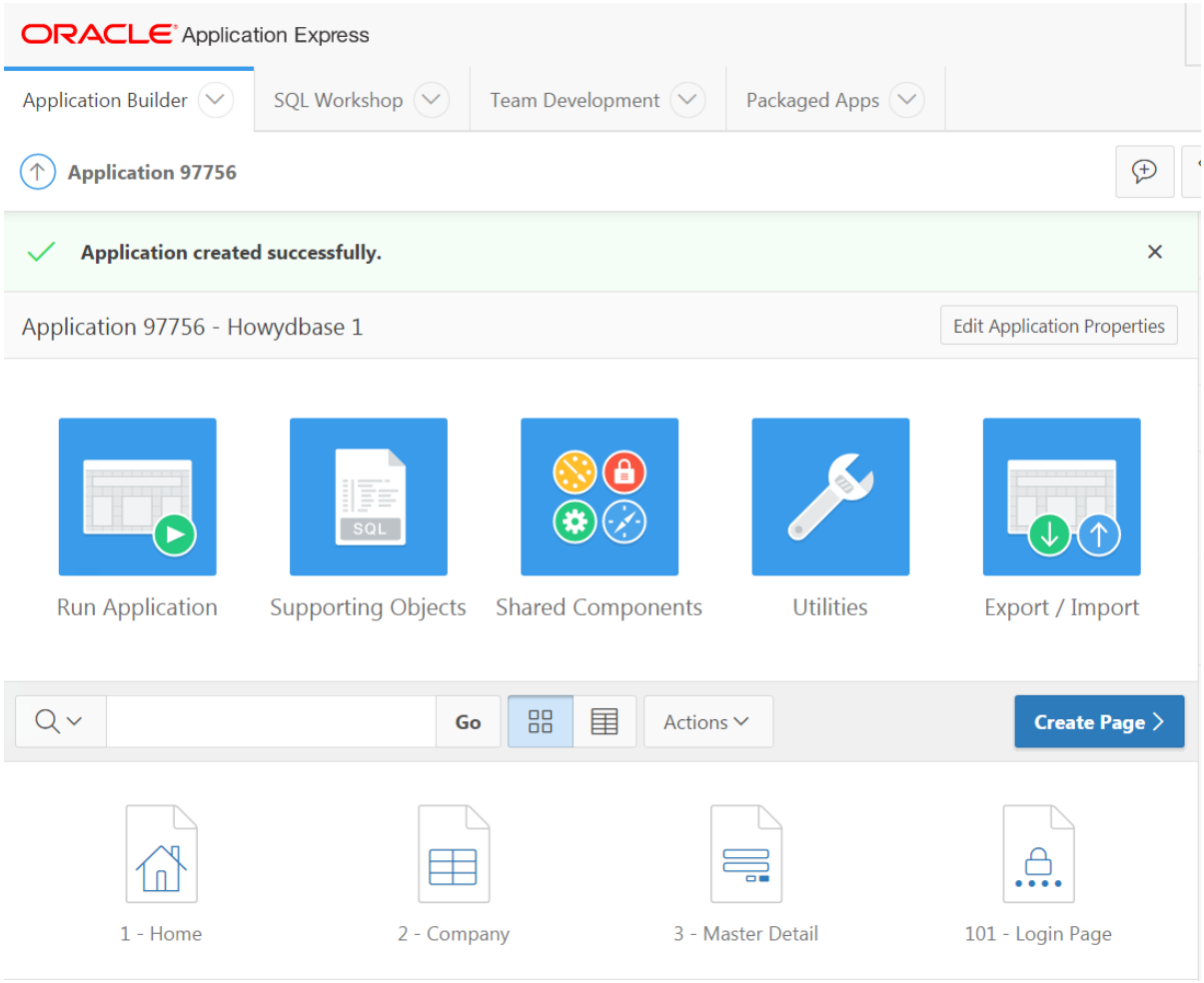


Application Pages List

23. On the **Shared components** page click **Next >**.

24. On the **Attributes** page just click **Next >**.

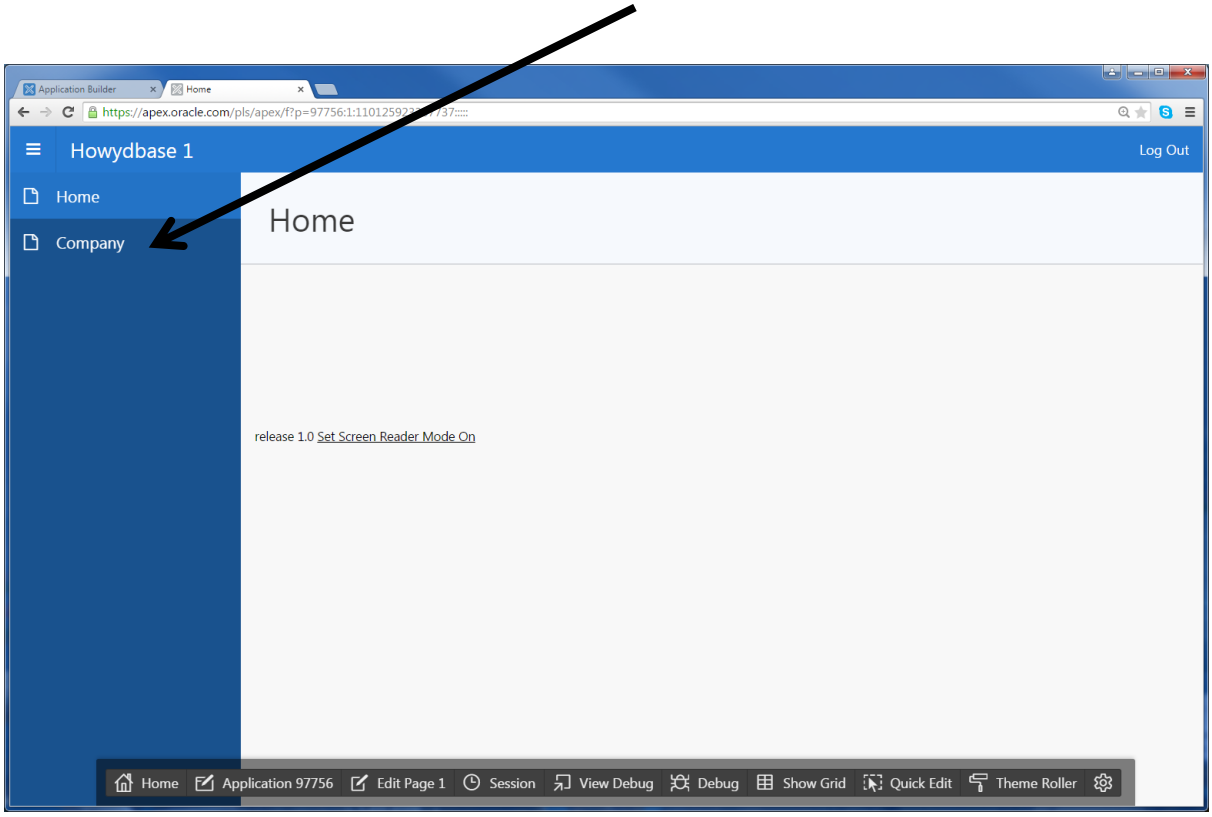
25. On the **Confirm** page click **Create Application**. After a short while the created application confirmation screen should appear, showing all four of the generated application pages.



