Relational databases - design and use

This item was submitted to Loughborough University's Institutional Repository by the/an author.


Additional Information:

- This is a book.

Metadata Record: https://dspace.lboro.ac.uk/2134/10055

Version: Published

Publisher: Group D Publications Ltd.

Please cite the published version.
This item was submitted to Loughborough’s Institutional Repository (https://dspace.lboro.ac.uk/) by the author and is made available under the following Creative Commons Licence conditions.

For the full text of this licence, please go to: http://creativecommons.org/licenses/by-nc-nd/2.5/
Relational Databases
Design and Use

E-Book Edition

Ray Dawson

© LOUGHBOROUGH UNIVERSITY February 2012

All rights reserved. No part of this book may be reproduced in any form, by photostat, microfilm, retrieval system, or by any other means, without the prior permission of the author.
## Contents

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Overview</td>
</tr>
<tr>
<td>2</td>
<td>Creating the Entity Relationship Diagram</td>
</tr>
<tr>
<td>3</td>
<td>Adding Detail to the Entity Relationship Diagram</td>
</tr>
<tr>
<td>4</td>
<td>Rationalising the Entity Relationship Diagram</td>
</tr>
<tr>
<td>5</td>
<td>Creating the Entity Relationship Data Tables</td>
</tr>
<tr>
<td>6</td>
<td>Entity Relationship Modelling Case Study</td>
</tr>
<tr>
<td>7</td>
<td>Basic Data Normalisation</td>
</tr>
<tr>
<td>8</td>
<td>Advanced Data Normalisation</td>
</tr>
<tr>
<td>9</td>
<td>Normalisation and Entity Relationship Modelling</td>
</tr>
<tr>
<td>10</td>
<td>Data Normalisation Case Study</td>
</tr>
<tr>
<td>11</td>
<td>Introduction To 'SQL'</td>
</tr>
<tr>
<td>12</td>
<td>Selecting Data From the Database With SQL</td>
</tr>
<tr>
<td>13</td>
<td>Joining Database Tables Together</td>
</tr>
<tr>
<td>14</td>
<td>The SQL Data Maintenance Commands</td>
</tr>
<tr>
<td>15</td>
<td>SQL Subqueries</td>
</tr>
<tr>
<td>16</td>
<td>Views</td>
</tr>
<tr>
<td>17</td>
<td>Performance of Relational Database Management Systems</td>
</tr>
<tr>
<td>18</td>
<td>Designing the Database for Integrity</td>
</tr>
<tr>
<td>19</td>
<td>Referential Integrity</td>
</tr>
<tr>
<td>20</td>
<td>Designing the User Interface for Integrity</td>
</tr>
<tr>
<td>21</td>
<td>Multi-User Systems</td>
</tr>
<tr>
<td>22</td>
<td>Database Application Development</td>
</tr>
<tr>
<td></td>
<td>Index</td>
</tr>
</tbody>
</table>
CHAPTER 1

Overview

<table>
<thead>
<tr>
<th>Contents</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1 Why a Book on Relational Databases?</td>
<td>4</td>
</tr>
<tr>
<td>1.2 Contents Overview</td>
<td>5</td>
</tr>
<tr>
<td>1.3 Book Style</td>
<td>8</td>
</tr>
</tbody>
</table>
1.1 Why a Book on Relational Databases?

This book is based on the experience of many years that expertise in building and handling databases is not as widespread as it should be.

Student project work often illustrates the problem. So many students eagerly start to build a database application based on what they thought was a fairly obvious data structure.... .... only to find to their cost later that the database design is causing serious difficulties.

They are then faced with the unfortunate choice of:

1. struggling on with more and more fudges to get round the inadequate data structure design,

or 2. redesigning their data structure from scratch and in the process scrapping much of the work they had done so far.

This stresses the need to get the data structure right from the start of a project.... .... this takes time and a fair degree of skill.

It is the skill aspect that seems to be so often missing and this is what this book hopes to redress.

This book is aimed to support a course on databases. It does not contain large numbers of exercises, however, as it is expected that course leaders will provide students with these themselves.
1.2 Contents Overview

This book starts with the main diagrammatic means of analysing data structure requirements - entity relationship modelling. This is a proven method of deriving a database structure for any given problem. It is not perfect but it is undoubtedly the best diagrammatic database design system on offer.

The advantage of a diagrammatic system is that it aids the thinking process. Most people think better in pictures as they need to "visualise" their work.

It is also a good communication tool. "A picture is worth a thousand words" is a saying with some meaning. It is far easier for other people to understand your reasoning if you can show your thinking in diagrammatic form.

Chapters 2 to 5 take the reader through the entity relationship modelling method of deriving a database structure and this is then illustrated in a case study in chapter 6.

Chapters 7 and 8 examine the process of database normalisation to derive a data structure. This is treated in a very "common sense" manner rather than the more mathematical approach in some texts.

Normalisation is a complex process but represents the ultimate check that a data structure is correct. It is hoped this book's approach won't be too difficult for even the most un-mathematically minded reader.
Chapter 9 attempts to further the reader's understanding by showing how entity relationship modelling and normalisation are not two completely different processes but are, in fact, closely related.

Chapter 10 then takes the reader through a case study example based on normalisation to consolidate the reader's understanding of the process.

The next chapters are all concerned with SQL, the standard query language used by all major database systems on the market.

The ability to handle SQL is a skill in its own right and one which is needed by the vast majority of professional computer users.

Which is the most frequent computer language used today? C, C++, Visual Basic, Java and even Cobol are usually given as the answer to this question. Yet SQL is undoubtedly used by more software professionals than any of these.

Many database systems attempt to hide the SQL from the users. It is possible to develop simple applications in Access, for example, without knowing any SQL. However, any significant development usually sees the developer manipulating the SQL directly sooner or later.

The SQL is introduced in chapter 11, with chapters 12, 13 and 15 devoted to the most complicated command, the **SELECT** statement.
Chapter 14 gives details of the SQL commands for maintaining and manipulating the data tables.

Chapter 16 describes the view, a feature available in SQL and many other database languages. The reasons for using a view are discussed as well as the syntax for creating and deleting views.

Chapter 17 looks at the performance of database management systems. The facilities provided by the software and the general methods for optimising the systems performance are described.

Chapters 18 to 21 all look at the methods for preserving database integrity.

Chapter 18 looks at the general facilities available to prevent data errors and chapter 19 looks at the particular case of referential integrity - a problem of particular importance in relational database systems.

Chapter 20 looks at features in the user interface that can assist in preventing data error, and chapter 21 looks at the particular problem of more than one user using the system at the same time.

Finally chapter 22 examines the whole software development process in relation to database systems. Prototyping is shown to be of particular relevance as it can assist in the design of the data structure, and is advocated as the main method of database system development.
1.3 Book Style

This book is written in a style which involves keeping paragraphs short, frequent itemising in point form, and separate topics on each page.

The style is perhaps more consistent with a set of notes than a text book. This is, in fact, where its origins lie. It is based on notes for modules taught to university students and on notes for short courses given to industry.

No attempt has been made to significantly change the style, however, as it not meant to be a "good read" that you would want to curl up with late at night.

This book is intended to be a working text. It will have the dual purpose of a teaching manual and a reference manual usable by both beginners and practitioners of the subject.

It is hoped the format will make the book easy to read, easy to find what the reader is looking for, and above all, easy to understand.

The reader is warned, however, understanding relational databases is not a simple matter .... you need to allow a bit of time and patience to get to the bottom of the subject!

Any constructive comments on the book can be addressed to the author directly via email on:

R.J.Dawson@Lboro.ac.UK
# CHAPTER 2

## Creating the Entity Relationship Diagram

<table>
<thead>
<tr>
<th>Contents</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1 Entity Relationship Modelling</td>
<td>10</td>
</tr>
<tr>
<td>2.2 Step 1 - Identify the Entities and Attributes</td>
<td>11</td>
</tr>
<tr>
<td>2.3 Data Tables For Entities</td>
<td>12</td>
</tr>
<tr>
<td>2.4 Key Fields For Each Table</td>
<td>13</td>
</tr>
<tr>
<td>2.5 Should an Attribute be an Entity?</td>
<td>14</td>
</tr>
<tr>
<td>Hint 1: Variable numbers of an Attribute</td>
<td></td>
</tr>
<tr>
<td>2.6 Should an Attribute be an Entity?</td>
<td>15</td>
</tr>
<tr>
<td>Hint 2: Optional Attributes</td>
<td></td>
</tr>
<tr>
<td>2.7 Should an Attribute be an Entity?</td>
<td>16</td>
</tr>
<tr>
<td>Hint 3: Attributes With Attributes</td>
<td></td>
</tr>
<tr>
<td>2.8 Should an Attribute be an Entity?</td>
<td>17</td>
</tr>
<tr>
<td>Hint 4: Repeated, Long Text Attributes</td>
<td></td>
</tr>
<tr>
<td>2.9 Should an Attribute be an Entity?</td>
<td>18</td>
</tr>
<tr>
<td>Hint 5: Text Attributes Used In Matchings</td>
<td></td>
</tr>
<tr>
<td>2.10 Single Instance Entities</td>
<td>19</td>
</tr>
<tr>
<td>2.11 Step 2 - Determine the Relationships</td>
<td>20</td>
</tr>
<tr>
<td>2.12 Processes Verses Relationships</td>
<td>21</td>
</tr>
<tr>
<td>2.13 Positioning the Relationship Name</td>
<td>22</td>
</tr>
<tr>
<td>2.14 Alternative Notations</td>
<td>23</td>
</tr>
<tr>
<td>2.15 Merging Entities</td>
<td>24</td>
</tr>
<tr>
<td>2.16 Should Two Entities be Merged?</td>
<td>25</td>
</tr>
<tr>
<td>2.17 Reflective Entity Relationships</td>
<td>26</td>
</tr>
<tr>
<td>2.18 Step 3 - Combine into One Diagram</td>
<td>27</td>
</tr>
</tbody>
</table>
2.1 Entity Relationship Modelling

The Objectives:

1. To help develop an understanding of the nature of the data complexities in the system.

2. To be able to organise the data into a logical, "structured" form with each data item where it may be expected (useful for maintenance).

3. To be able to organise the data in an efficient way so that data is neither duplicated nor omitted.

The End Result:

The end result is the "system data dictionary" which defines:

1. The data tables that appear in the system.

2. The fields (ie. columns) for each entry in the table.

3. A key for each table, consisting of one or more of these fields.
2.2 Step 1 - Identify the Entities and Attributes

An "entity" is something that has an independent existence.

   Eg. Items, Components, Sales reps, Customers

Anything that provides information about an entity is referred to as an "attribute" of the entity.

   Eg. Description, Quantity, Name, Identity Number

It may not be clear whether an item is an entity or an attribute . . . it is possible for the same thing to be an entity in one analysis, but an attribute in another.

   Eg. Shape, Tax, Manufacturer, Job

If in doubt, make it an entity to start with.

(The analysis will later determine whether it should be an attribute)

Step 1 is a first shot at determining the data bases for a given problem, giving:

- A data table for each type of entity.
- An entry for each entity of that type.
- Fields for each data table entry to represent the attributes.
2.3 Data Tables For Entities

Eg. The employee entity for a company may have the attributes: Name, ID number, Job title, Salary.

The resulting employee table would be:

<table>
<thead>
<tr>
<th>Name</th>
<th>ID number</th>
<th>Job title</th>
<th>Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fred Bloggs</td>
<td>10001</td>
<td>Director</td>
<td>30,000</td>
</tr>
<tr>
<td>Joe Brown</td>
<td>12941</td>
<td>Salesman</td>
<td>20,000</td>
</tr>
<tr>
<td>Alice Cooke</td>
<td>18802</td>
<td>Manager</td>
<td>23,000</td>
</tr>
<tr>
<td>John Smith</td>
<td>10457</td>
<td>Engineer</td>
<td>18,000</td>
</tr>
<tr>
<td>Sarah King</td>
<td>13098</td>
<td>Accountant</td>
<td>28,000</td>
</tr>
</tbody>
</table>

A company car entity with attributes: registration number, make, model and year would give the table:

<table>
<thead>
<tr>
<th>Reg.No.</th>
<th>Make</th>
<th>Model</th>
<th>Year</th>
</tr>
</thead>
<tbody>
<tr>
<td>N100 ARK</td>
<td>Ford</td>
<td>Mondeo</td>
<td>1995</td>
</tr>
<tr>
<td>P346 VRY</td>
<td>Vauxhall</td>
<td>Astra</td>
<td>1996</td>
</tr>
<tr>
<td>R882 PRG</td>
<td>Peugeot</td>
<td>406</td>
<td>1997</td>
</tr>
<tr>
<td>N241 WET</td>
<td>Ford</td>
<td>Mondeo</td>
<td>1995</td>
</tr>
<tr>
<td>K318 BAR</td>
<td>Vauxhall</td>
<td>Cavalier</td>
<td>1992</td>
</tr>
</tbody>
</table>

Note: Even though each company car may be associated with a single employee, there is no tie up between the entity tables at this stage.

ie. There is nothing in:

   the employee table to say which car is owned

or

   the company car table to say who owns it.

This comes later in the analysis.
2.4 Key Fields For Each Table

Each row in each data table must be unique.

If there is a chance that two rows may be the same a new attribute must be added to make each row unique.
   Eg. The ID number in the employee table.

Any column in which all entries are unique is called a key field. A table must have:

either: A single key field column.
   or: A combination of columns that together form a unique key, and is known as a multiple key.

Examples of single key fields are:
   1. The ID number in the employee table.
   2. The Reg.No. in the company car table.

An example of a multiple key would be:
Suppose all employees must belong to one of a number of recognised trade unions...
   ... the union membership number alone is not unique...
   ... but a combination of union name and the union membership number is unique, so these columns would provide a multiple key.

In such instances a new single key field is often introduced for convenience.
2.5 Should an Attribute be an Entity?

**Hint 1: Variable numbers of an Attribute**

Check the attributes of all entities -

*Are there variable numbers of an attribute?*

Eg. An item may have several different colours.

If so then make the attribute a separate entity.

*Otherwise there will be blank fields in the data base.*

This can give unexpected entities such as Colour.

If an item can have several colours, and different items have different numbers of colours then the colour must be a separate entity.

But: If all the items had multiple colours but each had the *same* number of colours there would be no need of a separate colour entity.
2.6 **Should an Attribute be an Entity?**

**Hint 2: Optional Attributes**

Check the attributes of all entities -

*Is any attribute optional?*

Eg. The passport number attribute of a person entity.

If so, then strictly speaking it should be made an entity.

*Otherwise there will be blank fields in the data base.*

In practice if most occurrences of this entity have this attribute (eg. most people have a passport.) then it is commonly left as an attribute.

This is a compromise . . .

. . . simplicity is being gained at the expense of wasted space in the data base.
2.7 **Should an Attribute be an Entity?**

**Hint 3 : Attributes With Attributes**

Check the attributes of all entities -  

*Has any attribute got an attribute of its own?*

Eg. A person entity may have a car type and a car length as attributes . . .  

. . . in this case the car length is really an attribute of the car type.

If an attribute has an attribute of its own then it should be made into a separate entity.

*Otherwise the attributes are interdependent and there is a risk that the data base could become inconsistent.*

Eg. A persons car type may be changed with the user forgetting to change the car length . . .  

. . . this could lead to two people with the same car type but with different car lengths.
2.8 **Should an Attribute be an Entity?**

**Hint 4 : Repeated, Long Text Attributes**

Check the attributes of all entities -

*Is any long text attribute likely to be repeated for different occurrences of the entity?*

ie. More than one person may have the same car type, such as "Ford Mondeo Estate".

If there are many repeats

or the text is long

then there can be a significant space saving by making this attribute into a separate entity referenced by an id number.

ie. A separate table should be created with the text and an associated identity number . . .

. . . other tables will then refer to the id number rather than the text itself.
2.9 **Should an Attribute be an Entity?**

**Hint 5: Text Attributes Used In Matchings**

Check the attributes of all entities -

*Is any text attribute likely to be tested for matching entries?*

Even if the text is short with only a few repeats . . .

if tests may be made for matching entries,
then it should be a separate entity.

Eg. A **TOWN** attribute could have any of the entries:

<table>
<thead>
<tr>
<th>Leicester</th>
<th>Leics</th>
<th>Leic</th>
<th>Leics.</th>
<th>Leic.</th>
</tr>
</thead>
<tbody>
<tr>
<td>LEICESTER</td>
<td>LEICS</td>
<td>LEIC</td>
<td>LEICS.</td>
<td>LEIC.</td>
</tr>
</tbody>
</table>

This is OK if it is only used as part of an address. . .

**But** . . . if matching on the name is to be done,  
(eg. to list all people from Leicester)
then entries may be missed!

*If* the field is made a separate entity
*then* each name would be stored once only  
*with only one representation.*
2.10 **Single Instance Entities**

Some entities by the nature of the way they are handled can only ever have one instance.


Other entities may have all instances considered to be identical with no need to distinguish between them.

Eg. Backing Store, Member of Public, Train

- **These do not qualify as proper entities!**
- **They should not appear in any entity diagram!**
- **They will not appear in any data table!**

To include these single entities in any entity diagrams would:

1. Increase the entity diagram size and hence increase the diagram's complexity.
2. Produce no useful outcome as no table needs to be derived to store data about them.

ie. Single instance entities hinder rather than help the understanding of the system data.

*In other words - forget all single instance entities!*
2.11 **Step 2 - Determine the Relationships**

These can be represented in a diagram using a [Entity name] to represent an entity with connecting lines to represent relationships in the form:

![Diagram showing a relationship between two entities, with labels for the relationship and inverse relationship.]

The relationship labelling can be above or below the connecting line.

For example:

- Item \hspace{2cm} is on \hspace{2cm} is for \hspace{2cm} Order
- Customer \hspace{2cm} places \hspace{2cm} is placed by \hspace{2cm} Order
- Sales Rep \hspace{2cm} drives \hspace{2cm} is driven by \hspace{2cm} Car
2.12 **Processes Verses Relationships**

Time considerations are largely ignored in entity relationship analysis.....

.... but the entity relationship diagram produced must be accurate for any *instant* in time.

Eg. The "relationship"

![Diagram]

is **not a valid relationship** because:

1. At any one instant in time a person cannot be both a job applicant and an employee.

2. At any one instant in time there is no connection between any of the current job applicants and any of the current employees.

*ie. "becomes/was created from" is a process....

....*not a relationship!*
2.13 **Positioning the Relationship Name**

To save space on an entity diagram:

- It is not necessary to put both the relationship and its inverse on the connecting line, providing the meaning is clear.

- The relationship name can be put centrally if the relationship direction is obvious.

eg. A Car would not drive a Sales Rep so the meaning and direction of the relationship is obvious:

![Diagram of a Car drives Sales Rep relationship](image1)

If the direction of the relationship is not obvious the relationship wording should be nearer the "from" entity:

![Diagram of an Accountant advise Engineer relationship](image2)

This shows that the engineers advise the accountants, not the other way round.
2.14 Alternative Notations

Some entity diagram notations involve a ◇ to label the relationship:

\[ \text{Car} \quad \text{drives} \quad \text{Sales Rep} \]

But this notation may make the diagrams more cluttered.

Some notations (eg. SSADM) allow the naming of the relationship to be omitted altogether ....

.... but this is **NOT** recommended.

It is important because there may be more than one relationship which will lead, eventually, to more than one data table.

\[ \text{Eg.} \]

\[ \text{Tradesman} \quad \text{Buys materials from} \quad \text{Company} \]

\[ \text{Tradesman} \quad \text{Sells goods to} \quad \text{Company} \]

Such multiple relationships can easily be missed if the relationships are not named.
2.15 Merging Entities

The attributes and relationships may show that some entities are very similar.

ie. • The number of attributes is the same.
    • The types of attributes are the same.
    • They relationships with other entities are the same (or nearly the same).

Eg. Two entities are EC Worker and Non EC Worker

Each has attributes: ID Number, Job Title, Salary

Each has relationships: belongs to Department and works on Project

The Non EC Worker also has the relationship:
    Non EC Worker must have a Work Permit

These could be merged into one entity, Worker, if:

1. The Work Permit relationship becomes optional.
    ie. Workers may have a Work Permit.

2. The attributes indicate the type of worker.

   This could mean either:
   1. adding a new "worker type" column
   2. Making an existing column specify the worker type (Eg. Only Non EC workers have ID numbers > 1000)
2.16 **Should Two Entities be Merged?**

There is no fixed rule over whether or not entities should be merged....

.... it depends on how the entity tables will be used.

If the information to be derived will usually require putting data from both tables together

*then* the table should be merged.

Eg. For the EC / Non EC Workers example, operations such as:

- totalling salaries for each department
- seeing which engineers work on a certain project

.... would be better served by a single merged table.

Whereas operations such as:

- checking who’s work permit will soon expire
- allocating only EC Workers to defence projects

.... would be better served by separate entity tables.

*It is for the database designer to judge which is the most appropriate data structure for each separate application.*
2.17 **Reflective Entity Relationships**

It is possible for entities to have a relationship with other entities of the same type.

eg. Person is a friend of person

and, if a sales item such as a "crate of wine" contains other sales items such as "bottle of wine"

Item contains item

These are known as **reflective** relationships and are shown on the entity diagram as follows:

![Reflection Diagram]

Note: Reflective relationships are relatively rare....

... frequently it is more convenient to split the entity. Eg. "Container item" and "Internal item"
2.18 Step 3 - Combine into One Diagram

No one entity appears in more than one place:
CHAPTER 3

Adding Detail to the
Entity Relationship Diagram

Contents

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Dependent Entities</td>
<td>29</td>
</tr>
<tr>
<td>3.2</td>
<td>Showing Dependent Entities</td>
<td>30</td>
</tr>
<tr>
<td>3.3</td>
<td>Classifying the Relationships</td>
<td>31</td>
</tr>
<tr>
<td>3.4</td>
<td>Alternative Notations</td>
<td>32</td>
</tr>
<tr>
<td>3.5</td>
<td>Obligatory and Non Obligatory Relationships</td>
<td>33</td>
</tr>
<tr>
<td>3.6</td>
<td>Using a Range of Membership to Show Optionality</td>
<td>34</td>
</tr>
<tr>
<td>3.7</td>
<td>Ambiguity in Notations for Optionality</td>
<td>35</td>
</tr>
<tr>
<td>3.8</td>
<td>Time Considerations for Optionality</td>
<td>36</td>
</tr>
<tr>
<td>3.9</td>
<td><em>Either - Or</em> Relationships</td>
<td>37</td>
</tr>
<tr>
<td>3.10</td>
<td>The Full Entity Relationship Diagram</td>
<td>38</td>
</tr>
</tbody>
</table>
3.1 Dependent Entities

Each instance of dependent entity only exists because of an instance of another, different entity.

These are sometimes called "detail" or "weak" entities, each being like an attribute of the other "master" or "strong" entity.

There are two types of dependent entities:

1. A detail entity where each instance is associated with only one of the master entity.
   
   *Eg. A company car entity where each and every car is allocated to just one sales rep entity.*
   
   Instances of detail entity cease to exist when the corresponding master entity no longer exists....
   
   ie. The car is sold when the sales rep leaves the company.

2. A detail entity table where each instance may be associated with several instances of the master entity.
   
   *Eg. A job title entity where each title may be associated with several employee entities.*
   
   The detail entity would be deleted when the last corresponding master entity is deleted.
   
   ie. There are no job titles kept for non existent employees.
3.2 **Showing Dependent Entities**

Both types of dependent entity are shown with a double outline box:

- **Sales rep** drives **Car**
- **Employee** has **Job Title**

It is not essential to show dependent entities any differently to other entities as the data table structure derived will be no different.

But .... knowing the dependent entities does enable the corresponding data tables to be handled correctly when the master entity table is modified.

Some entity relation diagram notations allow for *all* attributes of an entity to be shown in circular boxes:

eg.

.... but this is not recommended as it leads to cluttered diagrams that are hard to follow.
Adding Detail to the Entity Relationship Diagram

3.3 Classifying the Relationships

A relationship may be:

1 to 1
Eg.: Husband/Wife

1 to N
(one to many)
Eg.: Father/Child

N to M
(many to many)
Eg.: Person/ Possessions

These classifications may be represented in the form:

\[ \text{Father} \xrightarrow{1} \text{is parent of} \xrightarrow{\text{N}} \text{Child} \]
3.4 **Alternative Notations**

One to many relationships

One | Many
---|---
1 | N

---

---

---

---

---
3.5 **Obligatory and Non Obligatory Relationships**

If membership of a relationship is optional for an entity it may be shown by:

![Diagram](image)

**Eg.**

Employee **earns** Bonus

Some employees get one or more bonuses but some don't. Every bonus is given to an employee.

Employee **has own** Computer

Every employee has one or more computers. Some computers are not assigned to any particular employee.

Employee **gets** Expenses Payment

Some employees will get one or more expenses payments but some will not. Some expenses payments will go to employees but some will not (eg. get paid to consultants).
3.6 Using a Range of Membership to Show Optionality

If an entity must belong to a relationship it can be considered to be connected to a minimum of one of the other entity.

If it's membership of the relationship is optional then it is connected to a minimum of zero of the other entities.

A range giving minimum as well as maximum order of membership can be shown to indicate optionality:

\[
\begin{align*}
\text{Obligatory} & \quad \text{Optional} \\
\{ & \\
\text{ie.} & \\
\} & \\
\text{is equivalent to} & \\
\{ & \\
0-1 & 1-1 \\
0-N & 1-N \\
\} & \\
\text{or} & \\
\end{align*}
\]
3.7 **Ambiguity in Notations for Optionality**

A circle can also be used to indicate optionality:

![Diagram showing obligated and optional notations](image)

But a circle is sometimes used to indicate a minimum of zero with no corresponding lines to indicate a minimum of one:

![Diagram showing obligated and optional notations](image)

This means the diagram can be interpreted either way round!

Therefore, if a circle is to be used *always* put a key to make your notation absolutely clear!
3.8 Time Considerations for Optionality

The entity relationship diagram must be true for any instant in time....

ie. it must allow for even short lived situations.

This can make membership of some relationships optional even if intuitively this would not be the case.

Eg. If all sales reps drive company cars ....

but it may take a few weeks to arrange a company car when they first start ....

then the relationship must be shown as:

```
Sales Rep  ----- drives ----- Company Car
```

ie. At any instant in time there may be some sales reps without cars ....

.... so for the sales rep the relationship is considered to be optional.
3.9 *Either - Or* Relationships

Mutually exclusive alternative options are shown by an arc crossing each relationship line.

Suppose
- *all* sales reps drive cars
  - when they start they drive a hire car
  - this is then replaced by a company car

ie. All sales reps drive *either* a hire car *or* a company car.

This is shown by:

```
Sales Rep  Company Car  Hire Car
```

A dashed line is used *only* if sales reps may exist who drive *neither* a hire car *nor* a company car.

ie:

```
Sales Rep  Company Car  Hire Car
```
### 3.10 The Full Entity Relationship Diagram

The diagram shows the relationships between entities in a relational database. The entities are:

- **Customer**
- **Order**
- **Sales Rep**
- **Item**
- **Company car**
- **Hire car**

The relationships are:

- **Customer** places **Order**
- **Order** sells to **Sales Rep**
- **Sales Rep** drives **Item**
- **Order** contains **Item**
- **Item** sells **Order**
- **Order** buys **Item**
- **Order** wins **Item**

The diagram illustrates how these entities interact with each other in the context of a database system.
CHAPTER 4

Rationalising the Entity Relationship Diagram

Contents

4.1 Eliminating Redundant Relationships 40
4.2 Ambiguity When Eliminating Relationships 41
4.3 Loss of Connection When Eliminating Relationships 42
4.4 The Entity Relationship Diagram After Eliminating Redundant Relationships 43
4.5 Finding New Dependent Entities 44
4.6 Checking To See If The Relationships Tell Us Everything We Need To Know 45
4.7 3-Way Relationships 46
4.8 When Not To Introduce A Multi-way 47
4.9 Looking for "Hidden" Entities 48
4.10 Example of a "Discovered" Entity 49
4.11 New Relationships for a Discovered Entity 50
4.12 3-Way Relationships and Discovered Entities 51
4.1 **Eliminating Redundant Relationships**

Some relationships duplicate information shown in other relationships:

![Diagram showing relationships between Branch, Employee, and Department]

The fact that an employee works in a branch of the company follows from:

1. All employees work in a department
2. All departments form part of a branch

ie. The branch - employee relationship is redundant and can be eliminated:

![Diagram showing simplified relationships between Branch, Employee, and Department]
4.2 Ambiguity When Eliminating Relationships

Care must be taken that there is no loss in information when eliminating relationships:

```
Branch
  \_ employs
\     \_ comprises
  \_ Employee \_ Department
```

The above at first appears to have eliminated a redundant relationship - but which department does each employee belong to?