Application confirmation

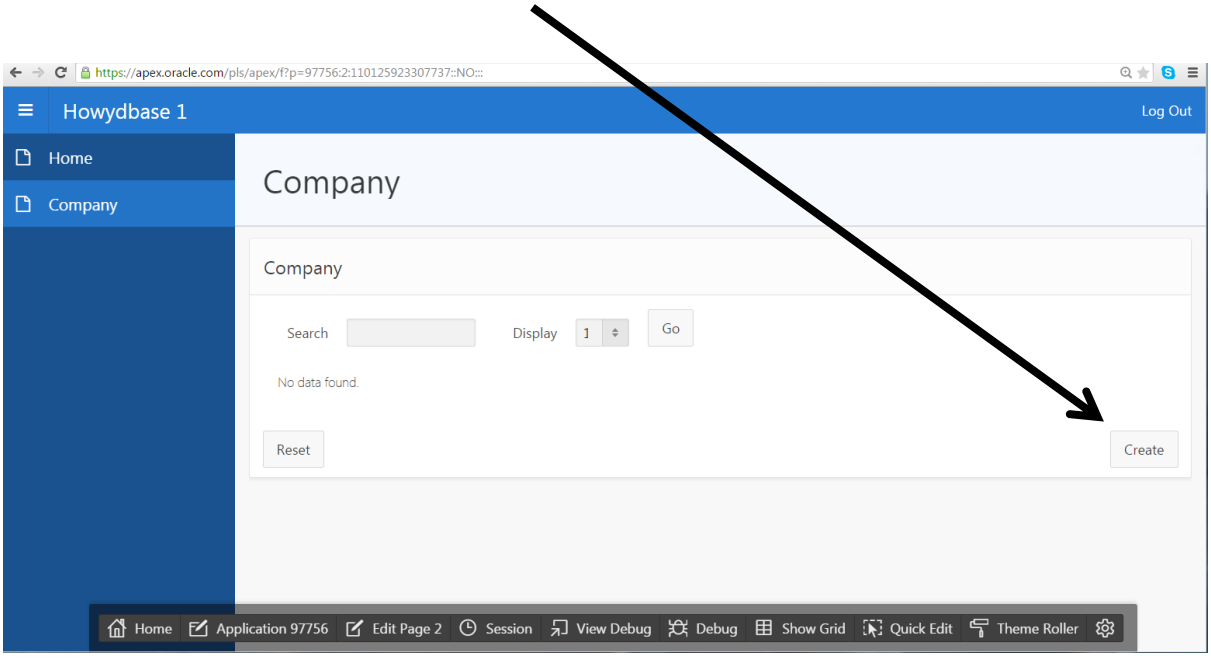
26. You are now ready to run the application. Click on the **Run Application** Icon. When the **Log In** page appears enter your username and Password if it is not already displayed and click on the **Log In** button.

27. When the **Home** page appears click on the **Company** page link.



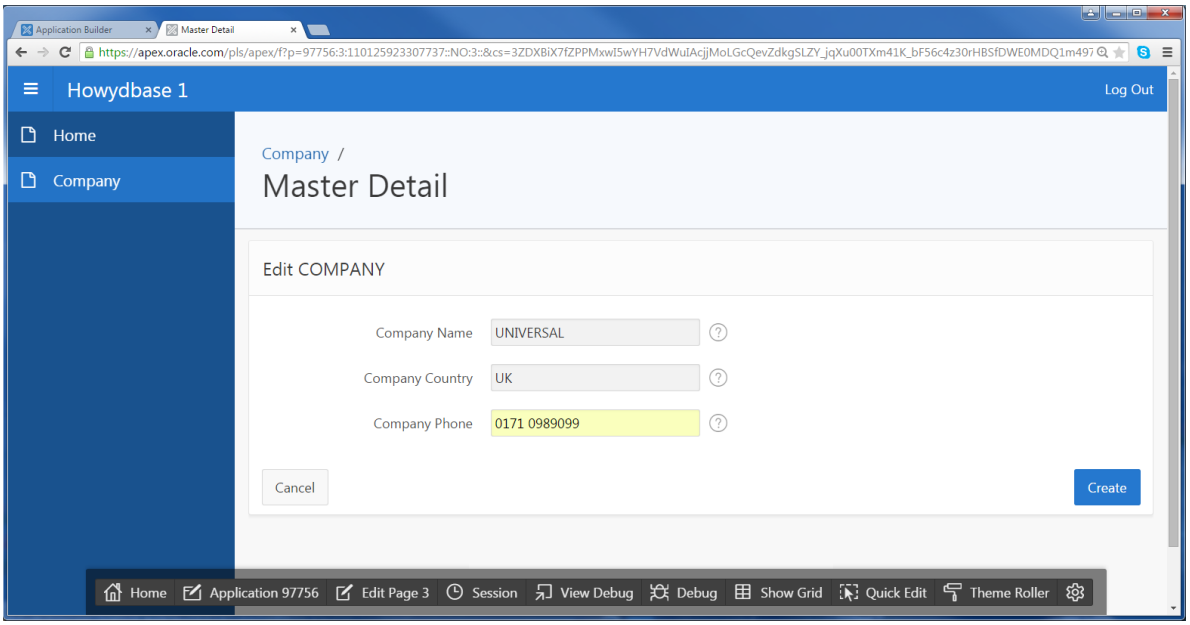
Home Page

28. As there are no rows of data in the Company table you can now add some through the company page form. Click on the **Create** button.



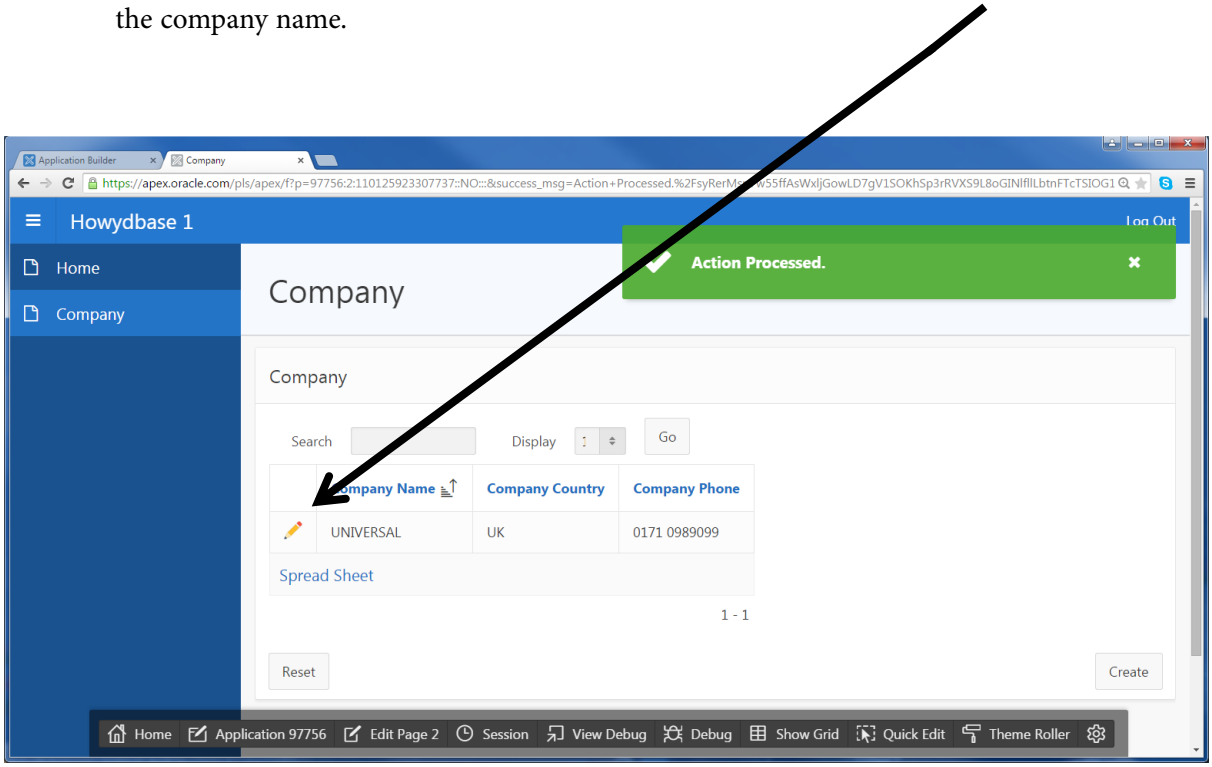
Company Page

29. The Edit Company page should appear at this point. Enter a company name, country and phone number. Note that you have not been asked to enter a company Id as this will be populated automatically by the table sequence when the row is created. Click on the **Create** button.



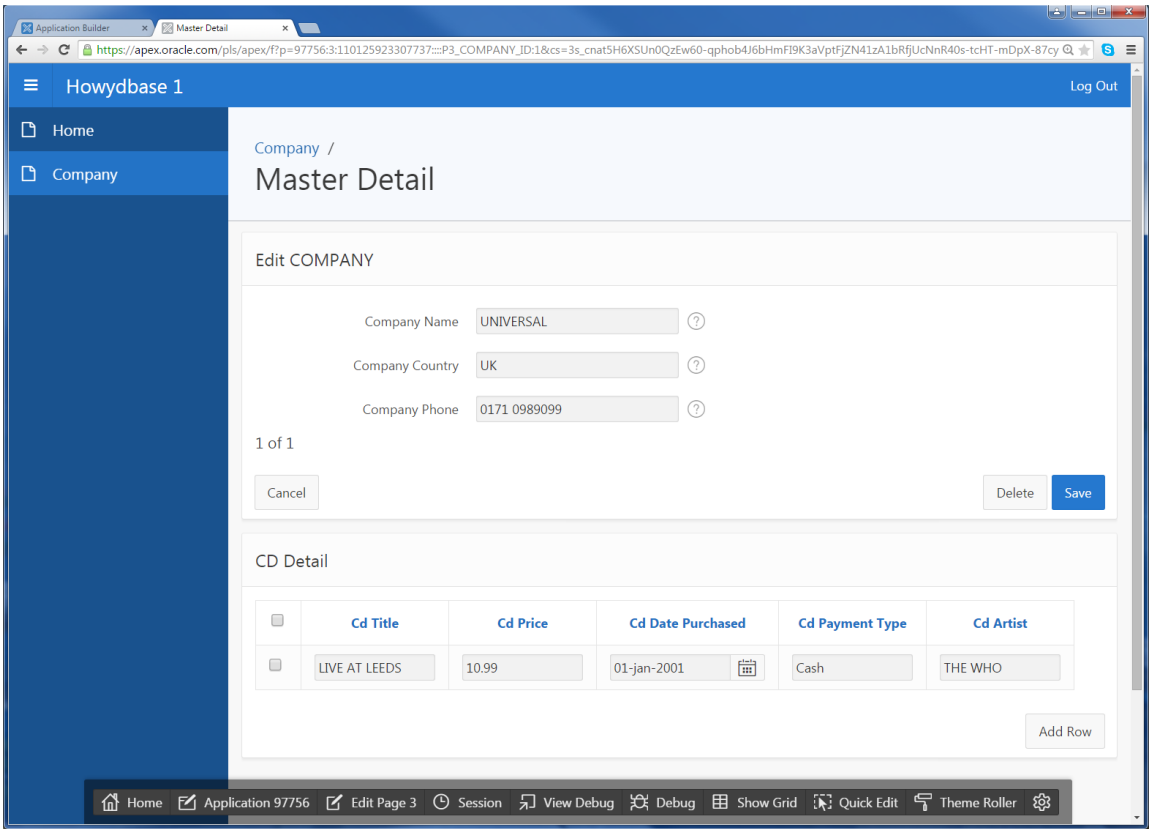
Company Page data entry

30. You are now going to create a cd row for this company. Click on the **Pencil** icon to the left of the company name.



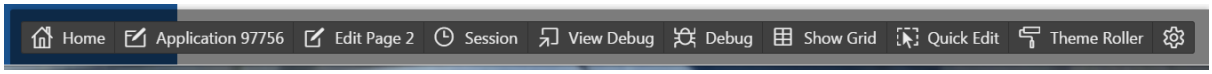
Company Record

The CD detail block will now be visible and you can add a row of data to the CD table by clicking on the **Add Row** button. Enter a CD's details then click the **Save** button.



Detail Row

31. To return to the APEX home page or the Application home page, click on the appropriate icon in the developer tool bar which is displayed at the bottom of the Apex page.