Care must be taken not to model a relationship with an indirect connection of the type N:... + ...:N such as:

```
A
  \_ B
\_ C
```

This gives ambiguous information about how A is connected to C.

This is known as the "fan trap" as there is a "fan" (ie. a \( \leftarrow \)) at both ends.
4.3 Loss of Connection When Eliminating Relationships

If some of the connections are optional then care must be taken to avoid losing connections when eliminating relationships.

This is known as the "chasm trap".

For example:

*If*

some branch employees were not allocated to any particular department (eg. directors)

*ie.*

the employees membership of the "works in" relationship is optional

*then*

the branch - employee relationship is **not** redundant.

For some employees the connection to the branch is now lost!
4.4 The Entity Relationship Diagram After Eliminating Redundant Relationships

Order places Customer
Sales Rep drives Company car
Item contains Hire car
Company car drives Item
Sales Rep wins Item
Order is for Item
Sales Rep sells Item

4.5 Finding New Dependent Entities

Once the entity relationship diagram has been simplified it will often reveal new dependent entities.

If any entity has a relationship with only one other entity and it must belong to that relationship ....

.... then it can be considered to be a dependent entity:
4.6 Checking To See If The Relationships Tell Us Everything We Need To Know

Eg In the example we have:

- We know which sales reps have sold what items.  
  *Eg. Fred and Flo may both have sold items A and B on various different customer's orders.*

- We know which sales reps have won what orders.  
  *Eg. Fred and Flo may both be responsible for order X.*

- We know which items are on what orders.  
  *Eg. Order X may be for items A and B.*

**But** We don't know who managed to get each individual item onto each order.

  *ie. Did Fred sell A and Flo sell B on order X or was it vice-versa?*
4.7 3-Way Relationships

Circular relationships that cannot be eliminated as in the Sales Rep - Order - Item relationships may need to be represented as a 3-way relationship:

Unfortunately it is difficult to represent this type of relationship in words!

Note:

1. It may not be possible to create a 3-way relationship. Not all cycles of relationship pairs can be represented this way.

2. It is theoretically possible to have 4-way or even more complex multi-way relationships - though these are rare in practice.
4.8 When Not To Introduce A Multi-way Relationship

This 3-way relationship should only be created if it provides information that is needed.

Eg. In the sales rep - order - item example do we need to know who is responsible for what on each order?

If not, the relationship should be left as 3 pairs as before.

Why?

Although it may look simpler there may be a lot of data needed to define a 3-way relationship.

Eg. If there are 10 sales reps, 10 orders and 10 items then ....

a 3-way relationship represents a possible $10 \times 10 \times 10 = 1000$ possible combinations ....

whereas, the 3 relationship pairs represent only $10 \times 10 + 10 \times 10 + 10 \times 10 = 300$ possible combinations.

Each situation must be considered separately to determine which is the most accurate and useful way to model the real relationships.
4.9 **Looking for "Hidden" Entities**

Many-Many Relationships can be difficult to handle - as will be seen later when special relationship tables will be needed.

It may be useful to invent another entity between these many-many entities to:

- Assist the understanding of these relationships
- Enable them to be handled more easily.

ie.

This new entity will be an obligatory 'N' of a 1 to N relationship with each of the other two entities.

When considered, this new entity may turn out to represent something meaningful.

In which case:

- It may have attributes of its own
- It may have relationships with other entities
### 4.10 Example of a "Discovered" Entity

If a 3-way has not been created in the sales entity relationship diagram there is still the many-many relationship:

![Diagram](image)

Introducing "Sale", meaning the sale of one type of item on one order gives:

![Diagram](image)

There is now two 1:N relationships with the new entity connected to one and only one of each of the original two entities.

**But note** - This new entity will not make any difference to the data tables that will be derived from the diagram unless it is found to have:

- either: its own attributes
- or: its own relationships with other entities

These make the new entity worth creating.

**Note**: Although this entity is not directly "tangible" itself, there are associated things which are very real - eg. the bill!
4.11 **New Relationships for a Discovered Entity**

Once a new entity is discovered it may be found that other relationships to this entity exist and these may lead to other relationships becoming redundant:
4.123-Way Relationships and Discovered Entities

3-way and other multi-way relationships may be represented by a discovered entity - this may assist in the understanding of the relationship.

Eg. A 3-way relationship created for the "Sales rep - Order - Item" cycle could also give a discovered entity called "Sale":

![Diagram of 3-way relationships]

This is exactly the same diagram as created when discovering an entity for the 2-way relationship and then finding a third relationship for the new entity.

_ie. There may be equally valid alternative routes to derive the final entity relationship diagram._
CHAPTER 5

Creating the Entity Relationship
Data Tables

Contents

5.1 Creating Entity Data Dictionary Entries 53
5.2 Creating the Relationship Links
  Method 1 : Merging the Entity Tables 54
5.3 Compromise Use of Method 1 55
5.4 Create the Relationship Links - Method 2 :
  Adding the Key of One Table to the Other 56
5.5 Compromise Use of Method 2 57
5.6 Create the Relationship Links
  Method 3 : Creating a Relationship Table 58
5.7 Example of a Relationship Table 59
5.8 Discovered Entity Tables 60
5.9 Either-Or Relationships 61
5.10 Handling Either-or Relationships as Options 62
5.11 Shared Foreign Key Fields 63
5.12 Shared Foreign Key Field Example 64
5.13 Single Column Entity Tables 65
5.14 Entities Forming a Continuous Sequence 66
5.15 Time Intervals 67
5.16 Advantages and Disadvantages of
  Entity Relationship Modelling 68
5.17 Entity Relationship Modelling : Conclusion 69
5.1 Creating Entity Data Dictionary Entries

The data dictionary is a description of the layout for each data table in the system with:

- A separate table for each entity type.
- A separate row for each instance of that type of entity in the entity table.
- A separate column for each entity attribute.
- A unique table key based on one or more of the associated columns.

Note: If a key consists of two or more fields, it may be convenient to invent a new field to act as a key, such as an identity number.

Eg.

*Item Table*

<table>
<thead>
<tr>
<th>Item Code</th>
<th>Description</th>
<th>Manufacturer</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1234</td>
<td>Pogo Stick</td>
<td>Bouncy Toys plc</td>
<td>£10</td>
</tr>
<tr>
<td>801-bd</td>
<td>&quot;Bonzo&quot; doll</td>
<td>LoadaRubbish ltd</td>
<td>£5</td>
</tr>
<tr>
<td>18/zap</td>
<td>Lazer gun</td>
<td>Futuristics UK ltd</td>
<td>£50</td>
</tr>
<tr>
<td>BossMk2</td>
<td>Inflatable HoD</td>
<td>Gas-filler plc</td>
<td>£1000</td>
</tr>
</tbody>
</table>

*Order Table*

<table>
<thead>
<tr>
<th>Order ID</th>
<th>Order Date</th>
<th>Delivered?</th>
<th>Discount</th>
</tr>
</thead>
<tbody>
<tr>
<td>2001</td>
<td>10/11/92</td>
<td>yes</td>
<td>0%</td>
</tr>
<tr>
<td>2002</td>
<td>01/02/93</td>
<td>no</td>
<td>10%</td>
</tr>
<tr>
<td>2003</td>
<td>01/02/93</td>
<td>partly</td>
<td>5%</td>
</tr>
</tbody>
</table>
5.2 Creating the Relationship Links

Method 1: Merging the Entity Tables

Applicable to:

Obligatory 1 to 1 Relationships

Eg. For the relationship:

Sales Rep \( \text{drives} \) Car

with entity tables:

Rep.ID \( \rightarrow \) Name, Sales, Commission
Reg.No. \( \rightarrow \) Make, Model, Miles

If every sales rep drives a car
and every car is driven by a sales rep
then the tables can be combined into:

Rep.ID \( \rightarrow \) Name, Sales, Commission, Reg.No.,
Make, Model, Miles

Alternatively the Reg.No. could be the key field.

Notice that any attribute mistakenly designated as a separate entity will be absorbed into the appropriate entity table at this point.
5.3 **Compromise Use of Method 1**

**Applicable to:**

**Obligatory-Optional 1 to 1 Relationships**

\[ \square - \cdot - \cdot - \square \]

*If* every car is driven by a sales rep  
*but* a sales rep does **not necessarily** drive a car  
*then* the tables can be merged to:

\[
\text{Rep.ID} \rightarrow \text{Name, Sales, Commission,} \\
\text{Reg.No., Make, Model, Miles.}
\]

**Note:** This is bending the rules with the disadvantage that:

- There will be some null fields in the database, this wastes space.

- Some fields will be interdependent in that, if any of the fields Reg.No., Make, Model and Miles are empty for a particular sales rep, then **all** of these fields **must** be empty.

Whether or not such a compromise is desirable will depend on the proportion of null fields in the table.

If there are too many null fields it would be better to use method 2 to create the relationship link.
5.4 **Create the Relationship Links**

**Method 2 : Adding the Key of One Table to the Other**

**Applicable to:**

- Obligatory-Optional 1 to 1 Relationships

![Diagram of Obligatory-Optional 1 to 1 Relationships]

- 1 to N Relationships Obligatory for the 'N' Entity

![Diagram of 1 to N Relationships Obligatory for the 'N' Entity]

In this case the key field of the optional or '1' entity table is added to the obligatory or 'N' entity table.

*If* every car is driven by one sales rep

*but* a sales rep does *not necessarily* drive a car

*or* a sales rep may drive *more than one* car

*then* the relationship can be modelled by:

```
Rep.ID -> Name, Sales, Commission
Reg.No. -> Make, Model, Miles, Rep.ID
```

Note the field(s) containing the key for a different table is known as a **foreign key**.

eg. Rep.ID is a foreign key in the Car table.
5.5 **Compromise Use of Method 2**

Applicable to:

Optional-Optional 1 to 1 Relationships

Optional for the 'N' Entity

1 to N Relationships Optional for the 'N' Entity

If a car is not necessarily driven by a sales rep
and a sales rep does not necessarily drive a car
or a sales rep may drive more than one car
then the relationship can be modelled by:

Rep.ID -> Name, Sales, Commission
Reg.No. -> Make, Model, Miles, Rep.ID

**But Note:** This is once again bending the rules giving some null fields in the Rep.ID field in the Car table - and this wastes space.

Whether or not such a compromise is desirable will depend on the proportion of null fields in the table.

If there are too many cars not driven by sales reps it would be better to create a relationship table as in method 3 to create the relationship link.
5.6 **Create the Relationship Links**

**Method 3 : Creating a Relationship Table**

**Applicable to:**

Optional-Optional 1 to 1 Relationships

Optional-Optional 1 to N Relationships Optional for the 'N' Entity

N to N Relationships, Optional or Obligatory

If a car is not necessarily driven by a sales rep

or a car may be driven by more than one sales rep

and a sales rep does not necessarily drive a car

or a sales rep may drive more than one car

then a **Relationship Table** can relate the entities with

- Fields (columns) corresponding to the key fields of the related entities.
- A separate entry (row) for each instance of a relationship between the entities.

**N.B.** This is the **only way** to model many to many relationships
5.7 Example of a Relationship Table

A relationship table will only have columns corresponding to the 2 keys of the 2 entities it relates ....

.... this usually gives a table with just 2 columns.

Eg. A many to many relationship between the Order and Item entities would give:

<table>
<thead>
<tr>
<th>Order-Item Relationship Table</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Order ID</strong></td>
</tr>
<tr>
<td>2001</td>
</tr>
<tr>
<td>2001</td>
</tr>
<tr>
<td>2002</td>
</tr>
<tr>
<td>2003</td>
</tr>
<tr>
<td>2003</td>
</tr>
<tr>
<td>2003</td>
</tr>
</tbody>
</table>

Neither column is unique .... the two columns make a multiple key.

3-way relationships would have a further column(s) corresponding to the key of the third entity:

<table>
<thead>
<tr>
<th>Order-Item-Sales Rep 3-way Relationship Table</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Order ID</strong></td>
</tr>
<tr>
<td>2001</td>
</tr>
<tr>
<td>2001</td>
</tr>
<tr>
<td>2002</td>
</tr>
</tbody>
</table>
5.8 Discovered Entity Tables

A discovered entity will have as its key a combination of fields that correspond to the key fields of the entities it was "discovered" between.

If a discovered entity has no further attributes of its own then the resulting table has no further columns...

.... but this is exactly the same as the relationship table that would have existed had the entity not been discovered!

This shows that unless:

1. The entity has attributes of its own

or

2. The entity is needed as further entities will relate to it

then "discovering" an entity gives no advantage.

In the example of the "Sale" being a discovered entity a possible extra attribute may be the number of each item on the order:

"Sale" Discovered Entity Table

<table>
<thead>
<tr>
<th>Order ID</th>
<th>Item Code</th>
<th>First Name</th>
<th>Second Name</th>
<th>Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>2001</td>
<td>P1234</td>
<td>Fred</td>
<td>Bloggs</td>
<td>12</td>
</tr>
<tr>
<td>2001</td>
<td>18/zap</td>
<td>Joe</td>
<td>Smith</td>
<td>100</td>
</tr>
<tr>
<td>2002</td>
<td>BossMk2</td>
<td>Fred</td>
<td>Bloggs</td>
<td>1</td>
</tr>
</tbody>
</table>
5.9 **Either-Or Relationships**

Often the two alternative entities, of an either-or relationship (B and C) are very similar in nature.

Eg. Company Cars and Hire Cars are clearly similar.

If the data record for each entity is also similar, (the entity tables have the same number and type of columns)

then

• the entity tables can be combined into one
• an extra column should define the entity type

Eg Suppose both the Company Car and the Hire Car entities had attributes:

\[
\text{Reg.No. } \rightarrow \text{ Make, Model, Miles}
\]

The tables should be combined into one with an extra Boolean field to signify whether the car was a company car or hire car:

\[
\text{Reg.No. } \rightarrow \text{ Make, Model, Miles, Car type}
\]

The entity diagram can then be revised and redrawn and the associated tables derived accordingly.
5.10 Handling Either-or Relationships as Options

If the alternative entities do not have records of identical format.
then it is possible to treat the relationship in exactly the same way as optional relationships.

ie. \[
\begin{array}{c}
A \\
B \\
C
\end{array}
\] can be treated as
\[
\begin{array}{c}
A \\
B \\
C
\end{array}
\]

But:
1. This hides the either-or nature of the relationship
2. The resulting tables would not prevent an 'A' entity having a relationship with both 'B' and 'C'.

However the relationships must be treated as optional relationships if either B or C are the 'N' entity of a 1 to N or N to N relationship:

ie. \[
\begin{array}{c}
A \\
B \\
C
\end{array}\] must be treated as
\[
\begin{array}{c}
A \\
B \\
C
\end{array}\]
and \[
\begin{array}{c}
A \\
B \\
C
\end{array}\] must be treated as
\[
\begin{array}{c}
A \\
B \\
C
\end{array}\]
5.11 **Shared Foreign Key Fields**

If the alternative fields in an either-or relationship are both the '1' end of a 1 to 1 or 1 to N relationship and the key fields of both are of the same format and the keys are distinct from each other then the keys of both may be stored in the same field in the linking entity table.

ie. In the case of

![Diagram](A) ![Diagram](B) ![Diagram](C)

a foreign key field in the field in the 'A' entity table could hold the key to either the 'B' table or the 'C' table.

And in the case of

![Diagram](A) ![Diagram](B) ![Diagram](C)

a foreign key field in the 'A' entity table could hold the key to either the 'B' table or the 'C' table or it could be null.

If the B and C keys are not distinct, or not obvious an additional field in the A entity table can be used to indicate whether the foreign key refers to B or C.
### 5.12 Shared Foreign Key Field Example

In the example:

![Diagram showing relationships between Sales Rep, Company car, and Hire car]

If the entity tables contain:

- **Sales Rep Table:**
  - Rep.ID -> Name, Sales, Commission

- **Company Car Table:**
  - Reg.No. -> Make, Model, Miles

- **Hire Car Table:**
  - Reg.No. -> Hire company, Hire rate

then in this case:

1. The Company Car table and Hire Car table cannot be merged as the data is different.

2. The relationships *could* be modelled by adding the Rep.ID to each car table:
   - Reg.No. -> Make, Model, Miles, Rep.ID
   - Reg.No. -> Hire company, Hire rate, Rep.ID

3. Better to add the Reg.No. to the Sales Rep table with a field to distinguish the car type:
   - Rep.ID -> Name, Sales, Commission, Reg.No., Car type
5.13 Single Column Entity Tables

An entity does not need to have any attributes other than the key field or fields.

ie. It is possible for an entity with a single field key to have its key as the only field ....

.... this will give a single column table!

Eg. A theatre seat entity may have only the seat number as a field.

This is perfectly acceptable . . .

But . . . A list of all possible instances of that entity is the only information this table provides.

ie. . . . If this information is not required . . .

. . . the entity table can be eliminated.

(though associated relationship tables will still exist)

Eg. A colour entity may exist because an item may have variable numbers of colours . . .

The table may have only one column - the colour.

If it is not necessary to list what colours the item may have, this table can be eliminated.
5.14 **Entities Forming a Continuous Sequence**

Entity Relationship Modelling assumes:

- entities are not part of any continuous sequence.
- there is, therefore, no ordering or sequence to the rows in the corresponding table.

But sometimes there is a significant sequence ....

.... to ignore this is a waste of data storage space.

Eg. Theatre seats may be given different prices depending on where they are in the theatre.

Conventional analysis would give a seat entity with a price attribute:

<table>
<thead>
<tr>
<th>Seat Row</th>
<th>Seat No.</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1</td>
<td>£5.00</td>
</tr>
<tr>
<td>A</td>
<td>2</td>
<td>£6.00</td>
</tr>
</tbody>
</table>

However, in practice adjacent seats will usually be at the same price which gives the possibility of a more memory efficient table as:

<table>
<thead>
<tr>
<th>Seat Row</th>
<th>Start No.</th>
<th>Finish No.</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1</td>
<td>20</td>
<td>£5.00</td>
</tr>
<tr>
<td>A</td>
<td>21</td>
<td>40</td>
<td>£6.00</td>
</tr>
</tbody>
</table>

It is possible that this may have been considered from the outset as a "seat block" entity with a price attribute for the seat block ......

.... however this could easily be missed.
5.15 **Time Intervals**

A common entity sequence is a series of time slots.

Eg. A recreation facility, such as a tennis court, may be bookable in half hour time slots.

Conventional analysis will once again produce a bookings table with a row for each time slot.

No account will be taken of the fact the tennis court is often booked for two or more hours at a time and this will give four or more rows in the bookings table.

Once again it would use less data storage to represent the data with a start time and a finish time.

As with the theatre seat example it is possible to recognise the problem from the outset by creating a time interval entity ....

.... but again this could be easily missed.

This type of storage optimisation is far more easily recognised at the end of the analysis when the tables are derived ....

.... and it only takes a simple adjustment to alter the tables to the optimum format.

*It is worth checking to see if a database derived through entity relationship analysis has a table with a sequence of entities that can be optimised in this way.*
5.16 **Advantages and Disadvantages of**

**Entity Relationship Modelling**

**Advantages:**

The structure given to the data design means that:

1. Data will not be duplicated
2. Data will not be accidentally omitted
3. Empty data fields are avoided
4. It is easy to add new data
5. It is easy to delete old or redundant data
6. It is easy to modify and update existing data
7. It is easy to select and retrieve data for processing

**Disadvantages:**

*None*: Entity relationship modelling can only be of assistance to analysis and design.

*But*: It may not be much use for certain types of system.

ie. It is of little use for systems that are complicated in ways other than data complexity.
5.17 **Entity Relationship Modelling : Conclusion**

Entity Relationship Modelling provides a systematic approach for handling complicated data systems.

This gives:

1. *An understanding of the system data*
   
   This is useful for detailed requirements analysis.
   
   It may give *all* the information and understanding required for some types of system.

2. *A overall structure to the data*
   
   This is useful for the overall system design and the detailed design.
   
   This may provide nearly all of the design work required for some types of system.

*ie.* **Entity Relationship Modelling is useful as both an analysis tool and a design tool.**
# CHAPTER 6

## Entity Relationship Modelling Case Study

### Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.1</td>
<td>Entity Relationship Modelling Example</td>
<td>71</td>
</tr>
<tr>
<td>6.2</td>
<td>Identifying Entities and Attributes</td>
<td>72</td>
</tr>
<tr>
<td>6.3</td>
<td>Identifying the Relationships</td>
<td>73</td>
</tr>
<tr>
<td>6.4</td>
<td>Considering the Attributes of the Entities</td>
<td>74</td>
</tr>
<tr>
<td>6.5</td>
<td>Considering the Merger of Entities</td>
<td>75</td>
</tr>
<tr>
<td>6.6</td>
<td>Updating The Relationship List74</td>
<td>76</td>
</tr>
<tr>
<td>6.7</td>
<td>Categorising The Relationships</td>
<td>77</td>
</tr>
<tr>
<td>6.8</td>
<td>The Full Entity Relationship Diagram</td>
<td>78</td>
</tr>
<tr>
<td>6.9</td>
<td>Looking for Redundant Relationships</td>
<td>79</td>
</tr>
<tr>
<td>6.10</td>
<td>Looking for 3-Way Relationships</td>
<td>80</td>
</tr>
<tr>
<td>6.11</td>
<td>Looking for Hidden Entities</td>
<td>81</td>
</tr>
<tr>
<td>6.12</td>
<td>The Final Entity Relationship Diagram</td>
<td>82</td>
</tr>
<tr>
<td>6.13</td>
<td>Final Solution: The Entity Data Tables</td>
<td>83</td>
</tr>
<tr>
<td>6.14</td>
<td>Final Solution: The Relationship Tables</td>
<td>84</td>
</tr>
</tbody>
</table>
6.1 Entity Relationship Modelling Example

**What data tables are needed for a company to keep the following records of personnel and project activities?**

For each permanent employee there is a requirement to keep:
- his/her name
- department title and identity code
- the department budget code
- job title
- current projects
- names and dates of training courses attended

The training courses are internal courses set up by each department for their own personnel. Training course details should only be recorded if at least one person has attended.

In addition there are a number of contract workers who need the following information recorded:
- his/her name
- contract hire company name and address
- job title
- current projects

Details of a contact hire company may be kept even if there is currently no contract worker on hire from that company.

There is a requirement to keep, for each project:
- the project budget code
- the start date
- the target completion date

Some larger projects are subdivided into smaller projects which also must be recorded.

A person may work on more than one project at one time. The number of hours worked on each project needs to be recorded for accounting purposes.

It would be useful to record details of a project before anybody participates in it.
### 6.2 Identifying Entities and Attributes

**Possible entities:**
- Employee, Department, Training Course, Contractor, Contract Company, Project, Hours

**Attributes for the suggested entities:**

<table>
<thead>
<tr>
<th>Entity</th>
<th>Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Employee</td>
<td>Name, Job Title</td>
</tr>
<tr>
<td>Department</td>
<td>Department Code, Department Name, Department Budget Code</td>
</tr>
<tr>
<td>Training Course</td>
<td>Course Title, Course Date</td>
</tr>
<tr>
<td>Contractor</td>
<td>Name, Job Title</td>
</tr>
<tr>
<td>Contract Company</td>
<td>Contract Company Name</td>
</tr>
<tr>
<td>Project</td>
<td>Project Name, Project Budget Code, Start Date, Target Date</td>
</tr>
<tr>
<td>Hours</td>
<td>Date, Start Time, Finish Time</td>
</tr>
</tbody>
</table>

**Note 1:** Many of these entities are also likely to need ID numbers for convenient handling.

**Note 2:** The "Hours" entity seems a little odd but the analysis should clarify this later.
6.3 Identifying the Relationships

- Employee works in Department
- Department sets up Tr. Course
- Employee has attended Tr. Course
- Contractor is hired from Contract Co.
- Employee works for Hours
- Contractor works for Hours
- Employee works on Project
- Contractor works on Project
- Hours are worked on Project
- Project is divided into Project
6.4 Considering the Attributes of the Entities

Two entity attributes are fairly long text fields that could be repeated for different entity instances.

An employee or a contractor has a job title:
  eg. "Software Engineering Manager"

A training course will have a course title:
  eg. "Using the Personnel Database"

For the purposes of this exercise we shall assume:
1. The same job title can occur many times.
2. Information will be required regarding people of the same job title.
   eg. How many Software Engineers worked on project X?
3. Course titles change too frequently so there will be few repeats.
4. With few repeats of course title no information based on course title will be practical.

Either (1) or (2) would be enough to suggest that job title is a separate entity:

  [Employee] has [Job Title]

eg.

Assumptions (3) and (4) suggest course title should remain as an attribute.
6.5 Considering the Merger of Entities

The Employee and Contractor entities are similar.

They both have attributes: ID Number, Name

Both have relationships:  
- works for Hours
- works on Project
- has Job Title

The Employee has the additional relationships:  
- works in Department
- has attended Training Course

The Contractor has the additional relationship:  
- is hired from Contract Company

Assuming the database will be frequently used to look at data on a project basis from both the employee and contractor tables ....

.... the entity tables should be merged into one, "Person" table by:

1. Making the ID Numbers distinctive  
eg. ID numbers for Contractors > 10000

2. Making the Department and Contract Company either-or relationships.

3. Making the Training Course relationship optional  
(though it was never obligatory for the employee entity anyway!)
6.6 Updating The Relationship List

Person \textit{works in} Department

Person \textit{is hired from} Contract Co.

Person \textit{has attended} Tr. Course

Department \textit{sets up} Tr. Course

Person \textit{works for} Hours

Person \textit{has} Job Title

Person \textit{works on} Project

Hours \textit{are worked on} Project

Project \textit{is divided into} Project
6.7 Categorising The Relationships

Either

Person \(\xrightarrow{\text{works in}}\) Department

or

Person \(\xrightarrow{\text{is hired from}}\) Contract Co.

Person \(\xrightarrow{\text{has attended}}\) Tr. Course

Department \(\xrightarrow{\text{sets up}}\) Tr. Course

Person \(\xrightarrow{\text{works for}}\) Hours

Person \(\xrightarrow{\text{has}}\) Job Title

Person \(\xrightarrow{\text{works on}}\) Project

Hours \(\xrightarrow{\text{are worked on}}\) Project

Project \(\xrightarrow{\text{is divided into}}\) Project

The relationships "Person works in Department" and "Person is hired from Contract Company" are shown as obligatory for the Person entity as the person must belong to either one or the other relationship.
6.8 The Full Entity Relationship Diagram

Note that the Contract Co. entity is not a dependant entity despite only having a relationship with one other entity, Person ....

.... This is because a Contract Co. can exist in the database without having any contract worker currently being hired from the company.
6.9 **Looking for Redundant Relationships**

In the Person/Department/Training Course cycle:

We know .....  
(1) who is in each department  
(2) at least one department member must have attended any course set up by the department  

So we can deduce which courses a department has set up  
*ie.* *The relationship "Department sets up Training Course" is redundant*  

Noting that .....  
(1) Department is related to Person only  
(2) A department without anyone in it doesn't exist  
*ie.* *Department is dependent on the Person entity*  

(1) Training Course is related to Person only  
(2) It is only necessary to keep a record of a Training Course if a person has attended  
*ie.* *Training Course is dependent on the Person entity*  

In the Person/Project/Hours cycle:  

*No redundancies possible because of the "Fan Trap".*
6.10 Looking for 3-Way Relationships

For the Person/Project/Hours relationship cycle:

If we need to know which individual hours (eg. 2pm to 5pm Tuesday of a particular week), are worked on each project by each person ....

then .... we need a three way relationship.

but .... it is only the total hours that is needed for each person on each project.