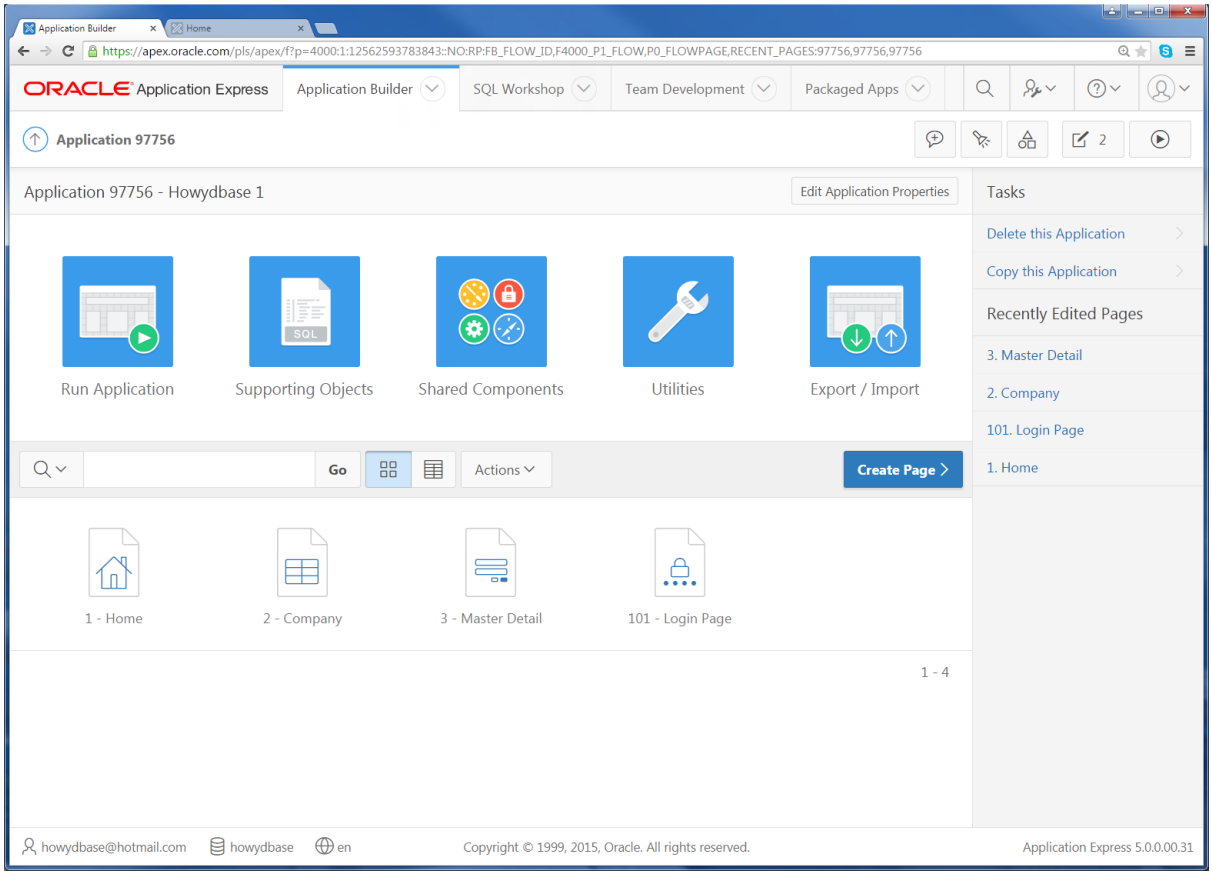


Developer Toolbar

APEX reports

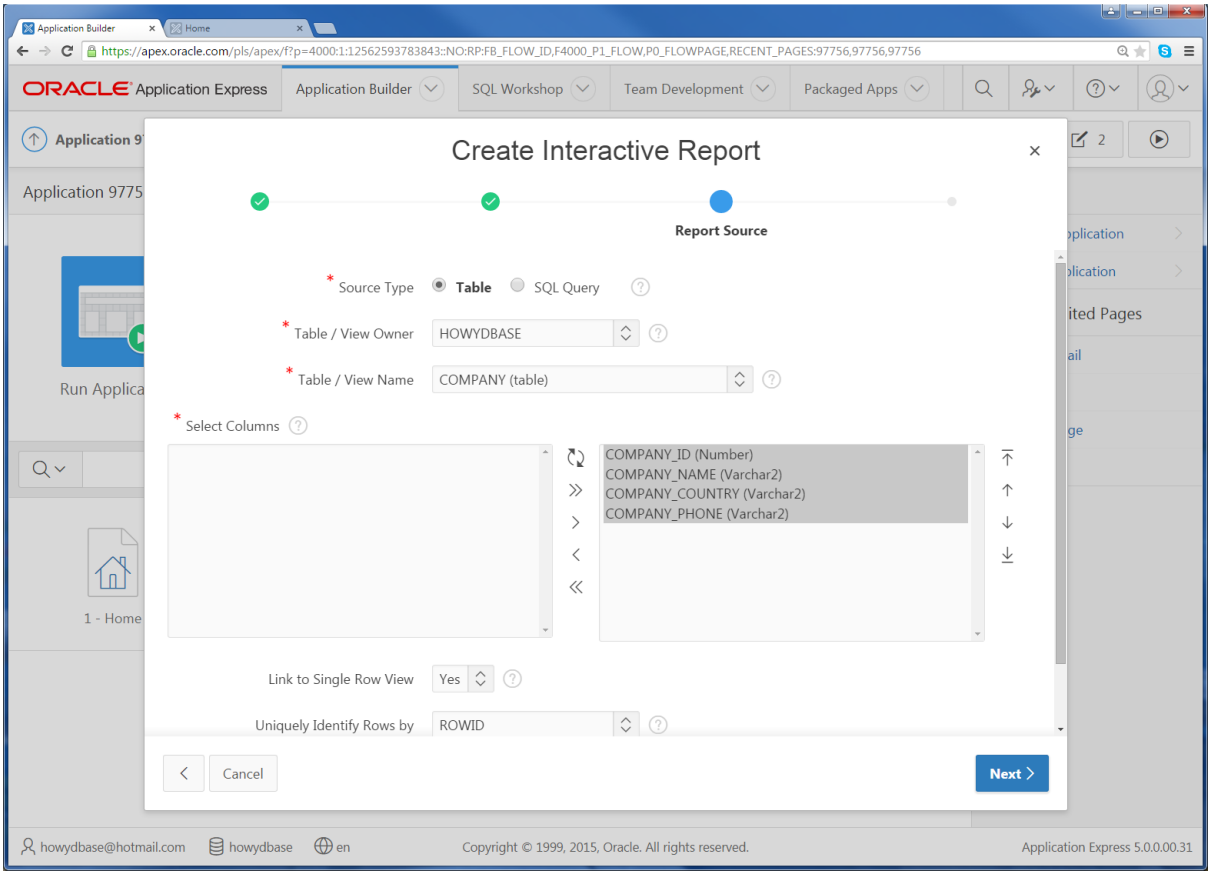
Most applications require reports which display table data in an easy to understand format.
APEX Application Builder lets you produce various types of reports. The following instructions will show you how to produce a simple interactive report.