This tells us a 3-way relationship is not required.

However, it brings into question the "Hours" entity ...

.... do we really need it in this form at all?

The next step may help sort out this rather odd entity.
6.11 **Looking for Hidden Entities**

Checking all remaining Many to Many relationships:

1. The Person - Training Course relationship has a potential discovered entity called, say, a "Training Course Place".

   But there are
   
   (1) no useful attributes of this entity,
   
   (2) no other relationships to this entity,
   
   so there is no advantage in discovering this entity.

2. The Person - Project relationship has a potential discovered entity called a "Project Role".

   Looking for attributes for Project Role ....

   .... the total hours spent on a project role is necessary for accounting purposes.

   Looking for relationships with Project Role ....

   .... there is obviously a relationship with the Hours entity but the new total hours attribute records all that is needed ....

   .... so we can get rid of the awkward "Hours" entity altogether!
6.12 The Final Entity Relationship Diagram

Note that the Department and Training Course are now dependant entities as they have a relationship with only one other entity, Person ....

.... Unlike a Contract Company, a Department or Training Course would not be needed in the database if there were no corresponding Person entity.
6.13 **Final Solution: The Entity Data Tables**

Putting the "1" entity key fields into the "N" entity tables to represent 1 to N relationships gives:

**Person Table:**
- Person ID -> Person Name, Job ID, Employee Type, Organisation Code

**Department Table:**
- Department Code -> Department Name, Department Budget Code

**Contract Company Table:**
- Contract Company Code -> Contract Company Name

**Job Table:**
- Job ID -> Job Title

**Training Course Table:**
- Course ID -> Course Title, Course Date

**Project Role Table:**
- Person ID, Project ID -> Total Hours Worked

**Project Table:**
- Project ID -> Project Name, Budget Code, Start Date, Target Date

**Note:** The Employee Type could be a boolean field if such a field type was available. Alternatively it could be a "C" for a contractor or an "E" for an employee.

The Organisation Code would correspond to either a Department Code or a Contract Company Code.
6.14 Final Solution: The Relationship Tables

The N to N relationship for Training Courses gives:

Training Course Attendance Table:
Person ID, Course ID

The Project - *is divided into* - Project relationship is 1 to N but it is *optional* for the "N" entity.

This means, strictly speaking, a table should be created to represent the relationship of the form:

Project Sub-Division Table:
Master Project ID, Sub Project ID

An alternative solution

If most projects are in fact sub projects of master projects then this relationship could be represented by extending the project table as in:

Project Table:
Project ID -> Project Name, Budget Code,
Start Date, Target Date,
Master Project ID

But it will mean some rows of the project table will have null entities for the Master Project ID field.

(This will be the only possible null field in the data.)
CHAPTER 7

Basic Data Normalisation

<table>
<thead>
<tr>
<th>Contents</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.1 Data Normalisation</td>
<td>86</td>
</tr>
<tr>
<td>7.2 Levels of Normalisation</td>
<td>87</td>
</tr>
<tr>
<td>7.3 First Normal Form</td>
<td>88</td>
</tr>
<tr>
<td>7.4 Problems With Tables Not In First Normal Form</td>
<td>89</td>
</tr>
<tr>
<td>7.5 Putting a Table into First Normal Form</td>
<td>90</td>
</tr>
<tr>
<td>7.6 Keys Fields</td>
<td>91</td>
</tr>
<tr>
<td>7.7 Dependent Fields</td>
<td>92</td>
</tr>
<tr>
<td>7.8 Second Normal Form</td>
<td>93</td>
</tr>
<tr>
<td>7.9 Problems of Tables Not in Second Normal Form</td>
<td>94</td>
</tr>
<tr>
<td>7.10 Putting Tables into Second Normal Form</td>
<td>95</td>
</tr>
<tr>
<td>7.11 Third Normal Form</td>
<td>96</td>
</tr>
<tr>
<td>7.12 Putting Tables into Third Normal Form</td>
<td>97</td>
</tr>
</tbody>
</table>
7.1 Data Normalisation

Using Entity-Relationship Analysis the aim is to create a well structured set of data tables without any unnecessary duplication of data.

But....

What check do we have that the optimum data structure has been achieved?

And....

What if we do not have a clean sheet start for the data structure design - suppose a set of tables already exist, how can we improve on the structure?

To achieve these aims a technique was invented by E.J.(Ted) Codd in 1974 known as

"Normalisation"
7.2 **Levels of Normalisation**

3 levels of normalisation were originally suggested but following input from others this has now been extended to 5.

These levels are known as the first through to the fifth normal forms, (1NF, 2NF, .... 5NF) each higher level being an improvement over lower levels.

A refinement of the third normal form also exists known as the Boyce-Codd normal form (BCNF).

An even higher level known as the domain-key normal form has also been suggested.... .... but this is rather obscure and is really only of theoretical interest.

The advantage of normalisation is that putting tables into ever higher normal forms:

1. Removes and duplication of data
2. Makes the tables easier to change and maintain

ie. *It can make a good data structure out of a bad one!*
7.3 First Normal Form

A table that is not in first normal form is easy to detect.... ..... to be in first normal form a table cannot have more than one entry in any field.

Eg. A table not in first normal form:

*Table of sales representatives showing each persons sales speciality and the office they use as a base:*

<table>
<thead>
<tr>
<th>First Name</th>
<th>Second Name</th>
<th>Sales Speciality</th>
<th>Base Office</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice</td>
<td>Alty</td>
<td>computers,printers,netware</td>
<td>London N.</td>
</tr>
<tr>
<td>Bert</td>
<td>Brown</td>
<td>software</td>
<td>Sheffield</td>
</tr>
<tr>
<td>Cecil</td>
<td>Clarke</td>
<td>computers,printers,software</td>
<td>London N.</td>
</tr>
<tr>
<td>Doris</td>
<td>Davis</td>
<td>software,netware</td>
<td>Bristol</td>
</tr>
</tbody>
</table>

The problem:

Most database management systems cannot handle tables that are not in first normal form unless implemented using:

- either (1) a long text field.
- or (2) a field repeated several times.
7.4 Problems With Tables Not In First Normal Form

In the example Sales Rep table:

The problems with using a long text field for the sales speciality are:

- Searching for a particular sales speciality requires searching for sub strings in the text field.... .... this is slow if it is possible at all.

- Inserting, modifying or deleting sales specialities will be difficult, slow and error prone.

- There could be wasted space with many partially filled text fields.

The problems with using a field repeated several times with, say, three sales speciality fields are:

- Three separate fields need to be searched when trying to find a particular speciality.

- Inserting, modifying or deleting sales specialities will be difficult, slow and error prone.

- There will be wasted space with null fields.

- If a fourth speciality was acquired by any Sales Rep the method could not handle it at all.
7.5 **Putting a Table into First Normal Form**

To put a table into first normal form, duplicate the rest of the row for each entry in the offending field:

*So the table:*

<table>
<thead>
<tr>
<th>First Name</th>
<th>Second Name</th>
<th>Sales Speciality</th>
<th>Base Office</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice</td>
<td>Alty</td>
<td>computers,printers,netware</td>
<td>London N.</td>
</tr>
<tr>
<td>Bert</td>
<td>Brown</td>
<td>software</td>
<td>Sheffield</td>
</tr>
<tr>
<td>Cecil</td>
<td>Clarke</td>
<td>computers,printers,software</td>
<td>London N.</td>
</tr>
<tr>
<td>Doris</td>
<td>Davis</td>
<td>software,netware</td>
<td>Bristol</td>
</tr>
</tbody>
</table>

*becomes:*

<table>
<thead>
<tr>
<th>First Name</th>
<th>Second Name</th>
<th>Sales Speciality</th>
<th>Base Office</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice</td>
<td>Alty</td>
<td>computers</td>
<td>London N.</td>
</tr>
<tr>
<td>Alice</td>
<td>Alty</td>
<td>printers</td>
<td>London N.</td>
</tr>
<tr>
<td>Alice</td>
<td>Alty</td>
<td>netware</td>
<td>London N.</td>
</tr>
<tr>
<td>Bert</td>
<td>Brown</td>
<td>software</td>
<td>Sheffield</td>
</tr>
<tr>
<td>Cecil</td>
<td>Clarke</td>
<td>computers</td>
<td>London N.</td>
</tr>
<tr>
<td>Cecil</td>
<td>Clarke</td>
<td>printers</td>
<td>London N.</td>
</tr>
<tr>
<td>Cecil</td>
<td>Clarke</td>
<td>software</td>
<td>London N.</td>
</tr>
<tr>
<td>Doris</td>
<td>Davis</td>
<td>software</td>
<td>Bristol</td>
</tr>
<tr>
<td>Doris</td>
<td>Davis</td>
<td>netware</td>
<td>Bristol</td>
</tr>
</tbody>
</table>

Clearly there is much duplication in the new format, but this is taken care of when putting the table into higher normal forms.
7.6 **Keys Fields**

To consider higher normal forms the concept of a key must be clear.

If a field is different for every row of a table then it can be used to identify each row and hence it is called a *key* field.

Alternatively, there may not be any one field that is unique for all rows - a combination of columns may be required for a key.

Eg. The key for the sales team table is a combination of the rows First name, Second name, and Sales speciality.

N.B. A key field or combination of fields must be unique for *all possible* rows as well as existing rows.

For the sales team it is assumed that no two people will have the whole name the same....

.... but it may be possible for two people to have either a common first name or a common second name....

.... therefore both names must be included in the key.

Note: The assumption that each whole name is unique is dangerous - this is why people are frequently referred to by an ID number of some sort.
7.7 Dependent Fields

The concept of dependent fields is also needed for consideration of higher normal forms.

If we know what the value in a particular field will be once we know the values in some other fields, we can say that field is dependent on those other fields.

Eg. Once we know the complete sales rep name we know what the base office will be.

ie. The base office is dependent on the first and second names.

However, we don't know the sales speciality for a given sales rep's name as there may be several possibilities.

ie. The sales speciality is not dependent on the sales rep's name.

All fields will be dependent on the key fields in a table.

If a field or set of fields determines another then these are said to be determinant fields.

Eg. The base office depends on the first and second names, therefore the first and second names are the determinants of the base field.
7.8 **Second Normal Form**

For a table to be in the second normal form all non key fields must depend on the whole table key.

ie. If a table is in first normal form it can only fail to be in second normal form if it has more than one field for the key.

Eg. The sales team table is not in second normal form because:

(1) The key consists of the first name, second name and sales speciality fields.

(2) The base office is dependent on only part of that key, the first name and second name.
7.9 **Problems of Tables Not in Second Normal Form**

(1) Information is duplicated.

eg. The data that Alice Alty is based in North London exists in three separate rows.

This is a waste of storage space

(2) Data entry is laborious and more error prone as unnecessary extra input is required

(3) Altering the data is difficult as it will need to be altered in more than one place

Eg. If Alice Alty moved to the Bristol office

(4) There is a possibility of anomalies in the data through errors of input or incomplete alteration such that the data becomes inconsistent.

(5) Null fields are more likely to be required.

Eg. If a new sales rep is entered before he has any sales speciality a row must be created with a null speciality field.

(6) Information is more easily lost when data is deleted.

Eg. If the sales company stops dealing in software and all rows containing this data are deleted, then the name Bert Brown and information giving his base office is lost.
### 7.10 Putting Tables into Second Normal Form

Two (or more) tables must be created to replace the original to put the table in second normal form.

Each field that depends on less than the whole key must be removed and put into a new table:

#### Table 1:

<table>
<thead>
<tr>
<th>First Name</th>
<th>Second Name</th>
<th>Sales Speciality</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice</td>
<td>Alty</td>
<td>computers</td>
</tr>
<tr>
<td>Alice</td>
<td>Alty</td>
<td>printers</td>
</tr>
<tr>
<td>Alice</td>
<td>Alty</td>
<td>netware</td>
</tr>
<tr>
<td>Bert</td>
<td>Brown</td>
<td>software</td>
</tr>
<tr>
<td>Cecil</td>
<td>Clarke</td>
<td>computers</td>
</tr>
<tr>
<td>Cecil</td>
<td>Clarke</td>
<td>printers</td>
</tr>
<tr>
<td>Cecil</td>
<td>Clarke</td>
<td>software</td>
</tr>
<tr>
<td>Doris</td>
<td>Davis</td>
<td>software</td>
</tr>
<tr>
<td>Doris</td>
<td>Davis</td>
<td>netware</td>
</tr>
</tbody>
</table>

#### Table 2:

<table>
<thead>
<tr>
<th>First Name</th>
<th>Second Name</th>
<th>Base Office</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice</td>
<td>Alty</td>
<td>London N.</td>
</tr>
<tr>
<td>Bert</td>
<td>Brown</td>
<td>Sheffield</td>
</tr>
<tr>
<td>Cecil</td>
<td>Clarke</td>
<td>London N.</td>
</tr>
<tr>
<td>Doris</td>
<td>Davis</td>
<td>Bristol</td>
</tr>
</tbody>
</table>

Any new personal information such as their telephone number can now be added to Table 2 without any repetition.

It is also more obvious that any information which is related to both name and sales speciality, such as the total value of sales must be added to Table 1.
7.11 Third Normal Form

Consider the following table in second normal form:

<table>
<thead>
<tr>
<th>First Name</th>
<th>Second Name</th>
<th>Base Office</th>
<th>Telephone Number</th>
<th>Extension Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice</td>
<td>Alty</td>
<td>London N.</td>
<td>0181-2345678</td>
<td>246</td>
</tr>
<tr>
<td>Bert</td>
<td>Brown</td>
<td>Sheffield</td>
<td>0114-2555666</td>
<td>421</td>
</tr>
<tr>
<td>Cecil</td>
<td>Clarke</td>
<td>London N.</td>
<td>0181-2345678</td>
<td>255</td>
</tr>
<tr>
<td>Doris</td>
<td>Davis</td>
<td>Bristol</td>
<td>0117-9444333</td>
<td>246</td>
</tr>
<tr>
<td>Eric</td>
<td>Evans</td>
<td>London E.</td>
<td>0181-2345678</td>
<td>322</td>
</tr>
</tbody>
</table>

The key is the combination of first and second name.

Note that all field entries depend on the whole key....

.... but notice the telephone number is also dependent on the base office (though not vice-versa).

ie. The information giving the telephone number of each office is repeated for as many times as there are sales reps based at that office.

*This duplication of data will give all the associated problems of tables not in second normal form.*

The table needs to be put in **third normal form** which requires the table to be in second normal form and to have all non key fields dependent only on the key.

*ie. To be in third normal form each field must depend on the key, the whole key, and nothing but the key!*
7.12 Putting Tables into Third Normal Form

To put a table into third normal form the action is similar to putting a table into second normal form.

ie. Put the dependent field into a separate table with its determinant field(s) as the key.

So the table is split into two tables as follows:

<table>
<thead>
<tr>
<th>Table 1:</th>
<th>First Name</th>
<th>Second Name</th>
<th>Base Office</th>
<th>Extension Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice</td>
<td>Alty</td>
<td>London N.</td>
<td>246</td>
<td></td>
</tr>
<tr>
<td>Bert</td>
<td>Brown</td>
<td>Sheffield</td>
<td>421</td>
<td></td>
</tr>
<tr>
<td>Cecil</td>
<td>Clarke</td>
<td>London N.</td>
<td>255</td>
<td></td>
</tr>
<tr>
<td>Doris</td>
<td>Davis</td>
<td>Bristol</td>
<td>246</td>
<td></td>
</tr>
<tr>
<td>Eric</td>
<td>Evans</td>
<td>London E.</td>
<td>322</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 2:</th>
<th>Base Office</th>
<th>Telephone Number</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>London N.</td>
<td>0181-2345678</td>
</tr>
<tr>
<td></td>
<td>Sheffield</td>
<td>0114-2555666</td>
</tr>
<tr>
<td></td>
<td>Bristol</td>
<td>0117-9444333</td>
</tr>
<tr>
<td></td>
<td>London E.</td>
<td>0181-2345678</td>
</tr>
</tbody>
</table>

(Note: Thanks to the wonders of modern telephony it is quite possible for two sites that are reasonably close to share a common telephone number.)

The saving in this particular table is very small as we have only a small sample of sales reps....

    .... if there had been many sales reps at each office the saving could have been considerable.)
CHAPTER 8

Advanced Data Normalisation

Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>8.1</td>
<td>Duplication of Data in Third Normal Form</td>
<td>99</td>
</tr>
<tr>
<td>8.2</td>
<td>Third Normal Form Inconsistency</td>
<td>100</td>
</tr>
<tr>
<td>8.3</td>
<td>The Boyce-Codd Normal Form</td>
<td>101</td>
</tr>
<tr>
<td>8.4</td>
<td>Joining Tables Together</td>
<td>102</td>
</tr>
<tr>
<td>8.5</td>
<td>Methods of Joining Tables Together</td>
<td>103</td>
</tr>
<tr>
<td>8.6</td>
<td>Possibilities of Higher Normal Forms</td>
<td>104</td>
</tr>
<tr>
<td>8.7</td>
<td>Multi-Value Dependencies</td>
<td>105</td>
</tr>
<tr>
<td>8.8</td>
<td>Example of More Than One Multi-Value Dependency</td>
<td>106</td>
</tr>
<tr>
<td>8.9</td>
<td>Putting a Table Into Fourth Normal Form</td>
<td>107</td>
</tr>
<tr>
<td>8.10</td>
<td>Related Multi-Value Dependencies</td>
<td>108</td>
</tr>
<tr>
<td>8.11</td>
<td>Fifth Normal Form</td>
<td>109</td>
</tr>
<tr>
<td>8.12</td>
<td>Putting a Table Into Fifth Normal Form</td>
<td>110</td>
</tr>
<tr>
<td>8.13</td>
<td>Example of Tables in Fifth Normal Form</td>
<td>111</td>
</tr>
<tr>
<td>8.14</td>
<td>Merging Tables</td>
<td>112</td>
</tr>
<tr>
<td>8.15</td>
<td>Example of Merging Tables</td>
<td>113</td>
</tr>
<tr>
<td>8.16</td>
<td>A Final Check</td>
<td>114</td>
</tr>
</tbody>
</table>
8.1 Duplication of Data in Third Normal Form

The trouble with using a name as a key is that a name such as John Smith is often not unique.

Suppose that the sales company required each sales rep's telephone number and extension to be entered as a unique identifier in all administrative procedures.

A look-up table to be able to find out the rep's name and base office may then be required:

<table>
<thead>
<tr>
<th>Telephone Number</th>
<th>Extension Number</th>
<th>First Name</th>
<th>Second Name</th>
<th>Base Office</th>
</tr>
</thead>
<tbody>
<tr>
<td>0181-2345678</td>
<td>246</td>
<td>Alice</td>
<td>Alty</td>
<td>London N.</td>
</tr>
<tr>
<td>0114-2555666</td>
<td>421</td>
<td>Bert</td>
<td>Brown</td>
<td>Sheffield</td>
</tr>
<tr>
<td>0181-2345678</td>
<td>255</td>
<td>Cecil</td>
<td>Clarke</td>
<td>London N.</td>
</tr>
<tr>
<td>0117-9444333</td>
<td>246</td>
<td>Doris</td>
<td>Davis</td>
<td>Bristol</td>
</tr>
<tr>
<td>0181-2345678</td>
<td>322</td>
<td>Eric</td>
<td>Evans</td>
<td>London E.</td>
</tr>
<tr>
<td>0117-9444333</td>
<td>234</td>
<td>Bert</td>
<td>Brown</td>
<td>Bristol</td>
</tr>
</tbody>
</table>

With the telephone and extension numbers as the key fields this table is already in third normal form!

ie. Each non key field is dependent on the whole key
+
No non key field is dependent on any other non key field

But once again ....

The information giving the telephone number of each office is repeated for as many times as there are sales reps based at that office!
8.2 Third Normal Form Inconsistency

An alternative key exists for the given table.

Instead of: Telephone number + Extension Number
the key could be: Base office + Extension number

This gives:

<table>
<thead>
<tr>
<th>Base Office</th>
<th>Extension Number</th>
<th>First Name</th>
<th>Second Name</th>
<th>Telephone Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>London N.</td>
<td>246</td>
<td>Alice</td>
<td>Alty</td>
<td>0181-2345678</td>
</tr>
<tr>
<td>Sheffield</td>
<td>421</td>
<td>Bert</td>
<td>Brown</td>
<td>0114-2555666</td>
</tr>
<tr>
<td>London N.</td>
<td>255</td>
<td>Cecil</td>
<td>Clarke</td>
<td>0181-2345678</td>
</tr>
<tr>
<td>Bristol</td>
<td>246</td>
<td>Doris</td>
<td>Davis</td>
<td>0117-9444333</td>
</tr>
<tr>
<td>London E.</td>
<td>322</td>
<td>Eric</td>
<td>Evans</td>
<td>0181-2345678</td>
</tr>
<tr>
<td>Bristol</td>
<td>234</td>
<td>Bert</td>
<td>Brown</td>
<td>0117-9444333</td>
</tr>
</tbody>
</table>

Looked at in this form the table fails to meet the requirements for second normal form ....

.... as the telephone number is based only on part of the key, the Base office.

ie. As before, the table must be split to remove the telephone number field and place it in a new table with the base office as its key.

This inconsistency in the third normal form can only occur if:

(1) There is more than one possible key
(2) Each key is a multiple field key
(3) The possible key's each share a common field.
8.3 The Boyce-Codd Normal Form

The possible inconsistency in the third normal form was pointed out by Raymond Boyce shortly after Codd invented the first three normal forms.

Codd then refined the third normal form to specify that the rules for the third normal form should apply regardless of which key is chosen.

This new form is referred to as the Boyce-Codd normal form to distinguish it from the original third normal form.

Another way of specifying the Boyce-Codd normal form is:

*If a field is dependent on one or more other fields then those other fields must be a possible key for the table.*

The problem with tables not in Boyce-Codd normal form:

Such tables still have duplication of information with all the associated problems this brings.

The remedy:

Split the table with the same technique as putting a table into the second or third normal form.
8.4 Joining Tables Together

Splitting tables into smaller tables may be a good idea to reduce duplication of information .... .... but what if we wanted the information in it's non normalised form?

eg. The telephone extension look-up table with:

Telephone no. + extension + names + base office

has been split into:

(1) base office + extension + names
(2) base office + telephone number

But ....

The sales company still needs to find out the name of a sales rep and their base office given their telephone number and extension.

Most database management systems provide some means of joining tables back together for displaying information or generating printed reports.

ie. It is still possible to print out the following even though the data has been normalised into two tables:

<table>
<thead>
<tr>
<th>Telephone Number</th>
<th>Extension Number</th>
<th>First Name</th>
<th>Second Name</th>
<th>Base Office</th>
</tr>
</thead>
<tbody>
<tr>
<td>0181-2345678</td>
<td>246</td>
<td>Alice</td>
<td>Alty</td>
<td>London N.</td>
</tr>
<tr>
<td>0114-2555666</td>
<td>421</td>
<td>Bert</td>
<td>Brown</td>
<td>Sheffield</td>
</tr>
</tbody>
</table>

etc...
8.5 Methods of Joining Tables Together

In the example, if the two tables were known as Name and Tel the standard SQL database query language command for displaying the original table could be:

```
SELECT Tel.TelNum, Name.ExtNum,
    Name.FirstName, Name.SecondName,
    Name.BaseOffice
FROM Name, Tel
WHERE Name.BaseOffice = Tel.BaseOffice;
```

The `WHERE` part of this command informs the database management system how to join the two tables at the time they are displayed.

Database management systems that do not use SQL commands may require the join condition to be specified before the display command is given.

In either case the lines of output are worked out when the display is generated....

.... the joined table is not actually created and stored anywhere.

Many systems allow the join mechanism to be permanently set up by creating a virtual table of the joined tables known as a "view".

Although it can be examined like a real table, in reality a view selects the information from the underlying base tables whenever it is used.
8.6 Possibilities of Higher Normal Forms

Suppose the sales reps are allocated to specific customers as well as having their sales speciality.

A table showing who sells what to which customer could be as follows:

<table>
<thead>
<tr>
<th>First Name</th>
<th>Second Name</th>
<th>Sales Speciality</th>
<th>Customer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice</td>
<td>Alty</td>
<td>computers</td>
<td>Hackett Software Co.</td>
</tr>
<tr>
<td>Alice</td>
<td>Alty</td>
<td>printers</td>
<td>Hackett Software Co.</td>
</tr>
<tr>
<td>Alice</td>
<td>Alty</td>
<td>netware</td>
<td>Hackett Software Co.</td>
</tr>
<tr>
<td>Alice</td>
<td>Alty</td>
<td>computers</td>
<td>Pi-Rates Ltd.</td>
</tr>
<tr>
<td>Alice</td>
<td>Alty</td>
<td>printers</td>
<td>Pi-Rates Ltd.</td>
</tr>
<tr>
<td>Alice</td>
<td>Alty</td>
<td>netware</td>
<td>Pi-Rates Ltd.</td>
</tr>
<tr>
<td>Bert</td>
<td>Brown</td>
<td>software</td>
<td>Hackett Software Co.</td>
</tr>
<tr>
<td>Bert</td>
<td>Brown</td>
<td>software</td>
<td>Flog-It-Quick Retailers</td>
</tr>
<tr>
<td>Bert</td>
<td>Brown</td>
<td>software</td>
<td>C.D.Finance Plc.</td>
</tr>
<tr>
<td>Cecil</td>
<td>Clarke</td>
<td>computers</td>
<td>Flog-It-Quick Retailers</td>
</tr>
<tr>
<td>Cecil</td>
<td>Clarke</td>
<td>printers</td>
<td>Flog-It-Quick Retailers</td>
</tr>
<tr>
<td>Cecil</td>
<td>Clarke</td>
<td>software</td>
<td>Flog-It-Quick Retailers</td>
</tr>
<tr>
<td>Cecil</td>
<td>Clarke</td>
<td>computers</td>
<td>Pi-Rates Ltd.</td>
</tr>
<tr>
<td>Cecil</td>
<td>Clarke</td>
<td>printers</td>
<td>Pi-Rates Ltd.</td>
</tr>
<tr>
<td>Cecil</td>
<td>Clarke</td>
<td>software</td>
<td>Pi-Rates Ltd.</td>
</tr>
</tbody>
</table>

There is no choice of key as all four columns are required for the key, so the table is automatically in Boyce-Codd normal form as well as third normal form.

There appears to be still a lot of duplication....

.... but can this duplication be avoided?
8.7 **Multi-Value Dependencies**

A multi-value dependency occurs when a field would contain more than one value if we went backwards and took the table out of first normal form.

eg. In our original table of the sales team:

<table>
<thead>
<tr>
<th>First</th>
<th>Second Name</th>
<th>Sales Speciality</th>
<th>Base Office</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice</td>
<td>Alty</td>
<td>computers, printers, netware</td>
<td>London N.</td>
</tr>
<tr>
<td>Bert</td>
<td>Brown</td>
<td>software</td>
<td>Sheffield</td>
</tr>
<tr>
<td>Cecil</td>
<td>Clarke</td>
<td>computers, printers, software</td>
<td>London N.</td>
</tr>
<tr>
<td>Doris</td>
<td>Davis</td>
<td>software, netware</td>
<td>Bristol</td>
</tr>
</tbody>
</table>

The sales speciality is a multi-valued dependency of the combined first and second names.

Further reduction of duplication of information by putting the table in normal forms higher than the Boyce-Codd normal form is possible....

**.... if there is more than one multi-value dependency.**

This in turn can only occur for tables in Boyce-Codd normal form if:

(1) there are at least three columns in the table
and (2) all the fields are required to make up the key
8.8 **Example of More Than One Multi-Value Dependency**

The table showing who sells what to which customer has two multiple dependencies:

<table>
<thead>
<tr>
<th>First Name</th>
<th>Second Name</th>
<th>Sales Speciality</th>
<th>Customer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice</td>
<td>Alty</td>
<td>computers, printers, netware</td>
<td>Hackett Software Co., Pi-Rates Ltd.</td>
</tr>
<tr>
<td>Cecil</td>
<td>Clarke</td>
<td>computers, printers, software</td>
<td>Flog-It-Quick Retailers, Pi-Rates Ltd.</td>
</tr>
</tbody>
</table>

These multi-value fields are completely independent of each other.

ie. When the table is put in first normal form there is a row for every possible combination of sales speciality and customer for each sales rep.

This independence means the table needs to be put into *fourth normal form*. 
8.9 **Putting a Table Into Fourth Normal Form**

To put a table into fourth normal form the table needs to be split so that there is no more than one multi-value dependency in each new table.

So the table showing who sells what to which customer becomes:

<table>
<thead>
<tr>
<th>Table 1:</th>
<th>First Name</th>
<th>Second Name</th>
<th>Sales Speciality</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice</td>
<td>Alty</td>
<td>computers</td>
<td></td>
</tr>
<tr>
<td>Alice</td>
<td>Alty</td>
<td>printers</td>
<td></td>
</tr>
<tr>
<td>Alice</td>
<td>Alty</td>
<td>netware</td>
<td></td>
</tr>
<tr>
<td>Bert</td>
<td>Brown</td>
<td>software</td>
<td></td>
</tr>
<tr>
<td>Cecil</td>
<td>Clarke</td>
<td>computers</td>
<td></td>
</tr>
<tr>
<td>Cecil</td>
<td>Clarke</td>
<td>printers</td>
<td></td>
</tr>
<tr>
<td>Cecil</td>
<td>Clarke</td>
<td>software</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 2:</th>
<th>First Name</th>
<th>Second Name</th>
<th>Customer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice</td>
<td>Alty</td>
<td>Hackett Software Co.</td>
<td></td>
</tr>
<tr>
<td>Alice</td>
<td>Alty</td>
<td>Pi-Rates Ltd.</td>
<td></td>
</tr>
<tr>
<td>Bert</td>
<td>Brown</td>
<td>Hackett Software Co.</td>
<td></td>
</tr>
<tr>
<td>Bert</td>
<td>Brown</td>
<td>Flog-It-Quick Retailers</td>
<td></td>
</tr>
<tr>
<td>Bert</td>
<td>Brown</td>
<td>C.D.Finance Plc.</td>
<td></td>
</tr>
<tr>
<td>Cecil</td>
<td>Clarke</td>
<td>Flog-It-Quick Retailers</td>
<td></td>
</tr>
<tr>
<td>Cecil</td>
<td>Clarke</td>
<td>Pi-Rates Ltd.</td>
<td></td>
</tr>
</tbody>
</table>

As well as a potential saving of storage space it will be now far easier to add new entries to these tables or to make changes to any speciality or customer.
### 8.10 Related Multi-Value Dependencies

Suppose a table has more than one multi-value dependency but they are not totally independent of each other:

<table>
<thead>
<tr>
<th>First Name</th>
<th>Second Name</th>
<th>Sales Speciality</th>
<th>Customer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice</td>
<td>Alty</td>
<td>computers</td>
<td>Hackett Software Co.</td>
</tr>
<tr>
<td>Alice</td>
<td>Alty</td>
<td>printers</td>
<td>Hackett Software Co.</td>
</tr>
<tr>
<td>Alice</td>
<td>Alty</td>
<td>printers</td>
<td>Pi-Rates Ltd.</td>
</tr>
<tr>
<td>Alice</td>
<td>Alty</td>
<td>netware</td>
<td>Pi-Rates Ltd.</td>
</tr>
<tr>
<td>Bert</td>
<td>Brown</td>
<td>software</td>
<td>Flog-It-Quick Retailers</td>
</tr>
<tr>
<td>Bert</td>
<td>Brown</td>
<td>software</td>
<td>C.D.Finance Plc.</td>
</tr>
<tr>
<td>Cecil</td>
<td>Clarke</td>
<td>computers</td>
<td>Flog-It-Quick Retailers</td>
</tr>
<tr>
<td>Cecil</td>
<td>Clarke</td>
<td>printers</td>
<td>Flog-It-Quick Retailers</td>
</tr>
<tr>
<td>Cecil</td>
<td>Clarke</td>
<td>software</td>
<td>Flog-It-Quick Retailers</td>
</tr>
<tr>
<td>Cecil</td>
<td>Clarke</td>
<td>printers</td>
<td>Pi-Rates Ltd.</td>
</tr>
<tr>
<td>Cecil</td>
<td>Clarke</td>
<td>software</td>
<td>Pi-Rates Ltd.</td>
</tr>
</tbody>
</table>

Every combination of sales speciality with customer does not necessarily exist for each sales rep.

ie.  
- Alice Alty does not sell netware to Hackett Software.
- Neither Alice Alty or Cecil Clarke sells computers to Pi-Rates Ltd.

Splitting the table into two as described for fourth normal form would lose this information.
8.11 **Fifth Normal Form**

If:

(1) a table has at least 3 columns

(2) all the fields are required to make up the key

(3) there are two or more multi-valued dependencies

(4) *there is a definable relationship between these multi-valued dependencies*

(5) *this relationship is independent of any other field*

.... then duplication of information can be reduced by putting the table in **fifth normal form**.

In the example given the definable relationship is:

1. Hackett Software does not buy netware from anybody

2. Pi-Rates Ltd. does not buy computers from anybody

3. All other customers will buy any of the products from any sales rep.

This relationship can be expressed in a table defining which customer buys which product specialities.
8.12 **Putting a Table Into Fifth Normal Form**

To put the table of who sells what to which company into fifth normal form three tables need to be created:

1. The table of who sells what
2. The table of who sells to which customer
3. The table of which customer buys what

The first two of these tables are the same as described for putting into fourth normal form, but now the extra table is necessary.

The relationship between the sales speciality and the customers is difficult to see from the original table.

This means on the original table it would be difficult to add a new sales speciality-customer relationship such as if Flog-It-Quick Retailers decided to buy printers....

.... but with the new table it is simple.

But ....

*the difficulty in seeing the relationship also means it can be difficult to detect that the tables need to be put into fifth normal form.*
### 8.13 Example of Tables in Fifth Normal Form

**Table 1:**

<table>
<thead>
<tr>
<th>First Name</th>
<th>Second Name</th>
<th>Sales Speciality</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice</td>
<td>Alty</td>
<td>computers</td>
</tr>
<tr>
<td>Alice</td>
<td>Alty</td>
<td>printers</td>
</tr>
<tr>
<td>Alice</td>
<td>Alty</td>
<td>netware</td>
</tr>
<tr>
<td>Bert</td>
<td>Brown</td>
<td>software</td>
</tr>
<tr>
<td>Cecil</td>
<td>Clarke</td>
<td>computers</td>
</tr>
<tr>
<td>Cecil</td>
<td>Clarke</td>
<td>printers</td>
</tr>
<tr>
<td>Cecil</td>
<td>Clarke</td>
<td>software</td>
</tr>
</tbody>
</table>

**Table 2:**

<table>
<thead>
<tr>
<th>First Name</th>
<th>Second Name</th>
<th>Customer Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice</td>
<td>Alty</td>
<td>Hackett Software Co.</td>
</tr>
<tr>
<td>Alice</td>
<td>Alty</td>
<td>Pi-Rates Ltd.</td>
</tr>
<tr>
<td>Bert</td>
<td>Brown</td>
<td>Hackett Software Co.</td>
</tr>
<tr>
<td>Bert</td>
<td>Brown</td>
<td>Flog-It-Quick Retailers</td>
</tr>
<tr>
<td>Bert</td>
<td>Brown</td>
<td>C.D.Finance Plc.</td>
</tr>
<tr>
<td>Cecil</td>
<td>Clarke</td>
<td>Flog-It-Quick Retailers</td>
</tr>
<tr>
<td>Cecil</td>
<td>Clarke</td>
<td>Pi-Rates Ltd.</td>
</tr>
</tbody>
</table>

**Table 3:**

<table>
<thead>
<tr>
<th>Sales Speciality</th>
<th>Customer Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>computers</td>
<td>Hackett Software Co.</td>
</tr>
<tr>
<td>printers</td>
<td>Hackett Software Co.</td>
</tr>
<tr>
<td>software</td>
<td>Hackett Software Co.</td>
</tr>
<tr>
<td>software software</td>
<td>C.D.Finance Plc.</td>
</tr>
<tr>
<td>computers printers</td>
<td>Flog-It-Quick Retailers</td>
</tr>
<tr>
<td>printers software</td>
<td>Flog-It-Quick Retailers</td>
</tr>
<tr>
<td>printers software</td>
<td>Pi-Rates Ltd.</td>
</tr>
<tr>
<td>software</td>
<td>Pi-Rates Ltd.</td>
</tr>
<tr>
<td>netware</td>
<td>Pi-Rates Ltd.</td>
</tr>
</tbody>
</table>
8.14 Merging Tables

Normalisation tends to fragment larger tables into many smaller tables ....

.... sometimes these smaller tables should then be put together to make larger tables again.

If the tables have the same key fields

and the table entries cover the same domain

(ie. each row in one table has a corresponding row in the other)

then the tables can be merged.

Notes:

1. Tables with the same key should only occur if they are derived from different larger tables.

2. If the domains of the two tables are not the same then some null fields will result in any merged table.

3. Strictly speaking different domains mean the tables should not be merged...

....but it may still be convenient to do so if they are nearly the same as there will only be a few null entries.
8.15 Example of Merging Tables

Suppose there is a table of sales rep information:

<table>
<thead>
<tr>
<th>ID No.</th>
<th>First Name</th>
<th>Second Name</th>
<th>Base Office</th>
</tr>
</thead>
<tbody>
<tr>
<td>1010</td>
<td>Alice</td>
<td>Alty</td>
<td>London N.</td>
</tr>
<tr>
<td>1623</td>
<td>Bert</td>
<td>Brown</td>
<td>Sheffield</td>
</tr>
<tr>
<td>1278</td>
<td>Cecil</td>
<td>Clarke</td>
<td>London N.</td>
</tr>
<tr>
<td>etc...</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

and also a table of the commission earned:

<table>
<thead>
<tr>
<th>ID No.</th>
<th>Commission</th>
</tr>
</thead>
<tbody>
<tr>
<td>1010</td>
<td>£2500</td>
</tr>
<tr>
<td>1623</td>
<td>£3500</td>
</tr>
<tr>
<td>1278</td>
<td>£1750</td>
</tr>
<tr>
<td>etc...</td>
<td></td>
</tr>
</tbody>
</table>

If this second table holds no record for anyone other than Sales Reps

and ALL Sales Reps earn commission

then these can be combined into:

<table>
<thead>
<tr>
<th>ID No.</th>
<th>First Name</th>
<th>Second Name</th>
<th>Base Office</th>
<th>Commission</th>
</tr>
</thead>
<tbody>
<tr>
<td>1010</td>
<td>Alice</td>
<td>Alty</td>
<td>London N.</td>
<td>£2500</td>
</tr>
<tr>
<td>1623</td>
<td>Bert</td>
<td>Brown</td>
<td>Sheffield</td>
<td>£3500</td>
</tr>
<tr>
<td>1278</td>
<td>Cecil</td>
<td>Clarke</td>
<td>London N.</td>
<td>£1750</td>
</tr>
<tr>
<td>etc...</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
8.16 A Final Check

Having put all tables into 5th Normal Form, including any merged tables, the normalisation process is complete ....

.... however, it is always worth making a common sense check that the data tables are in their optimum format:

• Are there any tables containing separate rows for a sequence of entities or time slots? If so, could this be better represented by columns for the start and finish of a range? (See sections 5.14 and 5.15)

Normalisation will not produce this optimisation.

• Are there any totals or other calculated fields stored? Eg. A parent table could have the number of children as an attribute, with the children listed separately in another table.

This is a form of duplication of data which is unlikely to be detected through normalisation.

• Can any data be derived from data elsewhere? Eg. A table may show which supervisors are on duty on each day of the week, but elsewhere there may be a table specifying which days of the week a supervisor is not working.

This is another form of duplication which is unlikely to be detected through normalisation.

This final, common sense check is the last step in the design of the data table structure by normalisation.
# CHAPTER 9

## Normalisation and Entity Relationship Modelling

<table>
<thead>
<tr>
<th>Contents</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>9.1 Objectives Compared</td>
<td>116</td>
</tr>
<tr>
<td>9.2 Identifying the Entities and Attributes</td>
<td>117</td>
</tr>
<tr>
<td>9.3 Where Do Multiple Field Keys come From?</td>
<td>121</td>
</tr>
<tr>
<td>9.4 E-R Modelling and the 2nd, 3rd and Boyce-Codd Normal Forms</td>
<td>122</td>
</tr>
<tr>
<td>9.5 Tables With 3 or More Fields in Their Keys</td>
<td>123</td>
</tr>
<tr>
<td>9.6 Three Way Relationship Tables</td>
<td>124</td>
</tr>
<tr>
<td>9.7 Normalisation of 3-Way Relationship Tables</td>
<td>125</td>
</tr>
<tr>
<td>9.8 Advantages of Entity Relationship Analysis and Normalisation Used Together</td>
<td>126</td>
</tr>
</tbody>
</table>
9.1 Objectives Compared

The objectives of E-R Modelling are:

1. To help develop an understanding of the nature of the data complexities in the system.

2. To be able to organise the data into a logical, "structured" form with each data item where it may be expected (useful for maintenance).

3. To be able to organise the data in an efficient way so that data is neither duplicated nor omitted.

The end result is (hopefully) the "system data dictionary" which defines:

1. The data tables that appear in the system.

2. The fields (ie. columns) for each entry in the table.

3. A key for each table, consisting of one or more of these fields.

This is similar to the Normalisation objectives so how are Normalisation and E-R Modelling related?
9.2 **Identifying the Entities and Attributes**

The first step was to identify the entities and the attributes for those entities.

It may not be clear whether an item is an entity or an attribute so the following hints were made:

**E-R Hint : Variable Numbers of an Attribute**

Check the attributes of all entities -

_Are there variable numbers of an attribute?_

Eg. An item may have several different colours.

If so then make the attribute a separate entity or there will be blank fields in the data base ....

..... this can give unexpected entities such as Colour.

ie. If an item can have several colours,

and different items have different numbers of colours

then the colour must be a separate entity.

but If all the items had multiple colours but each had the same number of colours there would be no need of a separate colour entity.

*This process is exactly the same as the first step of normalisation .....*

ie. it is putting the entity table into 1st normal form
E-R Hint: Attributes With Attributes

Check the attributes of all entities -

*Has any attribute got an attribute of its own?*

Eg. A person entity may have a car type and a car length as attributes .... 

.... in this case the car length is really an attribute of the car type.

If an attribute has an attribute of its own then it should be made into a separate entity .... 

.... otherwise the attributes are interdependent and there is a risk that the data base could become inconsistent.

Eg. A persons car type may be changed with the user forgetting to change the car length .... 

.... this could lead to two people with the same car type but with different car lengths.

*This process is exactly the same as the third step of normalisation* ..... 

*ie. it is putting the entity table into 3rd normal form, or 2nd normal form if an attribute depends on an "attribute" that is really part of the key.*
E-R Hint : Repeated, Long Text Attributes

Check the attributes of all entities -

Is any long text attribute likely to be repeated for different occurrences of the entity?

eg. More than one person may have the same car type, such as "Ford Mondeo Estate".

ie. If there are many repeats
   or the text is long
then there can be a significant space saving by making this attribute into a separate entity referenced by an ID number.

ie. A separate table should be created with the text and an associated identity number . . .
   . . . other tables will then refer to the ID number rather than the text itself.

This process is also the same as the third step of normalisation.....

ie. Once an ID number has been created the text field is dependent on that number, so this process is putting the entity table back into 3rd normal form.
E-R Hint : Text Attributes Used In Matchings

Check the attributes of all entities -

*Is any text attribute likely to be tested for matching entries?*

Even if the text is short with only a few repeats . . .

if tests may be made for matching entries,
then it should be a separate entity.

Eg. A **TOWN** attribute could have any of the entries:

Leicester Leics Leic Leics. Leic.
LEICESTER LEICS LEIC LEICS. LEIC.

This is OK if it is only used as part of an address. . .

**But ....** if matching on the name is to be done,  
(eg. to list all people from Leicester)  
then entries may be missed!

So .... If the field is made a separate entity then each name would be stored once only with only one representation.

*Once again process is the same as the third step of normalisation .....*

*ie. Once an ID number has been created the text field is dependent on that number, so this process is putting the entity table back into 3rd normal form.*
9.3 Where Do Multiple Field Keys come From?

1. Entities where the entity identifier requires two or more fields:
   eg. First name and second name
   Date and time
   Telephone number and extension

2. Relationships between entities.
   Relationships that are classified as:
   N to N (many to many)
   1 to N which is optional for the 'N' entity
   1 to 1 which is optional for both entities
   All (in theory) produce a separate relation table with the key being a combination
   of the key fields from the two entity tables.
   (In practice a separate relation table is not required for the 1 to N or 1 to 1
   relationships if blank fields are allowed in the entity tables).

3. Discovered Entity Tables
   These are similar to relationship tables except there may be further, non key
   fields for:
   (1) additional attributes.
   (2) additional 1 to 1 or N to 1 relationships that can be modelled by adding
   the other entity key as a field in the discovered entity table.
9.4 **E-R Modelling and the 2nd, 3rd and Boyce-Codd Normal Forms**

Data tables derived from entity relationship modelling should automatically be in 2nd Normal Form because:

- for entity tables with multiple field keys correct choice of entities and attributes should put the entity table in both 2nd and 3rd normal form.

- for relationship tables there are no non key fields.

- for discovered entities any attributes of the discovered entity must depend on the whole key....
  .... otherwise they will depend on the key for one of the original entities it is discovered between.

  ie. They should be in a different entity's table.

The 1st and 3rd normal form will also be satisfied by correctly choosing the entities and attributes ....

.... as will the Boyce-Codd normal form if enough care is taken over attribute dependencies ....

.... so a good E-R analysis will automatically give tables in Boyce-Codd Normal Form.
9.5 **Tables With 3 or More Fields in Their Keys**

Tables normalised to 3rd/Boyce-Codd normal form will not require further normalisation unless:

1. There are at least 3 fields in the key.
2. There are no non-key fields.

This will only occur if:

1. A single entity needs three fields for the key
   
   Eg. Day + Month + Year  
   Exchange no. + telephone no. + extension  
   These tables will normally be in 5th normal form.

2. A relationship table relates two entities, one of which has a two field key.  
   These tables will normally be in 5th normal form.

3. A relationship table relates 3 different entities.  
   *These tables may require further normalisation.*

4. An entity is discovered between two other entities, one of which has a two field key.  
   These tables will normally be in 5th normal form.

5. An entity is discovered between 3 other entities.  
   If there are no other attributes or relationships these tables resemble three way relationship tables *and may require further normalisation.*
9.6 Three Way Relationship Tables

3-way relationships are created when it is found that other links do not supply all the desired information:

Alternatively an entity can be "discovered" between two entities which then leads to relationships being remodelled to link to the new entity:

It is also possible to derive the 3-way discovered entity directly, but this is less intuitive.

However, if no further attributes or relationships are found for the new entity it will give the same table as the 3-way relationship.
9.7 Normalisation of 3-Way Relationship Tables

Normalisation to 4th or 5th normal forms has the opposite effect of creating a three way relationship.

ie. The single 3-way relationship table is broken down into 2-way relationship tables.

Normalising to 4th normal form is equivalent to converting:

\[\text{Original 3-way relationship table} \rightarrow \text{2 relationship tables} \]

.... which then results in two relationship tables.

Normalising to 5th normal form is equivalent to converting:

\[\text{Original 3-way relationship table} \rightarrow \text{3 relationship tables} \]

.... which then results in three relationship tables.

*If either of the above is possible it shows the original creation of the 3-way relationship was a mistake!*
9.8 Advantages of Entity Relationship Analysis and Normalisation Used Together

Entity Relationship Analysis has the advantages:

1. It gives a means of starting a data structure design from scratch.
2. It is diagrammatic - this makes it easier to understand which in turn:
   (1) Helps to clarify the mind of the user
   (2) Helps to communicate the ideas to others for checking

Normalisation helps to:

1. Check the database structure design is correct
2. Correct any database structure design faults
3. Clarify difficult areas of E-R data structure design such as 3-way relationship tables.

Together they give a data table structure where:

1. Data will not be duplicated or omitted.
2. Empty data fields are avoided.
3. It is easy to view, add, delete, or update data.
CHAPTER 10

Data Normalisation Case Study

Contents

10.1 Normalisation Example: A Bus Company 128
10.2 The Bus Company Tables 129
10.3 Restructuring The Driver Table into First and Second Normal Form 131
10.4 Restructuring The Driver Table into Third and Final Normal Form 132
10.5 Restructuring The Permission Table into First, Second and Third Normal Form 133
10.6 Restructuring The Permission Table into Fourth and Fifth Normal Form 134
10.7 Results From The Driver and Permission Tables 135
10.8 Restructuring The Log Table Into First, Second and Third Normal Forms 136
10.9 Restructuring The Log Table Into Boyce-Codd Normal Form 137
10.10 Restructuring the Service Table Into First, Second and Third Normal Form 138
10.11 End Result After Normalisation 139
10.1 Normalisation Example: A Bus Company

A bus company has a large fleet of busses which it runs over an extensive network of bus routes. It believes it gives a high level of service to its customers and it also takes pride in the training it gives its many drivers.

Each driver must have received a training session on each make and model of bus before he or she is allowed to drive that bus type. In addition, no driver can drive any particular route without being trained for that route.

In the interests of service a log is kept of which driver and which bus is used for every bus run so that any complaints can be traced to the person and vehicle responsible. A driver is only ever assigned to one route and one bus in a day, though the bus may be re-used on another route on a different shift.

Each bus is kept in good working order with a record being kept of when it was last serviced and how often a bus of that make and model requires a service. Most busses can be used on most routes but there are some restrictions about where the larger busses may go, and the smaller ones are not used on high demand routes.

The bus company's records are kept on computer using a modern database management system with the information organised and stored in four tables.

*How can the data structure be improved?*
10.2 **The Bus Company Tables**

1. **The Driver Table**
   This gives data about each driver in the following fields:
   - **Driver ID**: A unique number for each driver
   - **Title**: Mr./Mrs./Miss
   - **Name**: Driver's name
   - **Training**: The bus types (ie. makes and models) of each bus he/she is trained to drive
   - **D.O.B.**: Date of birth
   - **Grade**: The driver's job grade
   - **Salary**: The salary for drivers at this grade

2. **The Permission Table**
   This records all possible combinations of which driver is allowed to take which type of bus on which route, and has the following fields:
   - **Driver ID**: The unique ID number for each driver.
   - **Route**: The route identification number
   - **Bus Type**: The bus make and model
   - **Capacity**: The number of passengers a bus of this type can carry
3. The Log Table
Records for each bus trip (specified by the day, time and route) the bus number, the driver and other associated information in the following fields:

- **Day**: The date
- **Time**: The time of the start of a bus trip
- **Route**: The route identification number
- **Direction**: Bus direction= "outward" or "inward"
- **Driver ID**: The unique ID number for each driver
- **Bus Reg**: The registration number of the bus
- **Bus Type**: The bus make and model
- **Freq.**: The number of busses per day on the route
- **Duration**: The average length of time that a bus on this route is expected to take

4. The Service Table
Records details of the busses servicing in the following fields:

- **Bus Reg**: The registration number of the bus
- **Last Serv**: The date of the last service
- **Interval**: The prescribed number of months between services for this type of bus
10.3 Restructuring The Driver Table into First and Second Normal Form

First Normal Form

This table is not even in first normal form.

The *Training* field contains multiple entries of the type referred to in other tables as *Bus Type*.

Each row should be repeated for as many entries there are in the *Training* field so there is only one entry in this field in each row.

The *Training* field can then be renamed *Bus Type* to be consistent with the other tables.

Second Normal Form

The new key for the table is *Driver ID + Bus Type*. All other fields are dependent on the *Driver ID* only.

To put into second normal form the table should then be split into:

1. A **Bus Training Table** with fields:
   
   *Driver ID + Bus Type*
   
   This table cannot be further normalised.

2. A new **Driver Table** with fields:
   
   *Driver ID* -> *Title, Name, D.O.B., Grade, Salary*

   Notation = *key field + key field +... -> non key field, non key field...*
10.4 Restructuring The Driver Table into Third and Final Normal Form

Third Normal Form

In the modified Driver Table the Salary is dependent on the Grade. This should be split from the rest of the table to give:

1. A Salary Table with fields:
   
   Grade -> Salary
   
   This table cannot be further normalised.

2. A new Driver Table with fields:
   
   Driver ID -> Title, Name, D.O.B., Grade
   
   This table cannot be further normalised.

Result

The original Driver Table becomes:

1. Bus Training Table: Driver ID + Bus Type

2. Salary Table: Grade -> Salary

3. Driver Table: Driver ID -> Title, Name, D.O.B., Grade

Notation = key field + key field +... -> non key field, non key field...
10.5 **Restructuring The Permission Table into First, Second and Third Normal Form**

**First Normal Form**

The table is already in first normal form.

**Second Normal Form**

The key to this table is \textit{Driver ID} + \textit{Route} + \textit{Bus Type}.

However, Capacity depends only on \textit{Bus Type}.

The table should therefore be split to give:

1. A new **Permission Table** with fields:
   \textit{Driver ID} + \textit{Route} + \textit{Bus Type}

2. A **Bus Type Table** with fields:
   \textit{Bus Type} -> \textit{Capacity}
   
   This table cannot be further normalised.

**Third Normal Form, Boyce-Codd Normal Form**

The Permission Table is already in third normal form and Boyce-Codd normal form.
10.6 Restructuring The Permission Table into Fourth and Fifth Normal Form

Fourth Normal Form

There is two multi-value dependencies in the Permission table:

1. the routes a driver is trained to drive
2. the bus types the driver is trained to drive.

The two multi-value dependencies are not totally independent of each other, however, as not all bus types are used on all routes.

The table, therefore, already counts as being in fourth normal form.

Fifth Normal Form

A relationship between the multi-valued dependencies of the Permission Table can be simply established by introducing a table showing what bus types are used on each route.

Thus the Permission table becomes:

1. A **Bus Training Table** with fields:
   \[Driver ID + Bus Type\]

2. A **Route Training Table** with fields:
   \[Driver ID + Route\]

3. A **Bus Use Table** with fields:
   \[Bus Type + Route\]
10.7 **Results From The Driver and Permission Tables**

The Bus Training Table has been derived from both the original Driver Table and Original Permission Table....

These tables should correspond, if not, a data entry error has occurred!

Therefore, the resulting tables so far are:

1. **Bus Training Table**: $Driver \ ID + Bus \ Type$
2. **Salary Table**: $Grade \rightarrow Salary$
3. **Driver Table**: $Driver \ ID \rightarrow Title, \ Name, \ D.O.B., \ Grade$
4. **Bus Type Table**: $Bus \ Type \rightarrow Capacity$
5. **Route Training Table**: $Driver \ ID + Route$
6. **Bus Use Table**: $Bus \ Type + Route$
10.8 **Restructuring The Log Table Into First, Second and Third Normal Forms**

The table is already in first normal form.

The key to the table is \(\text{Day}+\text{Time}+\text{Route}+\text{Direction}\), but \(\text{Freq.}\) and \(\text{Duration}\) are only dependent on the \(\text{Route}\) field.

ie. To put into second normal form these fields should be split from the Log Table into a table with route information.

This still leaves the Log Table with \(\text{Bus Type}\) dependent on \(\text{Bus Reg.}\)

ie. To put into third normal form the Log Table should be further split to put these fields into a table with information about each bus.

Together this gives

1. **A Route Table** with fields:
   \(\text{Route} \rightarrow \text{Freq.}, \text{Duration}\)
   This table cannot be further normalised.