32. Return to your application home page and click on the **Create Page** > button.



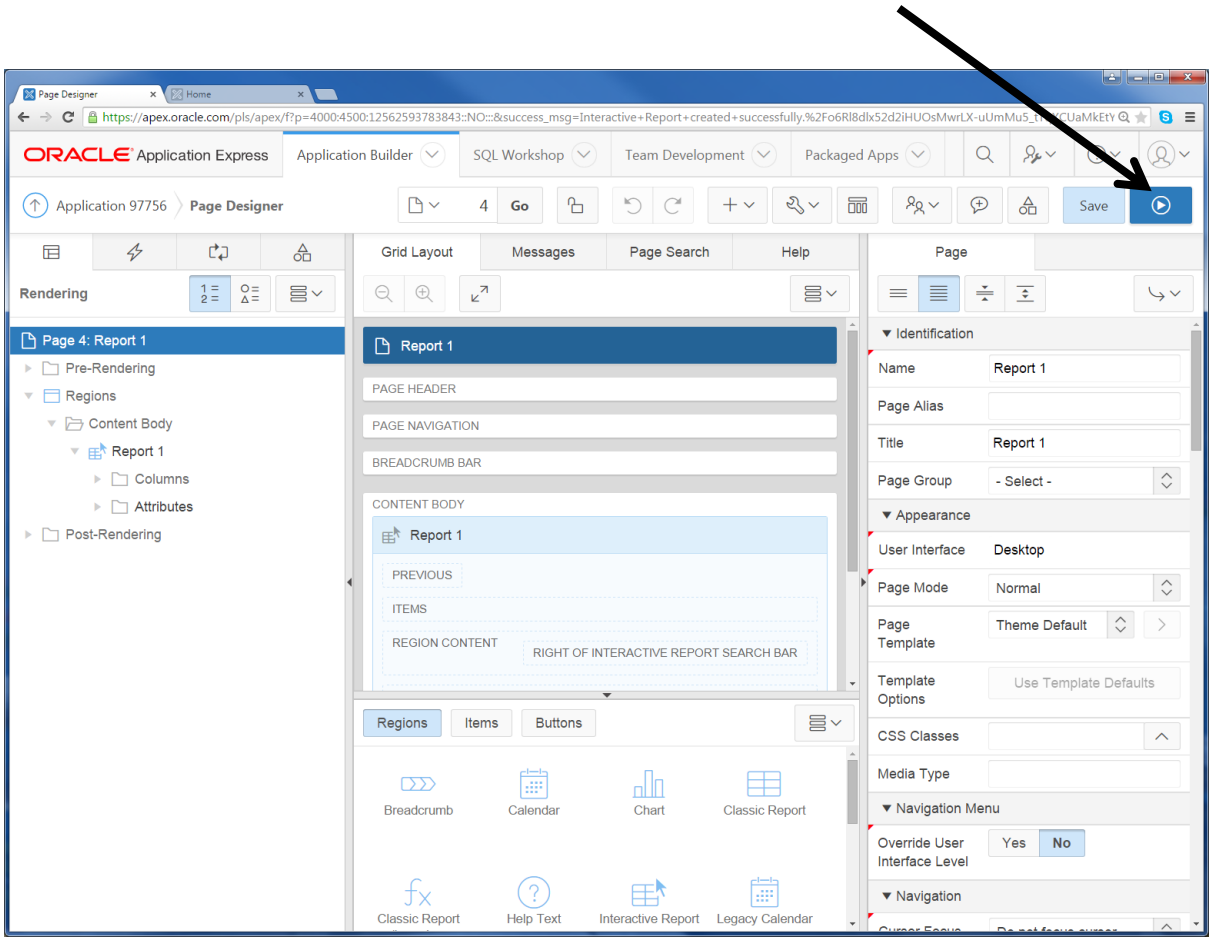
Application Home Page

33. Click on the **Report** icon on the **Create a page** screen. Then click on **Interactive Report** on the **Page and Region Attributes** screen, **click Next >**. On the **Navigation Menu** screen click **Next >**. When the Report source screen appears click in the **Table / View Name box** and **select COMPANY (table)** all the table columns will be selected automatically, click the **Next >** button.



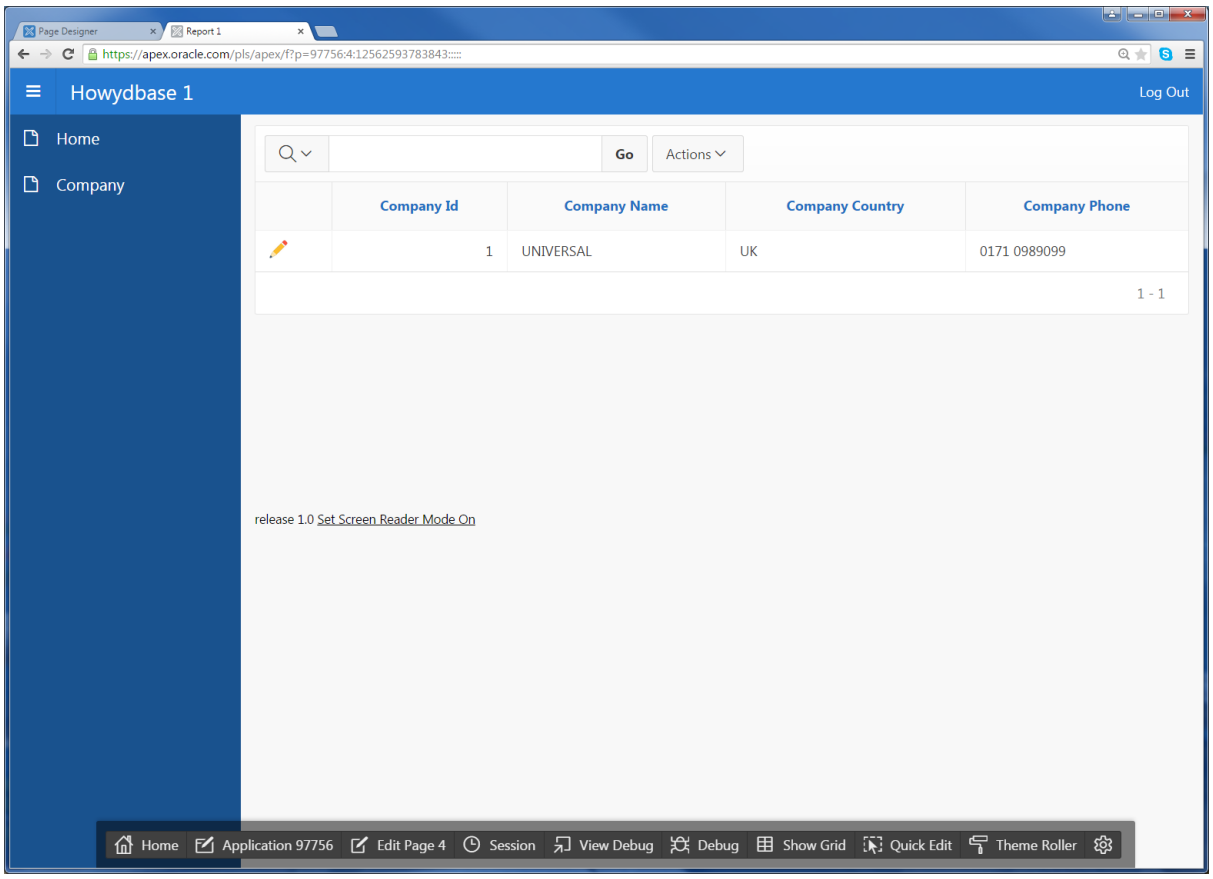
Create Interactive Report

34. On the **Confirm** page click the **Create** button. The new report will open in the Page Designer. As you are not going to make any changes click on the **Run** button.



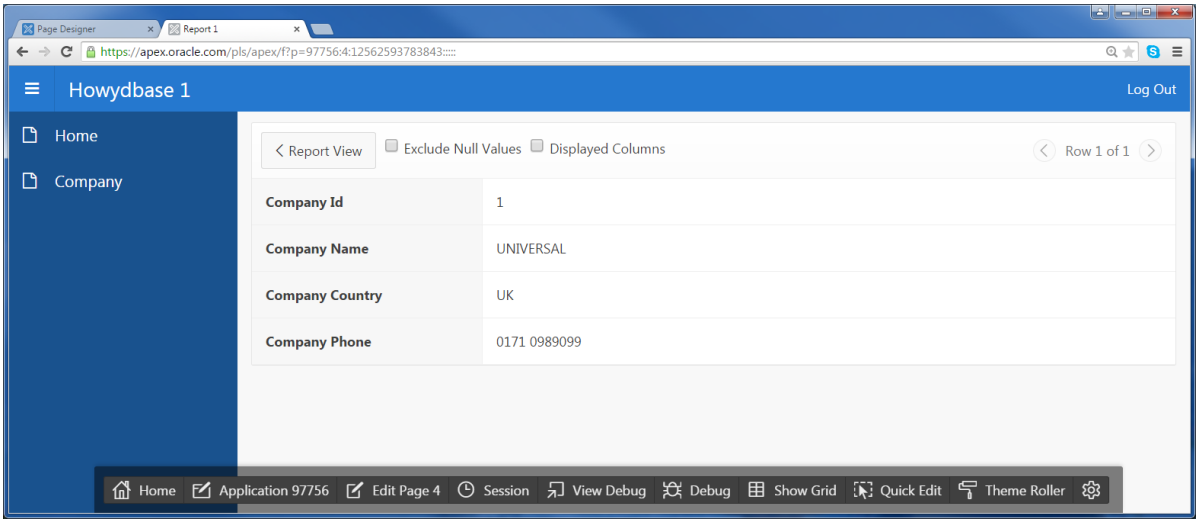
Page Designer

The **Company** Page will appear. Click on the **pencil** icon to access the report page.