2. **A Bus Table** with fields:
   \(\text{Bus Reg} \rightarrow \text{Bus Type}\)
   This table cannot be further normalised.

3. **A new Log Table** with fields:
   \(\text{Day}+\text{Time}+\text{Route}+\text{Direction} \rightarrow \text{Driver ID}, \text{Bus Reg}\)
10.9 Restructuring The Log Table Into Boyce-Codd Normal Form

The Log Table has three possible keys:

either:  \[ Day + Time + Route + Direction \]
or:  \[ Day + Time + Driver ID \]
or:  \[ Day + Time + Bus \]

If the second key is chosen then \( Route \) is dependent on \( Day+Driver ID \) only, not on \( Time \).

The Log Table should, therefore, be further split into:

1. A **Driver Allocation Table** with fields:
   \[ Day + Driver ID -> Route, Bus \]
   This table cannot be further normalised.

2. A **Driver Log Table** with fields:
   \[ Day + Time + Driver ID -> Direction \]
   This table cannot be further normalised.

*This is not a very obvious division of tables!*

Can the old Log Table be re-created to allow the driver or bus to be identified given a day, time, route and direction?

Yes - by joining the Driver Allocation Table and Driver Log Table by matching the *Driver ID* and *Day* fields .... a good use for a view!
10.10  **Restructuring the Service Table Into First, Second and Third Normal Form**

This table is already in first, second and third normal forms, with the key to this table being Bus Reg.

This has the same key and domain as the Bus Table derived from the original Log Table, so these tables should be merged to give a new Bus Table with fields:

\[
Bus \text{ Reg} \rightarrow Bus \text{ Type, Last Serv, Interval}
\]

But this table is not now in third normal form as \textit{Interval} is dependent on \textit{Bus Type}!

To put into third normal form these fields should be separated to give a table with the service table dependent on the bus type....

.... but this is the same key and domain as the Bus Type Table derived from the Permission Table, so these tables should then also be merged to give:

1. A new **Bus Table** with fields:
   \[
   Bus \text{ Reg} \rightarrow Bus \text{ Type, Last Serv}
   \]
   This table cannot be further normalised.

2. A new **Bus Type Table** with fields:
   \[
   Bus \text{ Type} \rightarrow Capacity, Interval
   \]
   This table cannot be further normalised.
10.11 **End Result After Normalisation**

From the original four tables we now have:

1. **Bus Training Table**: *Driver ID + Bus Type*
2. **Salary Table**: *Grade -> Salary*
3. **Driver Table**: *Driver ID -> Title, Name, D.O.B., Grade*
4. **Bus Type Table**: *Bus Type -> Capacity, Interval*
5. **Route Training Table**: *Driver ID + Route*
6. **Bus Use Table**: *Bus Type + Route*
7. **Route Table**: *Route -> Freq., Duration*
8. **Driver Allocation Table**:
   
   *Day + Driver ID -> Route, Bus Reg*
9. **Driver Log Table**:

   *Day + Time + Driver ID -> Direction*
10. **Bus Table**: *Bus Reg -> Bus Type, Last Serv*

Although it has more tables this new structure will:

- take up less storage space,
- be easier to change and keep up to date,
- be less likely to contain errors!
# CHAPTER 11

## Introduction To 'SQL'

<table>
<thead>
<tr>
<th>Contents</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>11.1  Introduction To 'SQL'</td>
<td>141</td>
</tr>
<tr>
<td>11.2  Where is SQL Used?</td>
<td>142</td>
</tr>
<tr>
<td>11.3  What is a Database?</td>
<td>143</td>
</tr>
<tr>
<td>11.4  Clients and Servers</td>
<td>144</td>
</tr>
<tr>
<td>11.5  SQL Language Syntax : Names</td>
<td>145</td>
</tr>
<tr>
<td>11.6  SQL Language Syntax : Commands</td>
<td>146</td>
</tr>
<tr>
<td>11.7  SQL Language Syntax : Constants</td>
<td>146</td>
</tr>
<tr>
<td>11.8  Example Tables</td>
<td>147</td>
</tr>
<tr>
<td>11.9  The TABLE Command</td>
<td>148</td>
</tr>
</tbody>
</table>
11.1 Introduction To 'SQL'

What is SQL?

- SQL is the **Structured Query Language** ....
  .... a very high level language to handle the data this language is known as a
  'Fourth Generation Language' or 4GL.

- It was originally developed by IBM for its DB2 database management system ....
  .... but later it became the ISO and ANSI standard language for database handling.

- There is more than one version of the SQL standard. This text describes the latest standard known as either SQL2 or SQL/92.

- This text does NOT describe every command and detail of the SQL/92 standard - this would be too complex for the space available, but it does describe all the commonly used commands.

- This text also describes some common additions to the SQL standard provided by widely used database management systems such as Access and Oracle.
11.2 Where is SQL Used?

- SQL the standard database query language used on most popular 'Database Management Systems' or DBS.
  
  Example of DBMSs using SQL are:
  Access, Microsoft SQL Server, Oracle, Sybase, MySQL.

- A DBMS is a program or series of programs to create, modify and output data within a data base.

- A DBMS that handles databases in accordance with the rules for 'relational' data bases is called a 'Relational Data Base Management System' or RDBMS.

- A DBMS that has facilities to build application programs, tailored to the users own needs, is known an 'Application Generator'.
11.3 What is a Database?

This term is not used in a consistent way.

It can refer to a single table . . .
   (ie. equivalent to a single array of records)

. . . or it can refer to a collection of tables.

This text will use the following definitions:

A 'database' is a collection of data tables.

A 'table' is single array of records.

A 'row' is a single record in a table.

A 'column' is a field that is common to all rows.

A 'field' is a single column of a single row.

In some texts a table may be referred to as a 'relation', a record as a 'tuple' and a field as an 'attribute'.

ie. The following are equivalent:

<table>
<thead>
<tr>
<th>Table</th>
<th>Relation</th>
<th>Array of records</th>
</tr>
</thead>
<tbody>
<tr>
<td>Row</td>
<td>Tuple</td>
<td>Record</td>
</tr>
<tr>
<td>Column</td>
<td>Attribute</td>
<td>Field</td>
</tr>
</tbody>
</table>
11.4 Clients and Servers

Large database systems, such as Oracle, are frequently accessed as a "server" by other software known as the "client". This is particularly common on networked systems:

- The server holds all the data on a powerful central computer.

- The client gives the user interface application software communicating with the server from a smaller computer such as a PC.

- The client may have its own database facility used to keep local data accessible only to the user of that particular client computer.

A standard known as "Open Data Base Connection" (referred to as ODBC) gives a standard communication protocol between clients and servers.

Oracle conforms to this standard allowing it to be accessed by client software produced by many different software companies.

Examples of client software that can interface to Oracle or other ODBC servers is:

- Visual Studio - MicroSoft's software for generating Windows applications using the Visual Basic or C# programming languages
11.5 **SQL Language Syntax : Names**

The data in a database is organised into tables and columns within the tables, each of which has a name.

On most SQL systems names of tables, column headings, etc. must:

- Start with a letter
- Contain only letters, digits and the underscore
- Be unique in 30 characters
- Be distinct from the many SQL reserved words

In Access/Visual Basic SQL names can include spaces, but in these cases the name must be enclosed in [] whenever used in an SQL command.

Tables names by default will refer to the users own tables.

Tables of other users can be accessed using:

```plaintext
username.tablename
```

but access will only be possible if permission has been granted by the table owner.

The "." notation can also be used to refer to columns if the same column name appears in more than one table:

```plaintext
tablename.columnname
```
11.6 **SQL Language Syntax : Commands**

Case is not significant except inside string constants.

Commands are normally terminated with a ';'.

In Visual Basic SQL the ';' terminator is not required.

Commands may be spread over more than one line.

11.7 **SQL Language Syntax : Constants**

Number constants in SQL commands are as written:

eg. 42 or 1.234

String constants in SQL commands are enclosed in ' ':

eg. 'All my own work'

N.B. ' ' are not the same as " ".

' ' are used for string constants

' ' may also be used for dates

" " may have special uses on different systems.

In Visual Basic SQL " " can often be used instead of ' '.

---

Case is not significant except inside string constants.
Commands are normally terminated with a ';'.
In Visual Basic SQL the ';' terminator is not required.
Commands may be spread over more than one line.

Number constants in SQL commands are as written:

eg. 42 or 1.234

String constants in SQL commands are enclosed in ' ':

eg. 'All my own work'

N.B. ' ' are not the same as " ".

' ' are used for string constants

' ' may also be used for dates

" " may have special uses on different systems.

In Visual Basic SQL " " can often be used instead of ' '.

---

**11.6 SQL Language Syntax : Commands**

Case is not significant except inside string constants.

Commands are normally terminated with a ';'.

In Visual Basic SQL the ';' terminator is not required.

Commands may be spread over more than one line.

**11.7 SQL Language Syntax : Constants**

Number constants in SQL commands are as written:

eg. 42 or 1.234

String constants in SQL commands are enclosed in ' ':

eg. 'All my own work'

N.B. ' ' are not the same as " ".

' ' are used for string constants

' ' may also be used for dates

" " may have special uses on different systems.

In Visual Basic SQL " " can often be used instead of ' '.

---
11.8 Example Tables

The following two tables are used in the examples given in this text. These are the tables normally supplied with Oracle systems in the Oracle user area with user name "SCOTT" and password "TIGER".

The EMP table, a typical company personnel table:

<table>
<thead>
<tr>
<th>Empno</th>
<th>Ename</th>
<th>Job</th>
<th>Mgr</th>
<th>Hiredate</th>
<th>Sal</th>
<th>Comm</th>
<th>Deptno</th>
</tr>
</thead>
<tbody>
<tr>
<td>7369</td>
<td>SMITH</td>
<td>CLERK</td>
<td>7902</td>
<td>17/12/80</td>
<td>800</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>7499</td>
<td>ALLEN</td>
<td>SALESMAN</td>
<td>7698</td>
<td>20/02/81</td>
<td>1600</td>
<td>300</td>
<td>30</td>
</tr>
<tr>
<td>7521</td>
<td>WARD</td>
<td>SALESMAN</td>
<td>7698</td>
<td>22/02/81</td>
<td>1250</td>
<td>500</td>
<td>30</td>
</tr>
<tr>
<td>7566</td>
<td>JONES</td>
<td>MANAGER</td>
<td>7839</td>
<td>02/04/81</td>
<td>2975</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>7654</td>
<td>MARTIN</td>
<td>SALESMAN</td>
<td>7698</td>
<td>28/09/81</td>
<td>1250</td>
<td>1400</td>
<td>30</td>
</tr>
<tr>
<td>7698</td>
<td>BLAKE</td>
<td>MANAGER</td>
<td>7839</td>
<td>01/05/81</td>
<td>2850</td>
<td>30</td>
<td></td>
</tr>
<tr>
<td>7782</td>
<td>CLARK</td>
<td>MANAGER</td>
<td>7839</td>
<td>09/06/81</td>
<td>2450</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>7788</td>
<td>SCOTT</td>
<td>ANALYST</td>
<td>7566</td>
<td>09/12/82</td>
<td>3000</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>7839</td>
<td>KING</td>
<td>PRESIDENT</td>
<td>7566</td>
<td>17/11/81</td>
<td>5000</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>7844</td>
<td>TURNER</td>
<td>SALESMAN</td>
<td>7698</td>
<td>08/09/81</td>
<td>1500</td>
<td>0</td>
<td>30</td>
</tr>
<tr>
<td>7876</td>
<td>ADAMS</td>
<td>CLERK</td>
<td>7788</td>
<td>12/01/83</td>
<td>1100</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>7900</td>
<td>JAMES</td>
<td>CLERK</td>
<td>7698</td>
<td>03/12/81</td>
<td>950</td>
<td>30</td>
<td></td>
</tr>
<tr>
<td>7902</td>
<td>FORD</td>
<td>ANALYST</td>
<td>7566</td>
<td>03/12/81</td>
<td>3000</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>7934</td>
<td>MILLER</td>
<td>CLERK</td>
<td>7782</td>
<td>23/01/82</td>
<td>1300</td>
<td>10</td>
<td></td>
</tr>
</tbody>
</table>

The DEPT table with details of the departments:

<table>
<thead>
<tr>
<th>Deptno</th>
<th>Dname</th>
<th>Loc</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>ACCOUNTING</td>
<td>NEW YORK</td>
</tr>
<tr>
<td>20</td>
<td>RESEARCH</td>
<td>DALLAS</td>
</tr>
<tr>
<td>30</td>
<td>SALES</td>
<td>CHICAGO</td>
</tr>
<tr>
<td>40</td>
<td>OPERATIONS</td>
<td>BOSTON</td>
</tr>
</tbody>
</table>
11.9 The TABLE Command

Although it is not in the SQL standard many systems, such as Access, provide a TABLE command which simply displays the content of a table.

Eg. TABLE emp;

This will display the contents of the emp table with suitable column headers.

This command is not available in all versions of SQL. If not available the simple alternative is to use the equivalent SELECT command:

ie. SELECT * FROM emp;
CHAPTER 12

Selecting Data From the Database
With SQL

Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>12.1</td>
<td>The SQL Command: <code>SELECT</code></td>
<td>150</td>
</tr>
<tr>
<td>12.2</td>
<td>Multiple Columns and Column Headers</td>
<td>151</td>
</tr>
<tr>
<td>12.3</td>
<td>Example Output From A <code>SELECT</code> Command</td>
<td>152</td>
</tr>
<tr>
<td>12.4</td>
<td>Further <code>SELECT</code> Examples</td>
<td>153</td>
</tr>
<tr>
<td>12.5</td>
<td>A <code>SELECT</code> Example With New Column Headers</td>
<td>154</td>
</tr>
<tr>
<td>12.6</td>
<td>The <code>SELECT...WHERE</code> Command</td>
<td>155</td>
</tr>
<tr>
<td>12.7</td>
<td><code>SELECT...WHERE</code> Examples</td>
<td>156</td>
</tr>
<tr>
<td>12.8</td>
<td>The <code>SELECT...ORDER BY</code> Clause</td>
<td>157</td>
</tr>
<tr>
<td>12.9</td>
<td><code>SELECT ... ORDER BY</code> Examples</td>
<td>158</td>
</tr>
<tr>
<td>12.10</td>
<td>The <code>SELECT DISTINCT</code> Command</td>
<td>159</td>
</tr>
<tr>
<td>12.11</td>
<td>The <code>SELECT TOP n</code> Command</td>
<td>160</td>
</tr>
<tr>
<td>12.12</td>
<td>Group Functions and the <code>GROUP BY</code> Clause</td>
<td>161</td>
</tr>
<tr>
<td>12.13</td>
<td>Group Function Output</td>
<td>162</td>
</tr>
<tr>
<td>12.14</td>
<td>Restrictions on <code>GROUP BY</code> Selections</td>
<td>163</td>
</tr>
<tr>
<td>12.15</td>
<td>The <code>HAVING</code> Clause</td>
<td>164</td>
</tr>
<tr>
<td>12.16</td>
<td>The <code>SELECT INTO</code> Command</td>
<td>165</td>
</tr>
<tr>
<td>12.17</td>
<td>The <code>SELECT ... FOR UPDATE</code> Command</td>
<td>166</td>
</tr>
<tr>
<td>12.18</td>
<td>The <code>IN</code> Table Qualifier</td>
<td>167</td>
</tr>
<tr>
<td>12.19</td>
<td>The Order of <code>SELECT</code> Clauses</td>
<td>167</td>
</tr>
</tbody>
</table>
12.1 **The SQL Command:** `SELECT`  

The purpose of this command to retrieve and display records from data tables.

The simple `SELECT` command format is:

```
SELECT select-list FROM table-name;
```

The `select-list` specifies which columns from the table are displayed and can be any of:

- * to indicate all columns in the table
- column name or names separated with commas
- real or integer number constants
- character string constants enclosed in ' ', *(not " ")*
- number expressions with columns, constants, variables or functions combined with + - */ ( )
- character columns, constants, variables or functions concatenated with &
- dates enclosed in ' '
  *(or in # # in Access/Visual Basic SQL)*
- group functions
12.2 Multiple Columns and Column Headers

If more than one column is to be displayed the column names are separated by commas in the SELECT command.

The output is in a table form with one space between each column and the column name as a header.

Each header is underlined with ---- to the width of the column displayed.

Most systems will allow a single word alternative header to be specified if the select-list option is followed by a space (in Oracle) or the keyword AS (in Access/Visual Basic) and new column header.

The system may allow alternative headers to contain spaces if the header text is enclosed in " " (in Oracle) or [ ] (in Access/Visual Basic).

Standard SQL select list examples:

```
*  
ENAME, SAL, JOB
```

Oracle select list example:

```
SAL Salary, SAL*1.1 "New Salary"
```

Access/Visual Basic select list example:

```
ENAME, SAL AS [Basic Salary], JOB
```
The SQL Command:

```
SELECT * FROM emp;
```

Would display the following output:

<table>
<thead>
<tr>
<th>EMPNO</th>
<th>ENAME</th>
<th>JOB</th>
<th>MGR</th>
<th>HIREDATE</th>
<th>SAL</th>
<th>COMM</th>
<th>DEPTNO</th>
</tr>
</thead>
<tbody>
<tr>
<td>7369</td>
<td>SMITH</td>
<td>CLERK</td>
<td>7902</td>
<td>17/12/80</td>
<td>800</td>
<td></td>
<td>20</td>
</tr>
<tr>
<td>7499</td>
<td>ALLEN</td>
<td>SALESMAN</td>
<td>7698</td>
<td>20/02/81</td>
<td>1600</td>
<td>300</td>
<td>30</td>
</tr>
<tr>
<td>7521</td>
<td>WARD</td>
<td>SALESMAN</td>
<td>7698</td>
<td>22/02/81</td>
<td>1250</td>
<td>500</td>
<td>30</td>
</tr>
<tr>
<td>7566</td>
<td>JONES</td>
<td>MANAGER</td>
<td>7839</td>
<td>02/04/81</td>
<td>2975</td>
<td></td>
<td>20</td>
</tr>
<tr>
<td>7654</td>
<td>MARTIN</td>
<td>SALESMAN</td>
<td>7698</td>
<td>28/09/81</td>
<td>1250</td>
<td>1400</td>
<td>30</td>
</tr>
<tr>
<td>7698</td>
<td>BLAKE</td>
<td>MANAGER</td>
<td>7839</td>
<td>01/05/81</td>
<td>2850</td>
<td></td>
<td>30</td>
</tr>
<tr>
<td>7782</td>
<td>CLARK</td>
<td>MANAGER</td>
<td>7839</td>
<td>09/06/81</td>
<td>2450</td>
<td></td>
<td>10</td>
</tr>
<tr>
<td>7788</td>
<td>SCOTT</td>
<td>ANALYST</td>
<td>7566</td>
<td>09/12/82</td>
<td>3000</td>
<td></td>
<td>20</td>
</tr>
<tr>
<td>7839</td>
<td>KING</td>
<td>PRESIDENT</td>
<td></td>
<td>17/11/81</td>
<td>5000</td>
<td></td>
<td>10</td>
</tr>
<tr>
<td>7844</td>
<td>TURNER</td>
<td>SALESMAN</td>
<td>7698</td>
<td>08/09/81</td>
<td>1500</td>
<td>0</td>
<td>30</td>
</tr>
<tr>
<td>7876</td>
<td>ADAMS</td>
<td>CLERK</td>
<td>7788</td>
<td>12/01/83</td>
<td>1100</td>
<td></td>
<td>20</td>
</tr>
<tr>
<td>7900</td>
<td>JAMES</td>
<td>CLERK</td>
<td>7698</td>
<td>03/12/81</td>
<td>950</td>
<td></td>
<td>30</td>
</tr>
<tr>
<td>7902</td>
<td>FORD</td>
<td>ANALYST</td>
<td>7566</td>
<td>03/12/81</td>
<td>3000</td>
<td></td>
<td>20</td>
</tr>
<tr>
<td>7934</td>
<td>MILLER</td>
<td>CLERK</td>
<td>7782</td>
<td>23/01/82</td>
<td>1300</td>
<td></td>
<td>10</td>
</tr>
</tbody>
</table>

14 records selected.
12.4 Further **SELECT** Examples

Command: `SELECT * FROM DEPT;`

Output:
```
<table>
<thead>
<tr>
<th>DEPTNO</th>
<th>DNAME</th>
<th>LOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>ACCOUNTING</td>
<td>NEW YORK</td>
</tr>
<tr>
<td>20</td>
<td>RESEARCH</td>
<td>DALLAS</td>
</tr>
<tr>
<td>30</td>
<td>SALES</td>
<td>CHICAGO</td>
</tr>
<tr>
<td>40</td>
<td>OPERATIONS</td>
<td>BOSTON</td>
</tr>
</tbody>
</table>
```

Command: `SELECT LOC,DNAME FROM DEPT;`

Output:
```
<table>
<thead>
<tr>
<th>LOC</th>
<th>DNAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>NEW YORK</td>
<td>ACCOUNTING</td>
</tr>
<tr>
<td>DALLAS</td>
<td>RESEARCH</td>
</tr>
<tr>
<td>CHICAGO</td>
<td>SALES</td>
</tr>
<tr>
<td>BOSTON</td>
<td>OPERATIONS</td>
</tr>
</tbody>
</table>
```

Command: `SELECT SAL,SAL*1.1 FROM EMP;`

Output:
```
<table>
<thead>
<tr>
<th>SAL</th>
<th>SAL*1.1</th>
</tr>
</thead>
<tbody>
<tr>
<td>800</td>
<td>880</td>
</tr>
<tr>
<td>1600</td>
<td>1760</td>
</tr>
<tr>
<td>1250</td>
<td>1375</td>
</tr>
<tr>
<td>2975</td>
<td>3272.5</td>
</tr>
<tr>
<td>1250</td>
<td>1375</td>
</tr>
<tr>
<td>2850</td>
<td>3135</td>
</tr>
<tr>
<td>2450</td>
<td>2695</td>
</tr>
<tr>
<td>3000</td>
<td>3300</td>
</tr>
<tr>
<td>5000</td>
<td>5500</td>
</tr>
<tr>
<td>1500</td>
<td>1650</td>
</tr>
<tr>
<td>1100</td>
<td>1210</td>
</tr>
<tr>
<td>950</td>
<td>1045</td>
</tr>
<tr>
<td>3000</td>
<td>3300</td>
</tr>
<tr>
<td>1300</td>
<td>1430</td>
</tr>
</tbody>
</table>
```
12.5 **A SELECT Example With New Column**

**Headers**

The following Oracle Command:

```sql
SELECT 'Mr.'||ENAME "Name",
       'has new income' "has new income",
       SAL*1.1 "New Salary"
FROM EMP;
```

Would display the following output:

<table>
<thead>
<tr>
<th>Name</th>
<th>has new income</th>
<th>New Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mr.SMITH</td>
<td>has new income</td>
<td>880</td>
</tr>
<tr>
<td>Mr.ALLEN</td>
<td>has new income</td>
<td>1760</td>
</tr>
<tr>
<td>Mr.WARD</td>
<td>has new income</td>
<td>1375</td>
</tr>
<tr>
<td>Mr.JONES</td>
<td>has new income</td>
<td>3272.5</td>
</tr>
<tr>
<td>Mr.MARTIN</td>
<td>has new income</td>
<td>1375</td>
</tr>
<tr>
<td>Mr.BLAKE</td>
<td>has new income</td>
<td>3135</td>
</tr>
<tr>
<td>Mr.CLARK</td>
<td>has new income</td>
<td>2695</td>
</tr>
<tr>
<td>Mr.SCOTT</td>
<td>has new income</td>
<td>3300</td>
</tr>
<tr>
<td>Mr.KING</td>
<td>has new income</td>
<td>5500</td>
</tr>
<tr>
<td>Mr.TURNER</td>
<td>has new income</td>
<td>1650</td>
</tr>
<tr>
<td>Mr.ADAMS</td>
<td>has new income</td>
<td>1210</td>
</tr>
<tr>
<td>Mr.JAMES</td>
<td>has new income</td>
<td>1045</td>
</tr>
<tr>
<td>Mr.FORD</td>
<td>has new income</td>
<td>3300</td>
</tr>
<tr>
<td>Mr.MILLER</td>
<td>has new income</td>
<td>1430</td>
</tr>
</tbody>
</table>

14 records selected

(Note: The `||` is the string concatenation operator in Oracle SQL)
12.6 The **SELECT...WHERE** Command

The **WHERE** clause is used on the **SELECT** command to specify which records are to be displayed.

*All* records are displayed if there is no **WHERE** clause.

The format is:

```
SELECT <select-list>
FROM <tablename>
WHERE <condition>;
```

Where `<condition>` is one of:

- `expression1 = expression2`
- `expression1 <> expression2`
- `expression1 > expression2`
- `expression1 < expression2`
- `expression1 >= expression2`
- `expression1 <= expression2`
- `expression [NOT] BETWEEN value1 AND value2`
- `expression [NOT] IN (value1,value2...)`
- `column IS [NOT] NULL`
- `char_expression [NOT] LIKE template`

These conditions can be combined with **AND** and **OR**.

A template is a character string used in matching character expressions that may contain characters with special meanings as follows:

<table>
<thead>
<tr>
<th>Oracle</th>
<th>Access</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>_</td>
<td>?</td>
<td>Matches a single character</td>
</tr>
<tr>
<td>*</td>
<td>#</td>
<td>Matches any number of characters</td>
</tr>
<tr>
<td>%</td>
<td>*</td>
<td>Matches any digit</td>
</tr>
</tbody>
</table>
12.7 **SELECT...WHERE Examples**

**Command:**
```
SELECT ENAME, JOB, SAL
FROM EMP
WHERE SAL>=2000;
```

**Output:**

<table>
<thead>
<tr>
<th>ENAME</th>
<th>JOB</th>
<th>SAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>JONES</td>
<td>MANAGER</td>
<td>2975</td>
</tr>
<tr>
<td>BLAKE</td>
<td>MANAGER</td>
<td>2850</td>
</tr>
<tr>
<td>CLARK</td>
<td>MANAGER</td>
<td>2450</td>
</tr>
<tr>
<td>SCOTT</td>
<td>ANALYST</td>
<td>3000</td>
</tr>
<tr>
<td>KING</td>
<td>PRESIDENT</td>
<td>5000</td>
</tr>
<tr>
<td>FORD</td>
<td>ANALYST</td>
<td>3000</td>
</tr>
</tbody>
</table>

**Command:**
```
SELECT EMPNO, ENAME, JOB,
SAL+COMM INCOME
FROM EMP
WHERE COMM IS NOT NULL
AND COMM>0
AND EMPNO IN (7499,7502,7519,
7654,7663,7701);
```

**Output:**

<table>
<thead>
<tr>
<th>EMPNO</th>
<th>ENAME</th>
<th>JOB</th>
<th>INCOME</th>
</tr>
</thead>
<tbody>
<tr>
<td>7499</td>
<td>ALLEN</td>
<td>SALESMAN</td>
<td>1900</td>
</tr>
<tr>
<td>7654</td>
<td>MARTIN</td>
<td>SALESMAN</td>
<td>2650</td>
</tr>
</tbody>
</table>

**Oracle Command:**
```
SELECT ENAME, JOB FROM EMP
WHERE ENAME LIKE 'A%'
OR ENAME LIKE 'B%';
```

**Output:**

<table>
<thead>
<tr>
<th>ENAME</th>
<th>JOB</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALLEN</td>
<td>SALESMAN</td>
</tr>
<tr>
<td>BLAKE</td>
<td>MANAGER</td>
</tr>
<tr>
<td>ADAMS</td>
<td>CLERK</td>
</tr>
</tbody>
</table>
12.8 The `SELECT...ORDER BY` Clause

The order of the records can be determined by the `ORDER BY` clause specified in the form:

```
SELECT <select-list>
FROM <tablename>
WHERE <condition>
ORDER BY <column(s)>;
```

Order is descending if `DESC` is added at the end of the clause in the form:

```
SELECT <select-list>
FROM <tablename>
WHERE <condition>
ORDER BY <column(s)> DESC;
```

If no `ORDER BY` clause is given the records are output in the order they are stored.