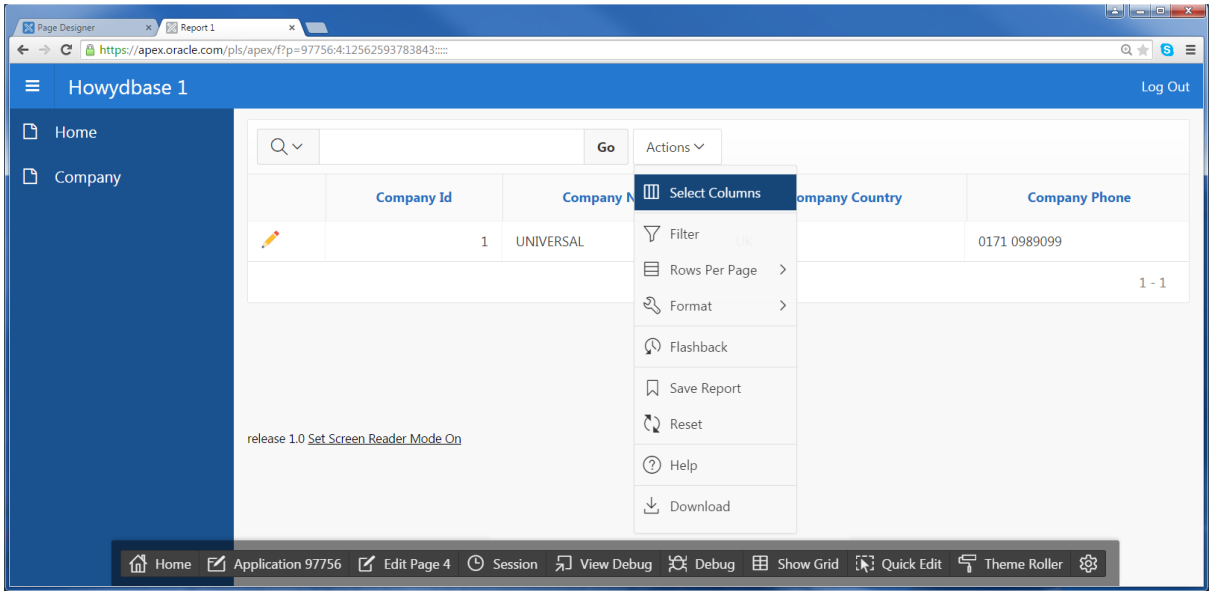
Company Page

Click on the < **Report View** button.



Report View

On the Report View page you can click on the **Actions** drop down list to alter the data displayed by choosing columns and setting the filter.



Report Actions

Exercise 1

1. Experiment by applying the different actions from the report drop down list.

Exiting the application

To exit your application just click on **Log out** in the top tool bar or choose **Home** from the developer toolbar.

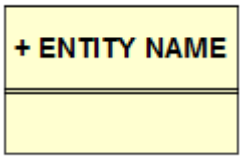
Exercise 2

1. Learn more about APEX by visiting http://www.oracle.com/technetwork/developer-tools/apex/learnmore/index.html
2. Create some more pages and reports for the Music System.

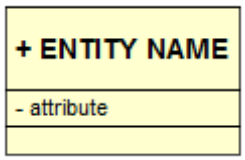
12 Appendices

12.1 Appendix A. UML Modelling Notation

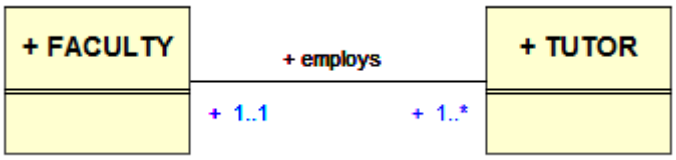
This appendix shows you how the UML Class Diagram notation can be used to produce a conceptual model showing entities as classes and relationships as associations. The following have been drawn using the QSEE CASE tool; other books and tools may show the symbols with slight differences.



A Class symbol representing an entity



A Class symbol representing an entity with an attribute



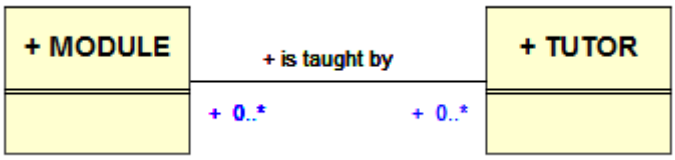
A class association (Binary relationship)

A class association is used to represent the relationship between two entity classes. It has a label and the multiplicity is shown to represent the cardinality and optionality of the relationship. The multiplicity has a minimum and a maximum value shown as follows:

0..1	is read as	zero or 1
1..1	is read as	one and only 1
0..*	is read as	zero 1 or many
1..*	is read as	one or many

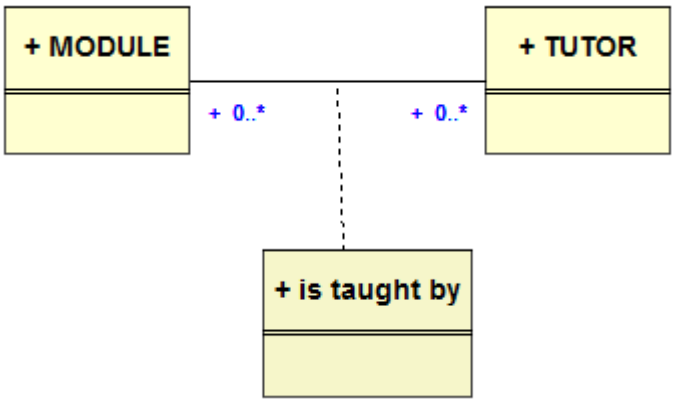
The above diagram would be read as "A faculty has one or many tutors and a tutor is employed by one and only one faculty".

To represent a many to many relationship the following is used:



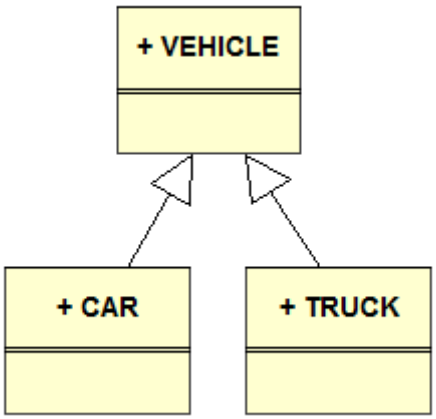
A many to many relationship

This would be read as a module is taught by zero, 1 or many tutors and a tutor teaches zero, 1 or many modules. As a many-to-many relationship holds information this information needs to be held in a new association class (think of it as a link entity) and is shown as follows:



In this case the module id and tutor id would be held in the "is taught by" class.

To represent subtypes the UML generalisation and specialisation notation is used as follows:

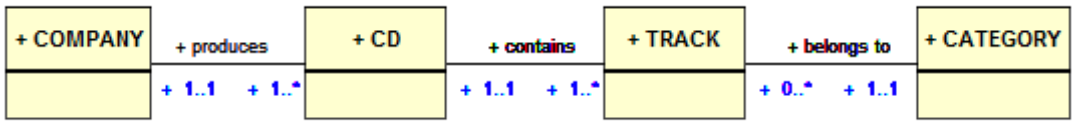


The vehicle class would be the generalisation (super class) and the car and truck classes would be specialisations (sub types).