If more than one column is specified in an `ORDER BY` clause the output is sorted as follows:

1. The whole output is sorted on the first column mentioned following the `ORDER BY`.
2. For equal values of the first mentioned column the second `ORDER BY` column is used.
3. Similarly, where the first and second columns are equal the third column is used, etc.

In Access, unlike the `WHERE` clause, the `ORDER BY` clause can also be used in a `TABLE` command.

Eg. `TABLE emp ORDER BY ename;`
12.9 SELECT ... ORDER BY Examples

Command:
```
SELECT DEPTNO, ENAME, JOB, SAL
FROM EMP
ORDER BY DEPTNO, SAL DESC;
```

Output:
```
<table>
<thead>
<tr>
<th>DEPTNO</th>
<th>ENAME</th>
<th>JOB</th>
<th>SAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>KING</td>
<td>PRESIDENT</td>
<td>5000</td>
</tr>
<tr>
<td>10</td>
<td>CLARK</td>
<td>MANAGER</td>
<td>2450</td>
</tr>
<tr>
<td>10</td>
<td>MILLER</td>
<td>CLERK</td>
<td>1300</td>
</tr>
<tr>
<td>20</td>
<td>SCOTT</td>
<td>ANALYST</td>
<td>3000</td>
</tr>
<tr>
<td>20</td>
<td>FORD</td>
<td>ANALYST</td>
<td>3000</td>
</tr>
<tr>
<td>20</td>
<td>JONES</td>
<td>MANAGER</td>
<td>2975</td>
</tr>
<tr>
<td>20</td>
<td>ADAMS</td>
<td>CLERK</td>
<td>1100</td>
</tr>
<tr>
<td>20</td>
<td>SMITH</td>
<td>CLERK</td>
<td>800</td>
</tr>
<tr>
<td>30</td>
<td>BLAKE</td>
<td>MANAGER</td>
<td>2850</td>
</tr>
<tr>
<td>30</td>
<td>ALLEN</td>
<td>SALESMAN</td>
<td>1600</td>
</tr>
<tr>
<td>30</td>
<td>TURNER</td>
<td>SALESMAN</td>
<td>1500</td>
</tr>
<tr>
<td>30</td>
<td>WARD</td>
<td>SALESMAN</td>
<td>1250</td>
</tr>
<tr>
<td>30</td>
<td>MARTIN</td>
<td>SALESMAN</td>
<td>1250</td>
</tr>
<tr>
<td>30</td>
<td>JAMES</td>
<td>CLERK</td>
<td>950</td>
</tr>
</tbody>
</table>
```

Command:
```
SELECT DEPTNO, ENAME, JOB, SAL
FROM EMP
WHERE COMM IS NULL
AND   SAL >= 1000
ORDER BY DEPTNO DESC, SAL DESC, ENAME;
```

Output:
```
<table>
<thead>
<tr>
<th>DEPTNO</th>
<th>ENAME</th>
<th>JOB</th>
<th>SAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>30</td>
<td>BLAKE</td>
<td>MANAGER</td>
<td>2850</td>
</tr>
<tr>
<td>20</td>
<td>FORD</td>
<td>ANALYST</td>
<td>3000</td>
</tr>
<tr>
<td>20</td>
<td>SCOTT</td>
<td>ANALYST</td>
<td>3000</td>
</tr>
<tr>
<td>20</td>
<td>JONES</td>
<td>MANAGER</td>
<td>2975</td>
</tr>
<tr>
<td>20</td>
<td>ADAMS</td>
<td>CLERK</td>
<td>1100</td>
</tr>
<tr>
<td>10</td>
<td>KING</td>
<td>PRESIDENT</td>
<td>5000</td>
</tr>
<tr>
<td>10</td>
<td>CLARK</td>
<td>MANAGER</td>
<td>2450</td>
</tr>
<tr>
<td>10</td>
<td>MILLER</td>
<td>CLERK</td>
<td>1300</td>
</tr>
</tbody>
</table>
```
12.10 The select distinct Command

The DISTINCT clause on the SELECT command removes duplicate records from the display output.

The format is:

```
SELECT DISTINCT select-list
FROM table
...etc. ;
```

Example:

Command:
```
SELECT DISTINCT DEPTNO, JOB
FROM EMP
ORDER BY DEPTNO;
```

Output:
```
DEPTNO JOB
------ --------
  10  CLERK
  10  MANAGER
  10 PRESIDENT
  20 ANALYST
  20  CLERK
  20  MANAGER
  30  CLERK
  30 MANAGER
  30  SALESMAN
```
12.11 The \texttt{SELECT TOP n} Command

Visual Basic/Access SQL also offers \texttt{SELECT TOP n}.

This command outputs only the first \textit{n} records of the select.

It is particularly useful when the output is ordered.

Example:

\textbf{Command:} \ \ \ \ \texttt{SELECT TOP 5 DEPTNO, ENAME, JOB, SAL \\
FROM EMP \\
ORDER BY DEPTNO, SAL DESC;}

\textbf{Output:}  \\
\begin{tabular}{llll}
DEPTNO & ENAME     & JOB         & SAL  \\
------ & ---------- & ----------- & ------
10  & KING       & PRESIDENT   & 5000  \\
10  & CLARK      & MANAGER     & 2450  \\
10  & MILLER     & CLERK       & 1300  \\
20  & SCOTT      & ANALYST     & 3000  \\
20  & FORD       & ANALYST     & 3000  \\
\end{tabular}
12.12 Group Functions and the GROUP BY Clause

SQL offers a means of looking at summary results of all or part of a table by using group functions in a SELECT command:

- **COUNT(column_expression)**: No. of non-null values
- **SUM(column_expression)**: Sum of values
- **AVG(column_expression)**: Average value
- **MAX(column_expression)**: Highest value
- **MIN(column_expression)**: Lowest value

eg. 
- `SELECT MAX(sal), AVG(comm) FROM emp;`
- `SELECT COUNT(ENAME) FROM EMP WHERE JOB='SALESMAN';`

The SELECT command with the GROUP BY clause can be used to give summary results for selected groups of rows.

eg. 
- `SELECT deptno, SUM(sal) FROM emp GROUP BY deptno;`

The group function can be modified with the DISTINCT keyword:

eg. 
- `SELECT COUNT(DISTINCT JOB) FROM EMP;`

This will count the number of different jobs.
12.13 **Group Function Output**

Grouping over the whole report:

```sql
SELECT COUNT(*) "Number",
       SUM(sal) "Total"
FROM emp;
```

would give an output (using Oracle) in the form:

<table>
<thead>
<tr>
<th>Number</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>------</td>
<td>-------</td>
</tr>
<tr>
<td>14</td>
<td>29025</td>
</tr>
</tbody>
</table>

(1 row of output only)

Grouping over selected ranges of records:

```sql
SELECT deptno, SUM(sal) "Total"
FROM emp
GROUP BY deptno;
```

would give the following output in Oracle:

<table>
<thead>
<tr>
<th>deptno</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>------</td>
<td>-----</td>
</tr>
<tr>
<td>10</td>
<td>8750</td>
</tr>
<tr>
<td>20</td>
<td>10875</td>
</tr>
<tr>
<td>30</td>
<td>9400</td>
</tr>
</tbody>
</table>

(1 row for each department in the emp table)

Note that the `GROUP BY` clause implies an `ORDER BY` on the same column(s).
12.14 Restrictions on GROUP BY Selections

As the GROUP BY clause causes only the summary information to be output the selected information must make sense in a summary.

ie. Selections are restricted to:

1. Group functions.
2. Constants.
3. Columns or column expressions also specified in the GROUP BY clause.

A group can be any size from a single record to the whole table.

A selection grouping can be made on multiple columns.

ie. To group by deptno, job will create a new group for every unique combination of dept and job.
12.15 The HAVING Clause

It is not necessary to display all groups on any column, groups can be selectively displayed with the HAVING clause.

The HAVING clause is similar to the WHERE clause except that it applies to conditions on groups.

eg. SELECT deptno, SUM(sal) "Total Sal"
    FROM emp
    WHERE deptno <> 10
    GROUP BY deptno
    HAVING MIN(sal) > 9000;

The HAVING clause should only be used with, and straight after a GROUP BY clause.

It is possible to use a HAVING clause where a WHERE clause could have been used:

eg. SELECT deptno, SUM(sal)
    FROM emp
    GROUP BY deptno
    HAVING deptno <> 10;

    is equivalent to SELECT deptno, SUM(sal)
    FROM emp
    WHERE deptno <> 10
    GROUP BY deptno;

A WHERE selection is more efficient than the HAVING equivalent, however, so a WHERE should always be used unless the condition contains a group function.
12.16 *The SELECT INTO Command*

In Visual Basic/Access SQL it is possible to direct the output of a `SELECT` statement to create a new table.

The format is:

```
SELECT field1,field2...fieldn
    INTO newtablename
    FROM ..... etc.
```

The column names and types for the new table are the same as the data selected.

Note that if the table already exists it is completely overwritten by the new data.

Example:

```
SELECT ename,dname
    INTO NameAndDept
    FROM emp INNER JOIN dept
        ON emp.deptno=dept.deptno;
```

This will create a new table called `NameAndDept` with two columns, `ename` and `dname` with the same types as in the `emp` and `dept` tables.

Note: SQL/92 and Oracle also have an INTO clause when the SQL is embedded within a program. In this case it is a program variable rather than another table that is the destination for the data.
12.17 The `SELECT ... FOR UPDATE` Command

In Oracle, the `SELECT` command can be used to flag rows in a table that are to be modified by the user by adding an extra clause in the form:

```sql
SELECT ......
FOR UPDATE OF column1,column2,... [NOWAIT];
```

eg.  
```sql
SELECT ename,sal,comm
    FROM emp
    FOR UPDATE OF sal,comm NOWAIT;
```

Notes:

- This command locks the rows so that other users cannot change the selected rows until the changes have been completed.

- The optional extra keyword `NOWAIT` prevents the system waiting while another user finishes their changes so that it can lock the selected rows.

- It is not essential to use this command before a modification is made to a table and for a single user system there would be no need to do so.

- This clause is not part of the SQL/92 standard and is not available in Access/Visual Basic.
12.18 The **IN** Table Qualifier

Visual Basic and Access normally store the data tables associated with a single database in a single file with a name `xxx.MDB`

In Visual Basic/Access SQL data tables in different files in the **FROM** and **INTO** clauses can be handled using an **IN** qualifier.

Examples:

```
SELECT * FROM emp IN C:\MYDIR\MYDB.MDB
```

```
SELECT ename,dname INTO NameAndDept IN C:\MYDIR\MYDB.MDB
FROM emp,dept
WHERE emp.deptno=dept.deptno;
```

Note: This is NOT part of the SQL/92 standard.

12.19 The Order of **SELECT** Clauses

The order of the clauses on a **SELECT** command is fixed and should always be as follows:

```
INTO (IN)
FROM (IN)
ON
WHERE
GROUP BY
HAVING
ORDER BY
FOR UPDATE OF
```
CHAPTER 13
Joining Database Tables Together

Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>13.1</td>
<td>Types of Table Join</td>
<td>169</td>
</tr>
<tr>
<td>13.2</td>
<td>The Natural Inner Join</td>
<td>170</td>
</tr>
<tr>
<td>13.3</td>
<td>Other Types of Inner Join</td>
<td>171</td>
</tr>
<tr>
<td>13.4</td>
<td>INNER JOIN Example</td>
<td>172</td>
</tr>
<tr>
<td>13.5</td>
<td>The Theta-Join</td>
<td>173</td>
</tr>
<tr>
<td>13.6</td>
<td>The CROSS JOIN</td>
<td>174</td>
</tr>
<tr>
<td>13.7</td>
<td>Converting a CROSS JOIN to an INNER JOIN</td>
<td>175</td>
</tr>
<tr>
<td>13.8</td>
<td>Inner and Outer Joins</td>
<td>176</td>
</tr>
<tr>
<td>13.9</td>
<td>Outer Join Syntax</td>
<td>177</td>
</tr>
<tr>
<td>13.10</td>
<td>Outer Join Example</td>
<td>178</td>
</tr>
<tr>
<td>13.11</td>
<td>Outer Joins in Oracle SQL</td>
<td>179</td>
</tr>
<tr>
<td>13.12</td>
<td>Table Alias Names</td>
<td>180</td>
</tr>
<tr>
<td>13.13</td>
<td>Example of a Table Joined to Itself</td>
<td>181</td>
</tr>
<tr>
<td>13.14</td>
<td>Example of a Three Table Join</td>
<td>182</td>
</tr>
</tbody>
</table>
13.1 **Types of Table Join**

Oracle allows the `SELECT` command to display information from more than one table at a time.

The information from the two tables is said to be "joined" when more than one table is displayed.

There is more than one type of join.

The types are:

**Inner Joins**
- The Natural Join
- The Equi-Join
- The Theta Join

**Outer Joins**
- The Natural Left Join
- Other Left Joins
- The Natural Right Join
- Other Right Joins
- The Natural Full Join
- Other Full Joins

**Cross Joins**
13.2 The Natural Inner Join

This is the most commonly required join.

Two tables are joined such that all rows in each table are linked to the rows in the other table such that any columns with the same name have corresponding values.

Eg. If the emp and dept tables are joined with a natural join they would be linked where the common column, deptno, had the same value in each table.

The syntax in standard SQL for this would be:

```sql
SELECT * FROM emp NATURAL JOIN dept;
```

This would cause the columns from both tables to be matched with the common column listed once only.

Standard SQL also gives the option of specifying which columns are to be matched using the USING qualifier.

```sql
eg. SELECT * FROM emp INNER JOIN dept
    USING deptno;
```

This would give an output similar to the natural join example as, in this case, there are no other matching columns anyway.

*Unfortunately neither type of natural join syntax is available in Oracle or Access SQL!*
13.3 Other Types of Inner Join

The other type of inner join available in standard SQL is where the join condition is explicitly specified.

The syntax is

\(<\text{Table1}>\text{ INNER JOIN } <\text{Table2}>\text{ ON } <\text{condition}>\)

Eg. \(\text{SELECT } * \text{ FROM emp INNER JOIN dept ON emp.deptno = dept.deptno;}\)

*This type of join is available in Visual Basic/Access SQL- but not in Oracle SQL.*

If, like the natural join, the condition involves matching columns in the two tables as above, this is called the equi-join.

The difference between the equi-join and the natural join is:

1. The matching columns need not have the same name.

2. The matching columns from both tables will be listed, even though their values are identical if all columns are selected.

A join involving a comparison other than "=" is known as a theta-join.
13.4 **INNER JOIN Example**

Access/Visual Basic Command:

```sql
SELECT empno,ename,emp.deptno,dname
FROM emp INNER JOIN dept
ON emp.deptno=dept.deptno;
```

Output:

<table>
<thead>
<tr>
<th>EMPNO</th>
<th>ENAME</th>
<th>EMP.DE</th>
<th>DNAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>7369</td>
<td>SMITH</td>
<td>20</td>
<td>RESEARCH</td>
</tr>
<tr>
<td>7499</td>
<td>ALLEN</td>
<td>30</td>
<td>SALES</td>
</tr>
<tr>
<td>7521</td>
<td>WARD</td>
<td>30</td>
<td>SALES</td>
</tr>
<tr>
<td>7566</td>
<td>JONES</td>
<td>20</td>
<td>RESEARCH</td>
</tr>
<tr>
<td>7654</td>
<td>MARTIN</td>
<td>30</td>
<td>SALES</td>
</tr>
<tr>
<td>7698</td>
<td>BLAKE</td>
<td>30</td>
<td>SALES</td>
</tr>
<tr>
<td>7782</td>
<td>CLARK</td>
<td>10</td>
<td>ACCOUNTING</td>
</tr>
<tr>
<td>7788</td>
<td>SCOTT</td>
<td>20</td>
<td>RESEARCH</td>
</tr>
<tr>
<td>7839</td>
<td>KING</td>
<td>10</td>
<td>ACCOUNTING</td>
</tr>
<tr>
<td>7844</td>
<td>TURNER</td>
<td>30</td>
<td>SALES</td>
</tr>
<tr>
<td>7876</td>
<td>ADAMS</td>
<td>20</td>
<td>RESEARCH</td>
</tr>
<tr>
<td>7900</td>
<td>JAMES</td>
<td>30</td>
<td>SALES</td>
</tr>
<tr>
<td>7902</td>
<td>FORD</td>
<td>20</td>
<td>RESEARCH</td>
</tr>
<tr>
<td>7934</td>
<td>MILLER</td>
<td>10</td>
<td>ACCOUNTING</td>
</tr>
</tbody>
</table>

(14 records selected.)
13.5 The Theta-Join

The theta-join is a join where a more complicated condition is used.

Eg. Suppose all managers are required to visit all the other departments.

An administrator may be asked to arrange the visits according to the following Access/Visual Basic command:

```
SELECT ENAME AS Name,
    'to visit' AS [to visit],
    LOC AS Place
FROM EMP INNER JOIN DEPT
    ON EMP.DEPTNO <> DEPT.DEPTNO
WHERE JOB='MANAGER';
```

This would give the following output:

<table>
<thead>
<tr>
<th>Name</th>
<th>to visit</th>
<th>Place</th>
</tr>
</thead>
<tbody>
<tr>
<td>JONES</td>
<td>to visit</td>
<td>NEW YORK</td>
</tr>
<tr>
<td>JONES</td>
<td>to visit</td>
<td>CHICAGO</td>
</tr>
<tr>
<td>JONES</td>
<td>to visit</td>
<td>BOSTON</td>
</tr>
<tr>
<td>BLAKE</td>
<td>to visit</td>
<td>NEW YORK</td>
</tr>
<tr>
<td>BLAKE</td>
<td>to visit</td>
<td>DALLAS</td>
</tr>
<tr>
<td>BLAKE</td>
<td>to visit</td>
<td>BOSTON</td>
</tr>
<tr>
<td>CLARK</td>
<td>to visit</td>
<td>DALLAS</td>
</tr>
<tr>
<td>CLARK</td>
<td>to visit</td>
<td>CHICAGO</td>
</tr>
<tr>
<td>CLARK</td>
<td>to visit</td>
<td>BOSTON</td>
</tr>
</tbody>
</table>

9 records are selected with every manager visiting every location but their own.
13.6 The **CROSS JOIN**

Standard SQL allows a cross join to be specified:

ie.    \(<\text{Table1}>\) CROSS JOIN \(<\text{Table2}>\)

or simply:    \(<\text{Table1}>, \ <\text{Table2}>\)

eg.    \(\text{SELECT} \ * \ \text{FROM emp CROSS JOIN dept;}\)

or    \(\text{SELECT} \ * \ \text{FROM emp, dept;}\)

In this example:

- Each line of output will contain all columns from both tables. ie. \(8+3 = 11\) columns

- As there are no \texttt{WHERE} restrictions the number of rows displayed will be:
  
  \(\text{no.of rows in 1st table} \ * \ \text{no.of rows in 2nd table}\)
  
  ie. every combination of a row from \texttt{emp} will be put with every row from \texttt{dept}.
  
  This will give \(14*4 = 56\) rows of output!

*This type of join is also known as the "Cartesian Product", but it has few practical uses!*

**Note:** The "," syntax is an older form of syntax and is the only form of \texttt{CROSS JOIN} syntax recognised by Visual Basic/Access SQL. The "," is also recognised by Oracle SQL.
13.7 Converting a CROSS JOIN to an INNER JOIN

Any cross join can be made into an ordinary inner join by specifying the join condition in a WHERE clause:

ie.  

\[
\begin{align*}
\text{SELECT ename, dept.deptno,} \\
\text{dname, loc} \\
\text{FROM emp INNER JOIN dept} \\
\text{ON emp.deptno = dept.deptno;}
\end{align*}
\]

is equivalent to:

\[
\begin{align*}
\text{SELECT ename, dept.deptno,} \\
\text{dname, loc} \\
\text{FROM emp CROSS JOIN dept} \\
\text{WHERE emp.deptno = dept.deptno;}
\end{align*}
\]

or:

\[
\begin{align*}
\text{SELECT ename, dept.deptno,} \\
\text{dname, loc} \\
\text{FROM emp, dept} \\
\text{WHERE emp.deptno = dept.deptno;}
\end{align*}
\]

The last syntax was the only format allowed in the earlier SQL standard (SQL/89) ....

.... it is still in common use today ....

.... and it is still the only format available for inner joins in many SQL systems.

The "," cross join modified by WHERE conditions is the only form of join allowed in Oracle SQL.
13.8 Inner and Outer Joins

Inner Joins

The examples so far given for the equi-join and theta-join have been inner joins.

An inner join means that if any entry in either table does not have a corresponding entry in the other table it is not listed.

eg. In the equi-join of emp and dept on deptno the Boston department does not appear in the output as there is no matching entry in emp.

Outer Joins

Entries that have no match in the joined table are output with null entries for the other table's fields.

An outer join in standard SQL is specified as a  
LEFT JOIN, RIGHT JOIN or FULL JOIN.

A LEFT JOIN ensures all entries of the left hand table are output regardless of whether there is a match for it or not.

Blank fields are entered from the other table if there is no match.

Similarly, a RIGHT JOIN outputs all rows from the right hand table, and a FULL JOIN outputs all entries from both tables.
13.9 Outer Join Syntax

In standard SQL the LEFT JOIN, RIGHT JOIN and FULL JOIN can be specified with NATURAL as a qualifier or with USING or ON clauses.

ie.  \[
\text{<Table1> NATURAL LEFT JOIN <Table2>}
\]
or  \[
\text{<Table1> LEFT JOIN <Table2> USING <column(s)>}
\]
or  \[
\text{<Table1> LEFT JOIN <Table2> ON <condition>}
\]

However, in Visual Basic/Access SQL:

- The LEFT JOIN or RIGHT JOIN can only be used with the ON clause
- The FULL JOIN is not recognised at all.

eg.  \[
\text{SELECT empno, ename,}
\text{ dept.deptno AS dept,}
\text{ dname, loc}
\text{FROM emp RIGHT JOIN dept}
\text{ON emp.deptno = dept.deptno;}
\]

This will cause the Boston department to be listed with blanks for the \text{empno} and \text{ename} columns.

\textit{Note:} \text{deptno} is specified to be from the \text{dept} table to ensure it is output on any "extended" entries.
13.10  **Outer Join Example**

Command:  
```
SELECT EMPNO, ENAME,  
    DEPT.DEPTNO,  
    DNAME, LOC  
FROM EMP RIGHT JOIN DEPT  
ON EMP.DEPTNO = DEPT.DEPTNO;
```

Output:

<table>
<thead>
<tr>
<th>ENAME</th>
<th>DEPT.D</th>
<th>DNAME</th>
<th>LOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMITH</td>
<td>20</td>
<td>RESEARCH</td>
<td>DALLAS</td>
</tr>
<tr>
<td>ALLEN</td>
<td>30</td>
<td>SALES</td>
<td>CHICAGO</td>
</tr>
<tr>
<td>WARD</td>
<td>30</td>
<td>SALES</td>
<td>CHICAGO</td>
</tr>
<tr>
<td>JONES</td>
<td>20</td>
<td>RESEARCH</td>
<td>DALLAS</td>
</tr>
<tr>
<td>MARTIN</td>
<td>30</td>
<td>SALES</td>
<td>CHICAGO</td>
</tr>
<tr>
<td>BLAKE</td>
<td>30</td>
<td>SALES</td>
<td>CHICAGO</td>
</tr>
<tr>
<td>CLARK</td>
<td>10</td>
<td>ACCOUNTING</td>
<td>NEW YORK</td>
</tr>
<tr>
<td>SCOTT</td>
<td>20</td>
<td>RESEARCH</td>
<td>DALLAS</td>
</tr>
<tr>
<td>KING</td>
<td>10</td>
<td>ACCOUNTING</td>
<td>NEW YORK</td>
</tr>
<tr>
<td>TURNER</td>
<td>30</td>
<td>SALES</td>
<td>CHICAGO</td>
</tr>
<tr>
<td>ADAMS</td>
<td>20</td>
<td>RESEARCH</td>
<td>DALLAS</td>
</tr>
<tr>
<td>JAMES</td>
<td>30</td>
<td>SALES</td>
<td>CHICAGO</td>
</tr>
<tr>
<td>FORD</td>
<td>20</td>
<td>RESEARCH</td>
<td>DALLAS</td>
</tr>
<tr>
<td>MILLER</td>
<td>10</td>
<td>ACCOUNTING</td>
<td>NEW YORK</td>
</tr>
<tr>
<td></td>
<td>40</td>
<td>OPERATIONS</td>
<td>BOSTON</td>
</tr>
</tbody>
</table>

15 records selected

Note it is possible to also have a WHERE clause on an outer join for conditions other than the join condition:

```
SELECT EMPNO, ENAME,  
    DEPT.DEPTNO,  
    DNAME, LOC  
FROM EMP RIGHT JOIN DEPT  
ON EMP.DEPTNO = DEPT.DEPTNO  
WHERE DNAME NOT IN ('SALES', 'RESEARCH');
```
13.11 Outer Joins in Oracle SQL

The **LEFT JOIN**, **RIGHT JOIN** and **FULL JOIN** syntax is not recognised in Oracle SQL so the standard, SQL/92 method of declaring outer joins is not possible.

Oracle provides an alternative syntax for an outer join, a (+) is put on the **WHERE** join condition.

Eg. The Oracle command:

```sql
SELECT EMPNO, ENAME, DEPT.DEPTNO "DEPT", DNAME, LOCATION
FROM EMP, DEPT
WHERE EMP.DEPTNO(+)= DEPT.DEPTNO;
```

This extends the marked table, **EMP**, with imaginary blank entries that will then correspond to all unmatched entries in the joined table, **DEPT**.

.... ie. The above is equivalent to a **RIGHT JOIN**.

(Note **DEPTNO** is specified to be from the **DEPT** table to ensure it is output on any "extended" entries)

In the above example if the **WHERE** condition was specified as:

```sql
WHERE EMP.DEPTNO = DEPT.DEPTNO(+);
```

.... this would have been equivalent to a **LEFT JOIN**.

A **FULL JOIN** is not possible in Oracle SQL.
13.12 **Table Alias Names**

Tables can be given alias names by which they can be referred in the `SELECT` command.

eg. 

```sql
SELECT empno, ename, De.deptno, dname, loc 
FROM emp Em INNER JOIN dept De 
ON Em.deptno = De.deptno;
```

In Access/Visual Basic SQL the "AS" keyword can optionally be used to specify the table alias name as in `emp AS Em`, but this is not essential.

Alias names may be used for the following:

1. For convenience, if a table with a particularly long name is referred to several times.

2. When a table is joined to itself, to distinguish which columns come from each use of the table.

eg. To list an employees name with the name of his manager a table of manager employee numbers and names is required ....

```sql
.... but this is available in the same EMP table.
```

In Access SQL this could be obtained by:

```sql
SELECT slave.ename AS Employee, 
       boss.ename AS Manager 
FROM emp slave INNER JOIN emp boss 
ON slave.mgr = boss.empno;
```
13.13 Example of a Table Joined to Itself

Access Command:

```
SELECT slave.ename AS Employee, 
     boss.ename AS Manager 
FROM emp slave INNER JOIN emp boss 
    ON slave.mgr = boss.Empno;
```

Output:

```
Employee   Manager
----------   -------
SMITH       FORD
ALLEN       BLAKE
WARD        BLAKE
JONES       KING
MARTIN      BLAKE
BLAKE       KING
CLARK       KING
SCOTT       JONES
TURNER      BLAKE
ADAM        SCOTT
JAMES       BLAKE
FORD        JONES
MILLER      CLARK
```

(13 records selected.)

Note that there is no entry for KING as he has no manager and the join is an inner join.

To list KING with a blank for the manager field a left outer join is required:

```
SELECT slave.ename AS Employee, 
     boss.ename AS Manager 
FROM emp AS slave LEFT JOIN emp AS boss 
    ON slave.mgr = boss.empno;
```
13.14 **Example of a Three Table Join**

Joins can be extended to include three or more tables by replacing the name of a table with a nested join of tables in brackets.

**eg.**

```sql
SELECT slave.ename AS Employee,
       boss.ename AS Manager,
       loc AS Location
FROM (emp slave INNER JOIN emp boss
       ON slave.mgr = boss.empno)
       INNER JOIN dept
       ON slave.deptno = dept.deptno;
```

**or**

```sql
SELECT slave.ename AS Employee,
       boss.ename AS Manager,
       loc AS Location
FROM dept INNER JOIN
     (emp slave INNER JOIN emp boss
      ON slave.mgr = boss.empno)
     ON slave.deptno = dept.deptno;
```

**Giving:**

<table>
<thead>
<tr>
<th>Employee</th>
<th>Manager</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMITH</td>
<td>FORD</td>
<td>DALLAS</td>
</tr>
<tr>
<td>ALLEN</td>
<td>BLAKE</td>
<td>CHICAGO</td>
</tr>
<tr>
<td>WARD</td>
<td>BLAKE</td>
<td>CHICAGO</td>
</tr>
<tr>
<td>JONES</td>
<td>KING</td>
<td>DALLAS</td>
</tr>
<tr>
<td>MARTIN</td>
<td>BLAKE</td>
<td>CHICAGO</td>
</tr>
</tbody>
</table>

....etc

**Note:** Any type of join (**INNER JOIN**, **LEFT JOIN** or **RIGHT JOIN**) can be nested inside an **INNER JOIN**.

**But:** It is *not possible* to nest any type of join inside a **LEFT JOIN** or a **RIGHT JOIN**.
CHAPTER 14

The SQL Data Maintenance Commands

<table>
<thead>
<tr>
<th>Contents</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>14.1 The SQL Command : CREATE TABLE</td>
<td>184</td>
</tr>
<tr>
<td>14.2 Date and Time Types</td>
<td>185</td>
</tr>
<tr>
<td>14.3 Additional Field Types</td>
<td>185</td>
</tr>
<tr>
<td>14.4 Creating a Table from a SELECT Command</td>
<td>186</td>
</tr>
<tr>
<td>14.5 The SQL Command : ALTER TABLE ... ADD</td>
<td>187</td>
</tr>
<tr>
<td>14.6 The Oracle Command : ALTER TABLE ... MODIFY</td>
<td>188</td>
</tr>
<tr>
<td>14.7 The SQL Command : ALTER TABLE ... DROP</td>
<td>189</td>
</tr>
<tr>
<td>14.8 The SQL Command : DROP TABLE</td>
<td>189</td>
</tr>
<tr>
<td>14.9 The SQL Command : INSERT</td>
<td>190</td>
</tr>
<tr>
<td>14.10 The SQL Command : DELETE</td>
<td>191</td>
</tr>
<tr>
<td>14.11 The SQL Command : UPDATE</td>
<td>192</td>
</tr>
<tr>
<td>14.12 Making Data Maintenance Changes Permanent</td>
<td>193</td>
</tr>
</tbody>
</table>
14.1 **The SQL Command: CREATE TABLE**

New tables are created with the **CREATE TABLE** command. This command creates an empty table and defines its structure.

The general syntax is:

```
CREATE TABLE table-name (column-details);
```

Where the column-details are a list of column names with the column types.

**eg.**

```
CREATE TABLE PERSON (  
  ID_NUM  INTEGER,  
  TITLE   VARCHAR(4),  
  NAME    CHAR(10),  
  DOB     DATE,  
  NOTES   MEMO  
);  
```

Possible column types are:

- **CHAR(n)** Fixed length string of n characters will be padded with spaces (n <= 255)
- **VARCHAR(n)** Variable length string of n characters (n <= 255)
- **INTEGER** 32 bit integer in the range ± 2G
- **SMALLINT** 16 bit integer in the range ± 32K
- **REAL** Single precision real number
- **DOUBLE** Double precision real number
14.2 Date and Time Types

Earlier SQL standards had no date or time types. The SQL/92 standard introduces the following:

**DATE** Holds a date with a 4 digit year.

**TIME** Holds a time in hours, minutes and seconds.

**TIMESTAMP** Holds both a date and a time.

Note, however, it is not uncommon to find, as in both Oracle and Access, that DATE fields can also hold times and that the above types are identical.

14.3 Additional Field Types

Visual Basic and Access also offer the following additional field types that are commonly found in other SQL systems:

**MEMO** Variable length character string of any length. Only one field of this type is normally possible in a table. This is known as type LONG in Oracle.

**BYTE** An integer in the range -128 to +127.

**MONEY** A real number with fixed, two digit decimal places.

**BOOLEAN** Holds 'True' or 'False'
14.4 Creating a Table from a SELECT Command

Although not part of the SQL/92 standard, most database management systems allow a table to be created from a SELECT command.

This enables the copying of part or all of one table to a new table.

Examples are:

Oracle has a CREATE TABLE ... AS command with syntax:

```
CREATE TABLE table-name (column-names)
    AS select-command;
```

Access/Visual Basic has a SELECT INTO command with syntax:

```
SELECT field1,field2...fieldn
    INTO newtablename
    FROM ..... 
    etc.
```

The column names and types for the new table are the same as the data selected.

Note that if the table already exists, its format and data are completely overwritten.
14.5 The SQL Command: ALTER TABLE . . . ADD

Once a table has been created it is possible to add an extra column to the table using the SQL command:

```
ALTER TABLE table-name ADD column-details;
```

The new columns are added to the existing columns in the table.

All rows that already exist in the table will have null values created for the new columns . . .

. . . these may then be altered using the UPDATE command if required.

The column details are in the same format as in the CREATE TABLE command.

Examples:

```
ALTER TABLE EMP ADD FIRST_NAME VARCHAR(10);
ALTER TABLE DEPT ADD RENT MONEY;
```

Note: Both the CREATE TABLE and the ALTER TABLE commands can be used to create or modify table constraints as well as columns ....

.... constraints are discussed later under database integrity issues.
14.6 The Oracle Command:

**ALTER TABLE . . . MODIFY**

A non standard Oracle command allows changes to the format of existing columns of a table. This is possible even though there are existing rows of data.

The syntax is:

```
ALTER TABLE table-name MODIFY
(column-details);
```

Columns with existing entries cannot change type or be reduced in width - they can only be made wider.

ie. Only if all rows have null entries for a column can that column change type or be reduced in width.

The ( ) round the column-details are not necessary for single columns.

Examples:

```
ALTER TABLE EMP MODIFY ENAME CHAR(12);
```

```
ALTER TABLE DEPT MODIFY
( RENT NUMBER(5),
  OWNER CHAR(20) );
```

Note: In Oracle SQL it is not possible to alter a table to remove columns.
14.7 **The SQL Command:**

**ALTER TABLE . . . DROP**

The **ALTER TABLE** command can also remove a column and any data it contains from a table.

The syntax is:

```
ALTER TABLE table-name DROP column-name;
```

Example: `ALTER TABLE EMP DROP COMM;`

Note: Although it is part of the SQL/92 standard the **ALTER TABLE . . . DROP** command is not available in Oracle SQL.

14.8 **The SQL Command:** **DROP TABLE**

A whole table is deleted using the SQL command: **DROP TABLE**

The syntax is:

```
DROP TABLE table-name;
```

Example: `DROP TABLE DEPT;`
14.9 The SQL Command: INSERT

Once a table has been created, new rows of data can be added to it with the INSERT command with syntax:

\[
\text{INSERT INTO table-name (columns) VALUES (value-list);}
\]

or

\[
\text{INSERT INTO table-name (columns) select-command;}
\]

- The (columns) are optional, if omitted all columns are assumed.
- If the columns are named then any unmentioned columns will be set to NULL.
- The first format is used to insert one record at a time into the table.
- The second format is used to copy data from one table to add it to another.

Examples:

\[
\text{INSERT INTO dept VALUES (50,'FOREIGN','LONDON');}
\]

\[
\text{INSERT INTO emp (empno,ename,hiredate) VALUES (1234,'SANTA','25-DEC-95');}
\]

\[
\text{INSERT INTO My_Table (Name,Sal,Department) SELECT ename,sal,dname FROM emp INNER JOIN dept ON emp.deptno = dept.deptno; WHERE job='CLERK';}
\]
14.10 **The SQL Command: DELETE**

This command is used to delete rows from a table.

The general syntax is:

```sql
DELETE FROM table-name
    WHERE conditions;
```

Rows are deleted that match the conditions.

**Warning:**

If there is no **WHERE** clause **all rows are deleted**, the table will still exist but it will be empty.

**Examples:**

```sql
DELETE FROM emp;

DELETE FROM emp WHERE ename = 'WARD';

DELETE FROM emp
    WHERE comm IS NOT NULL
    OR     ename LIKE 'A%';
```
14.11 The SQL Command : \textbf{UPDATE}

This command is used to alter one or more columns in one or more rows of a table.

The general syntax is:

\begin{verbatim}
UPDATE table-name
    SET column-assignments
WHERE conditions;
\end{verbatim}

The column-assignments are a list of assignments separated by commas where each is in the form:

\begin{verbatim}
column-name = new-value
or \hspace{1cm} column-name = NULL
\end{verbatim}

Warning:

If no \texttt{WHERE} clause is specified all rows are updated.

Examples:

\begin{verbatim}
UPDATE emp SET sal = sal*1.1;

UPDATE emp
    SET job = 'MANAGER',
        sal = 3000,
        comm = NULL,
        deptno = 40
WHERE ename = 'FORD';
\end{verbatim}
14.12 **Making Data Maintenance Changes**

**Permanent**

Changes made to a data base with **INSERT, DELETE or UPDATE** are all stored temporarily . . .

. . . they do not permanently alter the data!

To become permanent the following SQL command must be given:

```
COMMIT;
```

It is also possible to abandon all changes made in one session since the last **COMMIT** with the command:

```
ROLLBACK;
```

Some commands perform an implicit **COMMIT** command making all previous changes permanent.

These commands are: **CREATE, ALTER or DROP GRANT, REVOKE OR EXIT**

**N.B. The COMMIT and ROLLBACK commands do not exist in Visual Basic/Access SQL.**

From a Visual Basic program each command is immediately committed automatically unless the **BeginTrans database** function is used ....

.... in which case the database functions **CommitTrans** and **Rollback** provide the functionality of the SQL **COMMIT** and **ROLLBACK** commands.
CHAPTER 15

SQL Subqueries

Contents

15.1 Using a SELECT Subquery 195
15.2 SELECT Commands With SELECT Subqueries 196
15.3 Subqueries With ANY, SOME or ALL 197
15.4 Subqueries With IN and NOT IN 198
15.5 The WHERE EXISTS Subquery 199
15.6 Subqueries Needing a Table Alias Name 200
15.7 Different Ways of Making a Subquery 201
15.8 Combining Selections With the UNION Operator 202
15.9 Example Output From a UNION 203
15.10 Distinct Row Output With a UNION 204
15.11 Column Headings With a UNION 204
15.12 Ordering the Output With a UNION 205
15.13 Restrictions On The Use Of a UNION 205
15.14 The INTERSECT and EXCEPT Operators 206
15.15 Oracle and Access Implementations of the UNION, INTERSECT and EXCEPT Operators 207
15.1 Using a SELECT Subquery

The `SELECT` command can appear as part of other commands.

An example is the `INSERT` command:

```
INSERT INTO mytable (IDnumber,name)
SELECT empno,ename FROM emp
WHERE job='CLERK';
```

The `SELECT` command can, in fact, appear in any command with a `WHERE` clause such as `DELETE` or even `SELECT` itself:

```
DELETE FROM emp
WHERE deptno =
 (SELECT deptno FROM dept
  WHERE dname='ACCOUNTING');
```

This example will delete all records from the `EMP` table of employees working in the accounts department.

In an example of this type it is important that the select command retrieves just one value, or an error will occur. . .

. . . It would not make any sense to have more than one value in such a comparison anyway.
15.2 SELECT Commands With SELECT Subqueries

The WHERE clause on the SELECT command can also contain a SELECT subquery.

Eg. To find all employees with a salary greater than Scott use:

```sql
SELECT ename, sal FROM emp
WHERE sal > (SELECT sal FROM emp
            WHERE ename = 'SCOTT');
```

It is possible for the subquery to have further subqueries nested to any depth.

Eg. To find all employees with a salary greater than the manager of the accounts department:

```sql
SELECT ename, sal FROM emp
WHERE sal >
    (SELECT sal FROM emp
     WHERE job = 'MANAGER'
     AND deptno =
        (SELECT deptno FROM dept
         WHERE dname = 'ACCOUNTING'));
```

Note that the subquery can select from either the same table or a different table to the main SELECT command.
15.3 Subqueries With ANY, SOME or ALL

It is possible to use subqueries that may give multiple values if ANY, SOME or ALL is used:

Example of a subquery used with ANY or SOME:

To list all employees earning as much in salary as a manager:

```
SELECT ename, sal FROM emp
WHERE sal >= ANY
  (SELECT sal FROM emp
   WHERE job='MANAGER');
```

In this example, the WHERE clause will be true if the subquery search finds any manager's salary less than or equal to the salary of the record currently being examined.

SOME is simply an alternative word for ANY and has the same meaning.

Example of subquery used with ALL:

To list the employees with the minimum salary:

```
SELECT ename, sal FROM emp
WHERE sal <= ALL
  (SELECT sal FROM emp);
```

This WHERE condition is true only if the salary of the record currently being examined is less than or equal to every value selected in the subquery.
15.4 **Subqueries With IN and NOT IN**

An alternative form for a multi-valued subquery is to use the IN or NOT IN conditions:

**Eg1. To list every department that employs a clerk:**

```sql
SELECT dname FROM dept
WHERE deptno IN
  (SELECT deptno FROM emp
   WHERE job='CLERK');
```

Note: IN (subquery) is another way of writing = ANY(subquery).

**Eg2. To list every department that has no salesmen:**

```sql
SELECT dname FROM dept
WHERE deptno NOT IN
  (SELECT deptno FROM emp
   WHERE job='SALESMAN');
```

N.B. This would not have been achieved with:

```sql
SELECT dname FROM dept
WHERE deptno IN
  (SELECT deptno FROM emp
   WHERE job!='SALESMAN');
```

... as the sales department would be selected as it has some employees that are not salesmen.

Note: NOT IN (subquery) is another way of writing != ALL(subquery).
15.5 The *WHERE EXISTS* Subquery

It is sometimes more convenient or natural to use a *WHERE EXISTS* or *WHERE NOT EXISTS* subquery.

In the *WHERE EXISTS* subquery if any matching record is found the result is true, so the column specification in the subquery is irrelevant.

Similarly the result is false if any matching record for a *WHERE NOT EXISTS* subquery is found.

Eg. To list every department with employees recorded in the *EMP* table:

```
SELECT dname FROM dept
WHERE EXISTS
  (SELECT * FROM emp
   WHERE emp.deptno=dept.deptno);
```

This could also have been achieved using:

```
SELECT dname FROM dept
WHERE deptno IN
  (SELECT deptno FROM emp);
```

Similarly to list departments that have no recorded employees:

```
SELECT dname FROM dept
WHERE EXISTS
  (SELECT * FROM emp
   WHERE emp.deptno=dept.deptno);
```

This also has a *NOT IN* equivalent.
15.6 Subqueries Needing a Table Alias Name

If a subquery selects from the same table as the main query it may be necessary to use a table alias name.

Eg. To list all employees who manage other employees:

```sql
SELECT ename FROM emp AS boss
WHERE NOT EXISTS
  (SELECT * FROM emp
   WHERE emp.mgr = boss.empno);
```

In this example if the alias "boss" was not used the subquery would look for all employees who were their own managers!
15.7 Different Ways of Making a Subquery

A selection using a subquery may be done in different ways.

Eg. The following are equivalent:

```sql
SELECT ename FROM emp boss
WHERE EXISTS
    (SELECT * FROM emp
     WHERE emp.mgr=boss.empno);
```

```sql
SELECT ename FROM emp
WHERE empno = ANY
    (SELECT mgr FROM emp);
```

```sql
SELECT ename FROM emp boss
WHERE empno IN
    (SELECT mgr FROM emp);
```

... and frequently the same information can be obtained just as easily using a table join:

```sql
SELECT DISTINCT boss.ename
    FROM emp boss INNER JOIN emp slave
    ON slave.mgr = boss.empno;
```

On large tables the different methods may give significant differences in speed and efficiency.

This is a disadvantage of SQL as it is confusing . . . .

. . . users are unlikely to know which is more efficient.

Usually a WHERE EXISTS or WHERE NOT EXISTS subquery is more efficient than other methods.
15.8 Combining Selections With the **UNION** Operator

It is sometimes necessary to select information where each column of output may come from more than one table.

Eg. Suppose a company's database had tables:

- CUST with columns CNAME for a customer's name and CADDR for the customer's address
- SUPP with columns SNAME for a supplier's name and SADDR for the supplier's address

To list all outside organisations that the company deals with would require a selection from each table . . .

. . . but because a supplier may also be a customer for a different product, listing both tables with two `SELECT` commands would give some names twice.

To give the required selection the **UNION** operator is used to combine `SELECT` commands:

```sql
SELECT cname AS Name, caddress AS Address
FROM cust
UNION
SELECT sname, saddress
FROM supp;
```
15.9 Example Output From a UNION

The command:

```
SELECT ename, empno FROM emp
UNION
SELECT dname, deptno FROM dept;
```

Gives:

<table>
<thead>
<tr>
<th>ENAME</th>
<th>EMPNO</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACCOUNTING</td>
<td>10</td>
</tr>
<tr>
<td>ADAMS</td>
<td>7876</td>
</tr>
<tr>
<td>ALLEN</td>
<td>7499</td>
</tr>
<tr>
<td>BLAKE</td>
<td>7698</td>
</tr>
<tr>
<td>CLARK</td>
<td>7782</td>
</tr>
<tr>
<td>FORD</td>
<td>7902</td>
</tr>
<tr>
<td>JAMES</td>
<td>7900</td>
</tr>
<tr>
<td>JONES</td>
<td>7566</td>
</tr>
<tr>
<td>MARTIN</td>
<td>7654</td>
</tr>
<tr>
<td>MILLER</td>
<td>7934</td>
</tr>
<tr>
<td>OPERATIONS</td>
<td>40</td>
</tr>
<tr>
<td>RESEARCH</td>
<td>20</td>
</tr>
<tr>
<td>SALES</td>
<td>30</td>
</tr>
<tr>
<td>SCOTT</td>
<td>7788</td>
</tr>
<tr>
<td>SMITH</td>
<td>7369</td>
</tr>
<tr>
<td>TURNER</td>
<td>7844</td>
</tr>
<tr>
<td>WARD</td>
<td>7521</td>
</tr>
</tbody>
</table>

The command:

```
SELECT ename, job, deptno FROM emp
WHERE job = 'CLERK'
UNION
SELECT ename, job, deptno FROM emp
WHERE deptno = 10;
```

Gives:

<table>
<thead>
<tr>
<th>ENAME</th>
<th>JOB</th>
<th>DEPTNO</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADAMS</td>
<td>CLERK</td>
<td>20</td>
</tr>
<tr>
<td>CLARK</td>
<td>MANAGER</td>
<td>10</td>
</tr>
<tr>
<td>JAMES</td>
<td>CLERK</td>
<td>30</td>
</tr>
<tr>
<td>KING</td>
<td>PRESIDENT</td>
<td>10</td>
</tr>
<tr>
<td>MILLER</td>
<td>CLERK</td>
<td>10</td>
</tr>
<tr>
<td>SMITH</td>
<td>CLERK</td>
<td>20</td>
</tr>
</tbody>
</table>
15.10 Distinct Row Output With a UNION

A UNION implies a SELECT DISTINCT on the resulting output, any rows of output that are duplicated are eliminated.

ie. A row will be output only once if it is generated from both the SELECT commands.

The output will list duplicates if UNION ALL is used.

eg.  

```
SELECT ename,job,deptno FROM emp
  WHERE job = 'CLERK'
UNION ALL
SELECT ename,job,deptno FROM emp
  WHERE deptno = 10;
```

would cause the row

```
MILLER    CLERK     10
```

to be output twice.

15.11 Column Headings With a UNION

If two columns with different names are combined using the UNION operator, it is the name in the first SELECT that is used in the column header.

The header used is the only effect of changing the order in which the SELECT commands are specified...

    .... there is no difference in the rows displayed.

Any change of header using AS must be specified on the first SELECT command or it will be ignored.
15.12 Ordering the Output With a UNION

The UNION output is automatically ordered on the first column displayed unless this is changed with an ORDER BY clause.

It is not possible to specify individual ORDER BY clauses for each SELECT with a UNION operator but...

...a single ORDER BY clause can be used if it is at the end of the whole command.

eg. SELECT cname, caddress FROM cust WHERE caddress LIKE '*Loughborough*' UNION SELECT sname, saddress FROM supp ORDER BY cname DESC;

Note that the ORDER BY clause must refer to the column or columns in the first SELECT command.

15.13 Restrictions On The Use Of a UNION

1. The same number of columns must be selected from each SELECT command.

2. Corresponding columns of each SELECT command must be of the same type, though it is not necessary for the columns to be the same length.

3. Columns of type MEMO cannot be included.

There is no restriction on the number of SELECT commands that may be combined with UNION operators.
15.14 The \texttt{INTERSECT} and \texttt{EXCEPT} Operators

Like the \texttt{UNION} operator, the \texttt{INTERSECT} and \texttt{EXCEPT} operators operate on selections from more than one table and have the same restrictions and ordering.

Eg. To list all organisations that are both a supplier and a customer use \texttt{INTERSECT}:

\begin{verbatim}
SELECT cname "Name", caddress "Address"
  FROM cust 
INTERSECT 
SELECT sname, saddress 
  FROM supp;
\end{verbatim}

Eg2. To list all suppliers that are not also customers use \texttt{EXCEPT}:

\begin{verbatim}
SELECT sname "Name", saddress "Address"
  FROM supp 
EXCEPT 
SELECT cname, caddress 
  FROM cust;
\end{verbatim}

With \texttt{UNION} and \texttt{INTERSECT} it does not matter which \texttt{SELECT} is before or after the operator other than the effect on the column header.

For the \texttt{EXCEPT} operator the order is significant:

eg. \begin{verbatim}
SELECT cname "Name", caddress "Address"
  FROM cust 
EXCEPT 
SELECT sname, saddress 
  FROM supp;
\end{verbatim}

This will select all customers who are not suppliers.
15.15 **Oracle and Access Implementations of the UNION, INTERSECT and EXCEPT Operators**

- Both Oracle and Access/Visual Basic implement the Union operator as described.

- Oracle implements the INTERSECT operator but Access/Visual Basic does not.

- Oracle does not implement the EXCEPT operator but does implement a MINUS operator that is identical in operation.

- Access/Visual Basic does not implement the EXCEPT operator and does not have any equivalent.

- In Access/Visual Basic the UNION operator can also be used to combine the output of two TABLE commands or a TABLE command with a SELECT Command.

  Eg. `SELECT cname AS [Name],
      caddress AS [Address]
      FROM cust
      WHERE caddress LIKE '*Loughborough'*
      UNION
      TABLE supp;`
CHAPTER 16

Views

Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>16.1</td>
<td>What is a View?</td>
<td>209</td>
</tr>
<tr>
<td>16.2</td>
<td>How is a View Created Using SQL?</td>
<td>210</td>
</tr>
<tr>
<td>16.3</td>
<td>Why Create a View?</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(1) Security</td>
<td>211</td>
</tr>
<tr>
<td>16.4</td>
<td>Why Create a View?</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(2) Convenient Access to Joined Tables</td>
<td>212</td>
</tr>
<tr>
<td>16.5</td>
<td>Why Create a View?</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(3) Saving Complex Expressions</td>
<td>213</td>
</tr>
<tr>
<td>16.6</td>
<td>Why Create a View?</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(4) Allowing Different Table Formats</td>
<td>214</td>
</tr>
<tr>
<td>16.7</td>
<td>Why Create a View?</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(5) &quot;Tables&quot; of Summary Information</td>
<td>215</td>
</tr>
<tr>
<td>16.8</td>
<td>Why Create a View?</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(6) Logical Data Independence</td>
<td>216</td>
</tr>
<tr>
<td>16.9</td>
<td>Data Updates Through a View</td>
<td>217</td>
</tr>
<tr>
<td>16.10</td>
<td>Deleting a View</td>
<td>217</td>
</tr>
</tbody>
</table>
16.1 What is a View?

A view is a "virtual table".

ie. A user can examine the data it contains as though it was a real table.

In reality it is a "window" onto one or more underlying base tables selecting and arranging the information in its own presentation.

The display of the information in a view is generated at the time it is viewed ....

.... not at the time the view is defined.

Any change to any base table data automatically changes the corresponding information displayed in any associated view.

Many Database Management Systems, such as Oracle, have the capability of creating these views and it is a standard facility of SQL systems.

N.B.: A surprising omission of Access/Visual Basic SQL is that views are NOT implement. However, the Access "QueryDef" (which has to be created using Access facilities other than Access SQL) has similar properties to a view.
16.2 How is a View Created Using SQL?

By using the CREATE VIEW command in the form:

```
CREATE VIEW view-name (column-names)
   AS select command;
```

The select command in a view is identical to the usual SQL SELECT command with any clause except the ORDER BY clause.

The list of column names in ( ) can give new names to the columns in the view ....

   .... they can be omitted, if they are the same names as the base table columns are taken.

Example of use of a view:

```
CREATE VIEW earnings (name,income)
   AS SELECT ename,sal+NVL(comm,0)
            FROM emp,dept
            WHERE emp.deptno = dept.deptno;
```

(NVL is as Oracle function used here to assign a value of zero to all null fields in the comm column)

Example of a view with grouped data:

```
CREATE VIEW totals (deptno,sal,comm)
   AS SELECT deptno,SUM(sal),SUM(comm)
            FROM emp
            GROUP BY deptno;
```
16.3 Why Create a View?

(1) Security

A view can restrict a user's access to a subset of the data in a table .... .... users may be given access to a view where there is no access to the base table(s).

Example 1: Restricted access to columns.

A library data base may be available to public examination to allow users to see:

1. Which books are available
2. Which books are out on loan
   but not 3. Who the books are on loan to

Example 2: Restricted access to rows:

Members of a university department may have access to inspect the university's accounts .... .... but this may be restricted through a view so that only the transactions of that department can be displayed.
16.4 **Why Create a View?**

**(2) Convenient Access to Joined Tables**

When there is a need for regularly displaying data from two different, joined tables a view can create a permanent, virtual, composite table.

**Example**

There may be a regular need for details of a company's employees along with the details of their department ....

.... if this information is stored in two different tables such as the example "emp" and "dept" tables they could be combined into one by using:

```sql
CREATE VIEW mix_data AS
    SELECT number, ename, job, dname, place
    FROM emp, dept
    WHERE emp.deptno = dept.deptno;
```
16.5 Why Create a View?

(3) Saving Complex Expressions

There may be a requirement to regularly display complex calculated column expression values.

A view gives a means of "storing" the results of these expressions for easy access without using any storage space ....

.... as the storage is really only an illusion.

Example

The total income of each employee in the "emp" table could be saved in the following view:

```
CREATE VIEW emp_inc AS
    SELECT *, sal+NVL(comm,0) FROM emp;
```

(NVL is as Oracle function used here to assign a value of zero to all null fields in the comm column)
16.6 Why Create a View?

(4) Allowing Different Table Formats

A view can have a different, more convenient, format from that of the base table.

Example 1: Different order of columns

A table may have a column for a person's initials and a column for the surname. Different users may require output in:

1. "Telephone directory" format of: Dawson R.J.
2. The conventional name order of: R.J. Dawson

If suitable views are created these could both be output by
SELECT * FROM view-name;

Example 2: Different column formats

In Oracle there is a view called USER_VIEWS gives an output usually longer than one line if all columns are selected....

.... it may be more convenient to create a new view that truncates all columns to, say, half their length so that they fit on one line.