Exercise 1

1. Redraw the ERD for the music system shown in Appendix B using the UML notation.

Exercise 1 feedback



12.2 Appendix B. Music System Specification (ERD and Tables)

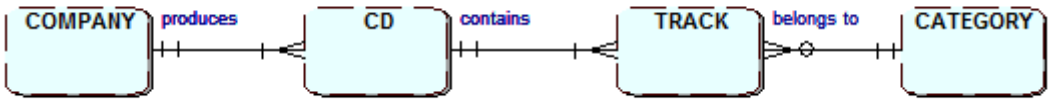


Table Name: company

Column Name	Data Type	Constraints
company_ID	NUMBER(3)	Primary Key
company_name	VARCHAR2(20)	NOT NULL
company_country	VARCHAR2(15)	NOT NULL
company_phone	Varchar2(15)	NOT NULL

Table Name: cd

Column Name	Data Type	Constraints
cd_IDno	NUMBER(3)	Primary Key
cd_title	VARCHAR2(50)	NOT NULL
cd_artist	VARCHAR2(30)	NOT NULL
cd_date_purchased	DATE	NOT NULL
cd_payment_type	VARCHAR2(6)	
cd_price	NUMBER(4,2)	
cd_company_ID	NUMBER(3)	Foreign key to company(company_ID)

Table Name: category

Column Name	Data Type	Constraints
cat_ID	NUMBER(3)	Primary Key
cat_description	VARCHAR2 (20)	Not Null

Table Name: track

Column Name	Data Type	Constraints
track_cd	NUMBER(3)	Primary Key part 1, Foreign key to cd(cd_IDno)
track_no	NUMBER(3)	Primary Key part 2
track_title	VARCHAR2(50)	Not Null
track_length	NUMBER(4,2)	
track_cat_ID	NUMBER(3)	Foreign key to category(cat_ID)

music.sql

```
DROP TABLE company CASCADE CONSTRAINTS;
CREATE TABLE company(
company_ID              number(3) PRIMARY KEY,
company_name            varchar2(20) NOT NULL,
company_country         varchar2 (15) NOT NULL,
company_phone           varchar2 (15) NOT NULL);

DROP TABLE cd CASCADE CONSTRAINTS;
CREATE TABLE cd(
cd_IDno                 number(3) PRIMARY KEY,
cd_title                varchar2(50) NOT NULL,
cd_artist               varchar2(30) NOT NULL,
cd_date_purchased       date NOT NULL,
cd_payment_type         varchar2(6),
cd_price                number(4,2),
cd_company_ID           number(3) references company(company_ID));

DROP TABLE category CASCADE CONSTRAINTS;
CREATE TABLE category(
cat_ID          number(3) primary key,
cat_description varchar2(20) NOT NULL);

DROP TABLE track;
CREATE TABLE track(
track_cd        number(3) references cd(cd_IDno),
track_no        number(3) NOT NULL,
track_title     varchar2(50) NOT NULL,
track_length    number(4,2),
track_cat_ID    number(3) references category(cat_ID),
Primary key(track_cd,track_no));
```

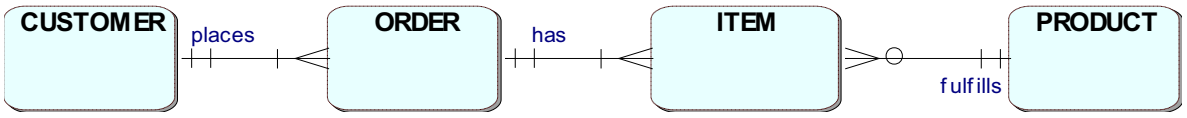
music_data.sql

```
INSERT INTO company VALUES (001,'UNIVERSAL','UK','0171 0989099');
INSERT INTO company VALUES (002,'EMI','UK','0171 0989088');
INSERT INTO company VALUES (003,'U.M.T.V.','UK','01817777666');
INSERT INTO company VALUES (004,'Ministry of Sound','UK','0181 888888');
INSERT INTO company VALUES (005,'TELSTAR','UK','0181 666666');
INSERT INTO company VALUES (006,'SONY','USA','0181 555555');

INSERT INTO cd VALUES (1,'LIVE AT LEEDS','THE WHO','01-jan-2001','Cash',10.99,001);
INSERT INTO cd VALUES (2,'CLUBLAND 4 THE NIGHT OF YOUR LIFE','VARIOUS',10-nov-2003','Cash',13.99,003);
INSERT INTO cd VALUES (3,'CLUBBERS GUIDE 2004','VARIOUS',05-jan-2005','Credit',13.99,004);
INSERT INTO cd VALUES (4,'THE RISING','BRUCE SPRINGSTEEN',30-jul-2002','Cash',10.99,006);
INSERT INTO cd VALUES (5,'ROLLING STONES FORTY LICKS','The ROLLING STONES',05-sep-2005','Credit',10.99,001);
INSERT INTO cd VALUES (6,'BEST OF BOWIE','DAVID BOWIE',08-sep-2008','Cash',10.99,002);
INSERT INTO cd VALUES (7,'PINK MISSUNDAZTOOD','PINK',28-jan-2002','Credit',5.99,006);

INSERT INTO category VALUES(1,'Rock');
INSERT INTO category VALUES(2,'Pop');
INSERT INTO category VALUES(3,'Dance');
INSERT INTO category VALUES(4,'Classical');

INSERT INTO track VALUES (1,1,'HEAVEN AND HELL',4.30,1);
INSERT INTO track VALUES (1,2,'I CAN"T EXPLAIN',2.16,1);
INSERT INTO track VALUES (1,3,'FORTUNE TELLER',2.34,1);
INSERT INTO track VALUES (1,4,'TATTOO',2.51,1);
INSERT INTO track VALUES (1,5,'YOUNG MAN BLUES',4.56,1);
INSERT INTO track VALUES (1,6,'SUBSTITUTE',2.07,1);
```

```
INSERT INTO track VALUES (1,7,'HAPPY JACK',2.13,1);
INSERT INTO track VALUES (1,8,'IM A BOY',2.40,1);
INSERT INTO track VALUES (1,9,'A QUICK ONE,WHILE HE"S AWAY',8.25,1);

NSERT INTO track VALUES (1,10,'AMAZING JOURNEY/SPARKS',7.34,1);
INSERT INTO track VALUES (1,11,'SUMMERTIME BLUES',3.20,1);
INSERT INTO track VALUES (1,12,'SHAKIN ALL OVER',4.15,1);
INSERT INTO track VALUES (1,13,'MY GENERATION',14.45,1);
INSERT INTO track VALUES (1,14,'MAGIC BUS',7.72,1);

INSERT INTO track VALUES (2,1,'IRISH BLUE',4.30,3);
INSERT INTO track VALUES (2,2,'I CANT LET YOU GO',4.30,3);
INSERT INTO track VALUES (2,3,'MARIA (I LIKE IT LOUD)',4.30,3);
INSERT INTO track VALUES (2,4,'ITS OVER NOW',4.30,3);
INSERT INTO track VALUES (2,5,'DONT LOOK BACK(ENERGY MIX)',4.30,3);
INSERT INTO track VALUES (2,6,'SHOW ME A SIGN',4.30,3);
INSERT INTO track VALUES (2,7,'I WOULD STAY(FLIP AND FILL MIX',4.30,3);
INSERT INTO track VALUES (2,8,'BEST DAYS OF OUR LIVES',4.30,3);
INSERT INTO track VALUES (2,9,'ELEMENTS',4.30,3);
INSERT INTO track VALUES (2,10,'MIXED UP WORLD',4.30,3);
IINSERT INTO track VALUES (2,11,'CONNECTED',4.30,3);
INSERT INTO track VALUES (2,12,'WILDERNESS',4.30,3);
INSERT INTO track VALUES (2,13,'CLOSE TO THE EDGE',4.30,3);
INSERT INTO track VALUES (2,14,'WHATS UP CLUB MIX',4.30,3);
INSERT INTO track VALUES (2,15,'HARDEN UP',4.30,3);
INSERT INTO track VALUES (2,16,'FAST DRIVING',4.30,3);
INSERT INTO track VALUES (2,17,'MUSIC IS LIFE',4.30,3);
INSERT INTO track VALUES (2,18,'SWEETHEART',4.30,3);

INSERT INTO track VALUES (3,1,'The Cry Little Sister (I Need U Now)',1.01,3);
INSERT INTO track VALUES (3,2,'The No Matter What You Do',1.02,3);
INSERT INTO track VALUES (3,3,'Angel City Love Me Right (Oh Sheila)',1.03,3);
INSERT INTO track VALUES (3,4,'Close To The Edge',1.04,3);
INSERT INTO track VALUES (3,5,'Remix – Dogzilla ',1.05,3);
INSERT INTO track VALUES (3,6,'Nocturnal Delight ',1.06,3);
INSERT INTO track VALUES (3,7,'Remix – Push',1.07,3);
INSERT INTO track VALUES (3,8,'Remix – Signum',1.08,3);
INSERT INTO track VALUES (3,9,'York I Need You',1.09,3);
INSERT INTO track VALUES (3,10,'Afterburn Winter Sun',1.01,3);

INSERT INTO track VALUES (3,11,'The Somebody To Love',1.01,3);
INSERT INTO track VALUES (3,12,'Step Right Up',1.01,3);

INSERT INTO track VALUES (4,1,'The Rising',2.01,1);
INSERT INTO track VALUES (4,2,'Land Of Hope And Dreams', 1.01,1);

INSERT INTO track VALUES (5,1,'Brown Sugar ',2.01,2);
INSERT INTO track VALUES (5,2,'Street Fighting Man ',4.01,2);
INSERT INTO track VALUES (5,3,'Paint It Black',4.01,2);
INSERT INTO track VALUES (5,4,'Fields Of Gold',3.01,2);
INSERT INTO track VALUES (5,5,'You Cant Always Get What You Want',3.01,2);
INSERT INTO track VALUES (5,6,'Angie',3.01,2);
INSERT INTO track VALUES (5,7,'Honky Tonk Women',3.01,2);
INSERT INTO track VALUES (5,8,'Start Me Up',3.00,2);
```

```
INSERT INTO track VALUES (6,1,'Space Oddity',2.01,2);
INSERT INTO track VALUES (6,2,'Life On Mars? ',4.01,2);
INSERT INTO track VALUES (6,3,'Changes',4.01,2);
INSERT INTO track VALUES (6,4,'Starman',3.01,2);
INSERT INTO track VALUES (6,5,'Ziggy Stardust',3.01,2);
INSERT INTO track VALUES (6,6,'Rebel Rebel',3.01,2);
INSERT INTO track VALUES (6,7,'Fame',3.01,2);
INSERT INTO track VALUES (6,8,'Golden Years',5.01,2);
INSERT INTO track VALUES (6,9,'Blue Jean',3.01,2);
INSERT INTO track VALUES (6,10,'China Girl',3.01,2);
INSERT INTO track VALUES (6,11,'Young Americans',5.01,2);

INSERT INTO track VALUES (7,1,'Get the party started',2.01,2);
INSERT INTO track VALUES (7,2,'Just like a pill',4.01,2);
INSERT INTO track VALUES (7,3,'Get the party started',4.01,2);
INSERT INTO track VALUES (7,4,'18 Wheeler',3.01,2);
INSERT INTO track VALUES (7,5,'Missundaztood',3.01,2);
INSERT INTO track VALUES (7,6,'Dear Diary',3.01,2);
INSERT INTO track VALUES (7,7,'Eventually',3.01,2);
INSERT INTO track VALUES (7,8,'Numb',5.01,2);
INSERT INTO track VALUES (7,9,'Family Portrait',3.01,2);
INSERT INTO track VALUES (7,10,'Misery',3.01,2);
```

12.3 Appendix C. Order System Specification (ERD and Tables)



CREATE TABLE customer(
Cust_no char(3) PRIMARY KEY,
Cust_name varchar2(10),
Cust_address varchar2(15));

As ORDER is a reserved word in Oracle porder is used as an alternative table name

CREATE TABLE porder(
Order_no char(3) PRIMARY KEY,
Cust_no char(3) REFERENCES customer(Cust_no),
Order_date date);

CREATE TABLE product(
Prod_no char(4) PRIMARY KEY,
Prod_desc varchar2(30),
Prod_price number(6,2));

```
CREATE TABLE item(
Order_no        char(3) REFERENCES porder(Order_no),
Prod_no         char(4) REFERENCES product(Prod_no),
Qty             number(5),
PRIMARY KEY(Order_no,Prod_no));
```

Sample data:

customer

Cust_no.	Name	Address
001	Sainsburys	Leeds 17
003	Morrisons	Leeds 1
007	Asda	Morley
015	Netto	Bradford

porder

Order No.	Cust_ No.	Date
X01	003	01-JAN-2012
Y01	007	23-DEC-2012
Z01	001	17-JAN-2013
Z02	015	04-FEB-2013

product

Prod no	Prod_Desc	Price
A301	Bread	0.50
A302	Milk	2.50
B001	Butter	1.50
C002	Chocolate	2.00

item

Order No.	Prod no	Qty
X01	A302	400
Z01	A301	750
X01	B001	900
Y01	C002	210
Z02	A302	340
Z01	C002	720

12.4 Appendix D. Normalisation Template

UNF	1NF	2NF	3NF	Relation / Table Name
1. List all the attributes below from a single document / table. 2. Identify the unique identifier / primary key. Show in bold or colour. May need an artificial key. 3. Identify any repeating attribute group(s). Show in (….) or colour.	1. Place repeated attribute group(s) if any in a new relation. 2. Include the UNF unique identifier as a foreign key in the new relation. 3. Identify the additional attribute(s) in the new relation to form a compound key with the foreign key.	1. Remove any part key dependent attributes to a new relation. 2. Identify identifier for each new relation. 3. Include foreign key in the original relation.	1. Remove any non-key dependent attributes to a new relation(s). 2. Identify the unique identifier for the new relation(s). 3. Include a foreign key in the original relation.	

13 Bibliography

Barker, R. 1990, "*CASE Method: Entity Relationship Modelling*". Addison-Wesley Professional.

Chen, Peter (March 1976). "*The Entity-Relationship Model – Toward a Unified View of Data*". ACM Transactions on Database Systems **1** (1): pp 9–36.

Codd, E.F. 1970, "*A Relational Model of Data for Large Shared Data Banks*". Communications of the ACM **13** No. 6: pp. 377–387.

Connolly, T. & Begg, C., 2015. "Database Systems A Practical Approach to Design, Implementation, and Management" 6th ed. Pearson Education.

Hay, D. & Lynott, M., 2008. *TDAN Newsletter*. [Online]
Available at: http://www.tdan.com/view-special-features/8457
[Accessed 23 February 2015].