This would allow a quick look at the table through the new view in a format that would be easy to read and would serve most purposes....

.... however, where some important details are required that have been cut by the truncation, the full details are still available by accessing the original USER_VIEWS table.
16.7 Why Create a View?

(5) "Tables" of Summary Information

A view is a convenient way of "storing" grouped data such as totals, averages etc. using group functions - without actually using storage space.

Many database management systems have application generation facilities that will build default forms to handle the data in a single table ....

.... the use of a view based on group data may enable a default form to be similarly built to handle data from group summary data from a table.

Example

A company accounting data base may need a view to show numbers of transactions, totals and balances for each department.

This virtual table would be produced by a command such as:

`CREATE VIEW summaries (deptname,transactions,balance) AS SELECT deptname,COUNT(*),SUM(trans) FROM dept_accounts GROUP BY deptname;`
16.8 Why Create a View?

(6) Logical Data Independence

A view can be used to make a selection of command procedures and forms independent of the data format.

This is often introduced by necessity if there is a change in base table format as it may:

1. Save retraining users
2. Reduce the need to change any program that accesses the data.

Example

The EMP table could at one time have stored only the total income for each employee ....

.... later changes in circumstances may have created the necessity to split the income into its salary and commission components.

For this change it is possible to overcome the need to change any existing command files by:

1. Naming the new format EMP table as something other than EMP.
2. Creating a view called EMP on this new table that recreates the old emp format.
16.9  **Data Updates Through a View**

It is possible to insert or update data though a view in some circumstances.

If this is done it is the *base tables* that are really altered.

Rows can be updated providing:

1. The view is made up from one base table only.
2. None of the columns are derived from an expression.
3. The query to create the view contained no group function, no `DISTINCT` clause or no `GROUP BY` clause.

New rows can be inserted if all of the above applies and also if all columns constrained to be not null are included in the view.

*But note:* Rows can never be deleted from a view.

16.10  **Deleting a View**

Views can be discarded when they are no longer needed using the SQL command `DROP VIEW`.

**Eg.**  
```
DROP VIEW my_view;
```
CHAPTER 17

Performance of Relational Database Management Systems

Contents

17.1 Why Are Relational Databases Slow? 219
17.2 Improving Access Time By De-Normalisation 220
17.3 Example 1: De-Normalisation by Adding Fields that are Not Dependent on the Key 221
17.4 Example 2: De-Normalisation by Repeating Fields 222
17.5 Improving Access Time by Using Indexes 223
17.6 How is an Index Used? 224
17.7 Maintaining Indexes 225
17.8 Implications of Index Maintenance 225
17.9 Different Types of Index 226
17.10 Creating an Index in SQL 227
17.11 Improving Access Time by Optimising Queries 228
17.12 The Oracle Query Optimiser 229
17.13 Optimising Join Queries 230
17.14 Join Query 'Search' and 'Driving' Tables 231
17.15 Correct Use of the SQL SELECT Command HAVING and WHERE clauses 232
17.16 Improving Access Using Multiple Disks 233
17.17 Improving Access Time With a Single Disk 233
17.18 Improving Access Time by Maximising Memory Use 234
17.1 Why Are Relational Databases Slow?

Relational database management systems are relatively easy to use . . .

. . . but they can be very slow in their operation.

Why?

1. The input commands have to be interpreted every time an instruction is given.

2. Data is stored in no particular order, this means that searches and sorts are repeated every time a command requires it.

3. Table joins are re-created for every instruction using the join - even joins made through a view. This involves a search through one table for a match for every record in the other.

4. Often no advantage can be made of available memory to store temporary tables, etc. as the system always uses disk storage.

5. The software is general purpose, no advantage is taken of particular features of an application for optimisation:

eg. A search may only ever produce one record in a particular application - but the search will continue to the end of the table regardless.
17.2 Improving Access Time By De-Normalisation

Normalisation of data produces an efficient storage structure .... but it may not produce the most efficient structure for speed of access to the data.

In particular:

Displaying data from two joined tables requires stepping through one table and, for each record to be displayed, searching for a match in the other table....

.... this is very slow if many records are displayed!

Depending on the proportion of times the table will be accessed in this way it may be worth de-normalising by merging the tables.

Merging tables can be done in two ways:

1. Adding fields that are not dependent on the key if the resulting table has only a limited number of records.

2. Repeating fields if there is a limited number of values in a multi-value dependency.

N.B. Remember! This is not just a trade off of speed verses storage .... .... it will also make the data more prone to errors of inconsistency!
17.3 **Example 1: De-Normalisation by Adding Fields that are Not Dependent on the Key**

If some normalised data gave the following tables:

Worker Table: \( \text{WorkerID} \rightarrow \text{Title, Name, DoB, Grade} \)

Wages Table: \( \text{Grade} \rightarrow \text{Wage} \)

If there are only 100 workers, the extra storage required to add the \text{Wage} field to the Worker table is negligible.

This would give fields:

\( \text{WorkerID} \rightarrow \text{Title, Name, DoB, Grade, Wage} \)

This now allows faster access to a workers wage....

but ....

- If there were only half a dozen grades the wages for each grade would be stored many times over.

ie. Each time a wage for a grade was altered it would need to be changed in the table many times.

*This is an obvious source of potential error!*
17.4 Example 2: De-Normalisation by Repeating Fields

If some normalised data gave the following tables:

Worker Table: \( \text{WorkerID} \rightarrow \text{Title}, \text{Name}, \text{DoB}, \text{Grade} \)
Training Table: \( \text{WorkerID} + \text{Qualification} \)

If every worker will never have more than three qualifications, the Qualification field could be repeated in the Worker table to give:

\[
\text{WorkerID} \rightarrow \text{Title}, \text{Name}, \text{DoB}, \text{Grade}, \text{Qualification1}, \text{Qualification2}, \text{Qualification3}
\]

This will allow faster access to a worker's qualifications....

but ....

- It will waste space with blank fields if some workers have less than three qualifications.
- It may be more awkward to handle.

Eg. Any search for a particular qualification would require searching all three qualification fields.

Another obvious source of potential error!
17.5 **Improving Access Time by Using Indexes**

**What is an Index?**

An index is formed on one or more columns in a table.

It is a table of row identifiers in the order of the given columns (the exact format is DBMS dependent).

**Why use an index?**

Once an index is made it is much faster to:

1. Sort the table on the given column(s).
2. Search the table for a particular column value - particularly significant for joins.
3. Select records using conditional clauses such as `field_value < constant_value`.

ie. The use of an index can greatly speed up a display command, especially for tables joined on the given column(s).

**But is there any overhead?**

1. Extra storage is required for the index equivalent to an extra column in the table.
2. Extra disk access means an index will only improve the speed of a display if less than 25% (approx) of the records are selected for display.
17.6 **How is an Index Used?**

On many database management systems the user need take no special action to use an index....

*The D.B.M.S. will use an index automatically where appropriate.*

However an index may not be used if:

1. The index column is used in an expression
2. The column is tested to be **NULL** or **NOT NULL**

Sometimes it is possible to re-format a conditional clause so that an index may be used where its use was previously prevented... 

... or vice-versa if it is likely that more than 25% of the records will be selected.

For example, if the column 'number' was indexed the index would normally be used in the expression:

```
WHERE number > 200/1.15
```

but it may not be in:  
```
WHERE number*1.15 > 200
```

and the index would normally be used in:

```
WHERE number > 0
```

but it may not be in:  
```
WHERE number IS NOT NULL
```

*It is up to the designer of database applications to know the behaviour of the system used.*
17.7 **Maintaining Indexes**

Whenever any insert, update or delete is done, it is necessary to update all the indexes for that table. ....

.... *most D.B.M.S.s do this automatically.*

ie. Once an index is created the user can usually forget about it ....

.... the D.B.M.S. will do all that is necessary to use and maintain it.

17.8 **Implications of Index Maintenance**

Keeping an index up to date takes some processing time, every time any alteration is made to the table.

ie. In order to decide whether to create an index the balance must be considered:

Frequency of sorting or selecting on the column(s)  \[ \text{ } \]  Frequency of adding to or modifying the table
17.9 **Different Types of Index**

**Unique Indexes**

SQL allows an index to be specified as "unique".

A unique index is one where every entry in the column or combination of columns is unique.

... a unique index can be used more effectively for table searching, speeding up a search operation.

A unique index also helps preserve the integrity of a column by preventing any error of the same value appearing twice.

**Additional Non Standard Index Types**

1. **Indexes in Oracle**
   
   These can be "compressed", which takes less disk space, or "non compressed", which takes more space but allows faster searches.

   This can help to further improve performance.

2. **Indexes in Visual Basic/Access**
   
   These can prevent a column having null entries, can specify a column as a "primary" key or as a "foreign" key.

   This can help to secure the database integrity.
17.10 Creating an Index in SQL

An index is created with the SQL command:

```
CREATE [UNIQUE] INDEX index-name
    ON table-name ( column(s) );
```

If more than one column is specified the rows are sorted in the order of the first mentioned column, then where values are the same, in the order of the second column, etc.

**Examples:**

```
CREATE INDEX nameidx ON emp(ename,deptno);
CREATE UNIQUE INDEX empindex
    ON emp(empno);
```

**How Many Indexes Can a Table Have?**

A table can have any number of indexes...

... but as each index takes storage space there may be limitations.

**How is an Index Deleted?**

The standard SQL command to delete an index is:

```
DROP INDEX index-name;
```

In Visual Basic/Access SQL, however, it is necessary to specify the table name in the DROP command:

```
DROP INDEX index-name ON table-name;
```
17.11 **Improving Access Time by Optimising Queries**

The structure and order of a query with multiple conditions as the selection criteria can effect the speed in which it can operate.

- A multiple selection criterion of the form:

  \[
  \text{WHERE condition1 AND condition2}
  \]

  will be faster if condition1 eliminates more records than condition2.

- A multiple selection criterion of the form:

  \[
  \text{WHERE condition1 OR condition2}
  \]

  will be faster if condition1 selects more records than condition2.

**But** .... if an index column is being tested:

- An indexed field should be tested before any other field.
- A unique indexed field should be tested before any non unique indexed field.

**But note:** *The D.B.M.S. may do this optimisation of indexed columns for you automatically!***
17.12 **The Oracle Query Optimiser**

Oracle will automatically try and optimise the order in which it tackles multiple queries as follows:

1. Indexed column queries come before non-indexed column queries.

2. Unique index queries come before non-unique index queries.

3. Non-compressed index queries come before compressed index queries.

4. **ROWID** queries come before column and expression queries.

5. Queries with = come before queries with a range of values.

6. Bounded range queries come before unbounded range queries.

Any ordering of queries by the user is only effective if the queries are "equal" by the above rules.
17.13 Optimising Join Queries

If there is a join of two or more tables then the join condition should be placed last.

Eg. Consider the workers example which had the following tables:

Wage Table: \textit{Grade \rightarrow Wage}

Worker Table: \textit{WorkerID \rightarrow Title, Name, DoB, Grade}

To find a worker's wage given his name the SQL command could be:

\begin{verbatim}
SELECT Name, Wage
FROM Workers, Wages
WHERE Workers.Grade = Wages.Grade
    AND Name = 'Bloggs J.';
\end{verbatim}

This would join the entire \texttt{Workers} table with the \texttt{Wages} table and then, from the result, select the joined record for the name 'Bloggs J.'.

Alternatively, it would be much faster to use:

\begin{verbatim}
SELECT Name, Wage
FROM Workers, Wages
WHERE Name = 'Bloggs J.'
    AND Workers.Grade = Wages.Grade;
\end{verbatim}

This would select the single record from the \texttt{Wages} table with name 'Bloggs J.' and then find the corresponding record in the \texttt{Wages} table.
17.14 **Join Query 'Search' and 'Driving' Tables**

When tables are joined in a query, either table can be the search table or driving table.

ie. The D.B.M.S. steps through every record in the driving table and looks for a match in the joined, search table.

The choice of which table is the driving table and which is the search table can greatly affect the speed of the query:

- If one of the tables is joined on an indexed column it is more efficient to make this the search table.
- If both tables are joined on indexed columns but only one index is unique, then this should be the column in the search table.
- If both columns are of equal type it is more efficient to make the table with the smaller number of relevant rows the driving table...
  
  ie. Reduce the number of searches to be made.

Often the order in which the tables are specified determines which is the driving table and which is the search table.

Example: 'Oracle' takes the first mentioned in the query as the search table.

But note: The D.B.M.S. may automatically optimise table joins involving an index.
17.15 **Correct Use of the SQL SELECT Command**

**HAVING and WHERE clauses**

In a query selecting group data the number of rows of output could be reduced by either a *WHERE* clause or a *HAVING* clause on the *SELECT* command.

**eg.**

```sql
SELECT deptno, SUM(sal)
FROM emp
GROUP BY deptno
HAVING deptno > 10;
```

is equivalent to:

```sql
SELECT deptno, SUM(sal)
FROM emp
WHERE deptno > 10
GROUP BY deptno;
```

But:

A *WHERE* selection reduces the size of the table by eliminating rows before the group data is calculated.

A *HAVING* clause eliminates groups only after all group data has been calculated....

.... throwing away some of the calculated data.

ie. A *WHERE* should always be used unless the condition contains a group function.
17.16  **Improving Access Using Multiple Disks**

If the database can be spread over several disks then access speed can be affected by the placing of tables and indexes on different disks.

Eg. Searching on an indexed column involves two disk accesses to obtain each record, one for the index and one for the table ....

.... faster access may be achieved by putting the index and table on different disks to minimise the movement of the disk heads.

For the same reason it may be effective to put tables that are frequently joined onto different disks.

17.17  **Improving Access Time With a Single Disk**

Some database management systems allow some control over tables storage on an individual disk.

Eg. Oracle achieves this by using of table "clusters" which store related data items that are used together in the same disk segment.

Clustering will usually improve access times, especially for tables joined on the cluster column....

and ... it may also reduce disk storage requirements

*but* ... you must know what you are doing or it can have the opposite effect!
17.18 Improving Access Time by Maximising Memory Use

Database management systems tend to be slow simply because disc space is the normal means for storing data.

Even if the hardware used has vast amounts of spare memory, this is usually ignored.

Some database management systems provide a means for using available memory, but the method is different for each system.

Eg. Oracle provides the `SET ARRAYSIZE` command to specifies how many records can be held in the host computer’s memory at one time.

The default is 20 but up to 5000 can be specified.

*Using the system's spare memory may be a simple way of improving performance if the D.B.M.S. provides this level of control.*
## CHAPTER 18

### Designing the Database for Integrity

<table>
<thead>
<tr>
<th>Contents</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>18.1 What is Database Integrity?</td>
<td>236</td>
</tr>
<tr>
<td>18.2 Examples of Human Error Damaging System Integrity</td>
<td>237</td>
</tr>
<tr>
<td>18.3 Reducing Error With Default Values</td>
<td>238</td>
</tr>
<tr>
<td>18.4 Default Field Value Example</td>
<td>239</td>
</tr>
<tr>
<td>18.5 Reducing Error by Constraining the Data</td>
<td>240</td>
</tr>
<tr>
<td>18.6 CHECK Constraints</td>
<td>241</td>
</tr>
<tr>
<td>18.7 Notes on CHECK Constraints</td>
<td>242</td>
</tr>
<tr>
<td>18.8 Constraint Names</td>
<td>243</td>
</tr>
<tr>
<td>18.9 Assertions</td>
<td>244</td>
</tr>
<tr>
<td>18.10 Reducing Error with Mandatory Fields</td>
<td>245</td>
</tr>
<tr>
<td>18.11 Access/Visual Basic Mandatory Fields</td>
<td>246</td>
</tr>
<tr>
<td>18.12 Reducing Error by Enforcing Unique Values</td>
<td>247</td>
</tr>
<tr>
<td>18.13 Adding and Deleting UNIQUE Constraints</td>
<td>248</td>
</tr>
<tr>
<td>18.14 Temporary Use of the UNIQUE Constraint</td>
<td>249</td>
</tr>
</tbody>
</table>
18.1 What is Database Integrity?

Database integrity is all about keeping the data meaningful and free from error.

Errors can come from two sources:

1. program error
2. human error

Program error can be eliminated by careful design and thorough testing ....

.... human error is harder to control!

It is possible for the database system to be designed to reduce the chances of human error:

- There may be checks built into the database management system that will automatically be used to reduce input error.

- There are further checks that can be added by the programmer when tailoring a database system to a particular user's needs.

- The data structure design itself can reduce error.

This last method of reducing error can be achieved through normalisation.
18.2 **Examples of Human Error Damaging System Integrity**

Human error can damage the system integrity when:

1. The wrong input is entered.
   Eg. Entering a new employee with an ID number that is already used for another person

2. A modification is made that is incomplete
   Eg. Deleting a employee's record without altering other tables that refer to the employees ID number.

3. An inexperienced user changes the data without realising.
   Eg. A user may try to get out of the system by typing "exit" and accidentally insert the word into a database field.

4. More than one user at a time tries to alter the same data.
   Eg. Each may read the data at the same time to find the next free ID number and then each assigns the same number to a new entity.

All of these human errors can be reduced by careful system design.
18.3 Reducing Error With Default Values

One way of reducing human error is to minimise the need for manual input of data.

Eg. By entering a pre-defined default value in a table field if no value is otherwise specified in a new record.

In SQL/92 a default value can be specified when the column is defined in the CREATE TABLE or ALTER TABLE ... ADD commands.

If no default is specified then a null value default is assumed.

The format for specifying a column default is:

```
CREATE TABLE tablename (  
  .............,  
  columnname datatype DEFAULT value,  
  .............);  
```

```
ALTER TABLE tablename ADD  
columnname datatype DEFAULT value;  
```

where the value is either a constant or NULL.

In Oracle SQL it is possible to modify the default value using the ALTER TABLE ... MODIFY command:

```
ALTER TABLE tablename MODIFY  
columnname DEFAULT value;  
```
18.4 **Default Field Value Example**

The effect of the commands:

```sql
CREATE TABLE employee (  
    empno INTEGER,  
    ename CHAR(15),  
    job CHAR(10) DEFAULT 'CLERK',  
    sal INTEGER DEFAULT 1000,  
    dept SMALLINT  
);  

ALTER TABLE employee ADD  
    comm INTEGER DEFAULT 0;  

INSERT INTO employee(empno,ename)  
    VALUES (1234,'SMITH');  

INSERT INTO employee(empno,ename,sal)  
    VALUES (1235,'JONES',1200);  
```

would be to create a new table with the following two rows of data:

<table>
<thead>
<tr>
<th>empno</th>
<th>ename</th>
<th>job</th>
<th>sal</th>
<th>dept</th>
<th>comm</th>
</tr>
</thead>
<tbody>
<tr>
<td>1234</td>
<td>SMITH</td>
<td>CLERK</td>
<td>1000</td>
<td>&lt;NULL&gt;</td>
<td>0</td>
</tr>
<tr>
<td>1234</td>
<td>JONES</td>
<td>CLERK</td>
<td>1200</td>
<td>&lt;NULL&gt;</td>
<td>0</td>
</tr>
</tbody>
</table>

N.B. The `DEFAULT` clause is NOT available in Access/Visual Basic SQL. However, an identical facility is available by setting the DefaultValuye property for the table fields.
18.5 Reducing Error by Constraining the Data

The SQL/92 standard allows constraints to applied to table data.

The constraints are used to:

1. Check specified conditions relating to the table fields are always satisfied.
2. Check specified columns always contain unique values.
3. Specify table key fields and associated foreign key fields in other tables.

The constraints come in three forms.

1. Column constraints which refer to an individual column.
2. Base table constraints which refer to more than one field in a single table.
3. Assertion constraints which refer to fields from different tables.

Note: (1) is just a special case of (2), which in turn is a special case of (3).

But... Neither Oracle or Access/Visual Basic implements constraints of type (3).
18.6 **CHECK Constraints**

A **CHECK** constraint is specified for a column in the form:

```
CREATE TABLE tablename (  
    ..................  
    columnname datatype CHECK condition,  
    ..................);  
```

Eg. 
```
CREATE TABLE person (  
    idnum SMALLINT CHECK idnum > 0,  
    title CHAR(4) CHECK title IN  
    ('MR','MRS','MISS','MS'),  
    name CHAR(20)  
);  
```

A **CHECK** constraint for the base table is specified in the form:

```
CREATE TABLE tablename (  
    columnname datatype,  
    ....................  
    CHECK condition,  
    ....................);  
```

Eg. 
```
CREATE TABLE person (  
    idnum SMALLINT,  
    title CHAR(4),  
    name CHAR(20),  
    sal INTEGER,  
    comm INTEGER,  
    CHECK title IN  
    ('MR','MRS','MISS','MS'),  
    CHECK sal+comm < 3000  
);  
```
18.7 **Notes on CHECK Constraints**

1. The constraint will prevent any row being inserted into the table or being modified such that the constraint condition is violated....

        .... the error message generated, if any, will depend on the system.

2. In the SQL/92 standard the condition can be any type of condition that can appear in a **WHERE** clause.

3. In some database management systems, such as Oracle, conditions can not involve fields from other rows in the same table or any fields from other tables....

        .... this rules out the use of any select subquery in the condition.

4. Access/Visual Basic SQL does not implement **CHECK** constraints, but the same effect can be achieved by setting the validation properties of a table record or field.

5. Unlike other types of constraint (described later) there is no significant difference between a column **CHECK** constraint and a base table **CHECK** constraint.
18.8 **Constraint Names**

Each constraint can be given a name if the `CHECK` keyword is preceded by:

```
CONSTRAINT constraintname.
```

Eg.

```
CREATE TABLE person (  
idnum SMALLINT,  
title CHAR(4)  
CONSTRAINT tcon CHECK title  
    IN ('MR','MRS','MISS','MS'),  
name CHAR(20),  
sal INTEGER,  
comm INTEGER,  
CONSTRAINT scon CHECK sal+comm < 3000 );
```

Constraints can then be dropped and base table constraints added using the `ALTER TABLE` command with a `DROP CONSTRAINT` or `ADD CONSTRAINT` clause.

Eg.

```
ALTER TABLE person DROP CONSTRAINT scon;  
ALTER TABLE person ADD CONSTRAINT scon2  
    CHECK sal+comm < 4000;
```

*In Oracle* column constraints can be modified as in:

```
ALTER TABLE person MODIFY CONSTRAINT scon  
    CHECK sal+comm < 4000;
```

In *Oracle* a constraint cannot be dropped, but it can be disabled if the `CHECK` condition is followed by the key word `DISABLE`.

18.9 Assertions

SQL/92 also offers the ability to create **CHECK** constraints that are not attached to any particular table.

This is done with the **CREATE ASSERTION** command which has the format:

```
CREATE ASSERTION assertionname
  CHECK condition;
```

As it is not in the context of any table the condition necessarily involves **SELECT** subqueries.

Eg.  
```
CREATE ASSERTION balance
  CHECK (SELECT startbalance
         FROM accountstable
         WHERE accnum=12345678)
+ (SELECT SUM(transaction)
    FROM logtable
    WHERE accnum=12345678)
= (SELECT endbalance
    FROM accountstable
    WHERE accnum=12345678);
```

The assertion can be dropped using the command:

```
DROP ASSERTION balance;
```

But .... few database management systems implement this SQL/92 facility, and neither Oracle nor Access/Visual Basic recognises the assertion commands.
18.10 Reducing Error with Mandatory Fields

In many Database Management Systems it is possible to prevent a null value being entered into a particular field ... ... this is particularly useful for key fields.

In SQL this is done by adding the words NOT NULL after the field type in the CREATE TABLE command:

Eg. CREATE TABLE Person(
    fname CHAR(20) NOT NULL,
    sname CHAR(20) NOT NULL,
    salary INTEGER,
    comm INTEGER);

These constraints can also be given names as in:

CREATE TABLE Person(
    fname CHAR(20)
    CONSTRAINT fnotnull NOT NULL,
    sname CHAR(20)
    CONSTRAINT snotnull NOT NULL,
    sal INTEGER,
    comm INTEGER);

The NOT NULL constraint is in fact a short form of writing:

    CHECK columnname IS NOT NULL

so it can be dropped or added as a base table constraint like any other CHECK constraint.
18.11 Access/Visual Basic Mandatory Fields

Access/Visual Basic SQL does not implement the\ CHECK\ constraint and, therefore, does not implement the\ NOT\ NULL\ constraint....

But .... it is still possible to define mandatory fields using an additional index facility.

In Visual Basic/Access the table must first be created with the standard\ CREATE TABLE\ command.

An index can then be formed on the column or columns concerned with the extra syntax:

```
CREATE INDEX PersIndx
    ON Person(fname,sname)
    WITH DISALLOW NULL;
```

This would prevent either the fname or the sname fields being made null when a row is inserted or modified.

The index can be either an ordinary index or a unique index as required.

It can also be useful to create and then delete an index with the "DISALLOW NULL" option as a means of checking for null fields in the table.

It is also possible to achieve the same effect as the "NOT NULL" clause in Access or Visual Basic by setting the 'Required' property on a table field.
18.12 Reducing Error by Enforcing Unique Values

SQL/92 offers a form of constraint that enforces all column entries to be unique within a table.

A single column can be made to be unique in each row by adding the keyword `UNIQUE` in the column definition:

Eg. ```
CREATE TABLE person ( 
  idnum INTEGER NOT NULL UNIQUE, 
  name  CHAR(20), 
  sal   INTEGER 
) ;
``` 

A combination of columns can be made to be unique in each row by adding the `UNIQUE` base table constraint.

```
CREATE TABLE person ( 
  fname CHAR(12) NOT NULL, 
  sname CHAR(12) NOT NULL, 
  sal   INTEGER, 
  UNIQUE(fname,sname) 
) ;
``` 

In SQL/92 specifying `UNIQUE` on a column means at most one row can have a null value for this column.

Alternatively some database systems assume:

1. The `UNIQUE` constraint also implies `NOT NULL`.
2. The `UNIQUE` constraint is not permitted unless `NOT NULL` is also specified.
18.13 Adding and Deleting UNIQUE Constraints

Like all constraints the UNIQUE constraint can be named, and in Access/Visual Basic SQL this must be done as in:

```
CREATE TABLE person (
    fname CHAR(12) NOT NULL,
    sname CHAR(12) NOT NULL,
    sal   INTEGER,
    CONSTRAINT key UNIQUE(fname,sname)
);
```

This then allows the constraints to be dropped or added with the `ALTER TABLE` command.

Eg. `ALTER TABLE person ADD CONSTRAINT key UNIQUE(fname,sname);`
```
ALTER TABLE person DROP CONSTRAINT key;
```

Most database systems implement the UNIQUE constraint by creating a UNIQUE INDEX with the same name as given to the constraint.

The above commands are therefore identical to:
```
CREATE UNIQUE INDEX key
    ON person(fname,sname);
DROP INDEX key;
```

Both Oracle and Access/Visual Basic implement the UNIQUE constraint by creating a UNIQUE index.

The error message, if any, for any attempted UNIQUE constraint violation will depend on the system.
18.14 Temporary Use of the **UNIQUE** Constraint

As the **UNIQUE** constraint is usually implemented with a unique index it has the same overhead as a unique index ....

ie. Extra disk space is needed and there is some delay on maintenance operations.

It may be useful, therefore, to create the constraint or index temporarily as a test for uniqueness....

....if the constraint or the index cannot be created then the column or columns are not unique and an error is generated....

....the constraint or index can then be dropped once the test has been made.

As a matter of good practice....

....if the database system does not provide a **UNIQUE** constraint or index facility, a search should always be made to ensure any new value for a unique field does not already exist.
## CHAPTER 19

Referential Integrity

<table>
<thead>
<tr>
<th>Contents</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>19.1 Reducing Errors by Knowing the Foreign Keys</td>
<td>251</td>
</tr>
<tr>
<td>19.2 Action To Taken When a Record Is Deleted</td>
<td>232</td>
</tr>
<tr>
<td>19.3 Master-Detail Tables</td>
<td>253</td>
</tr>
<tr>
<td>19.4 Preventing a Deletion</td>
<td>254</td>
</tr>
<tr>
<td>19.5 Action To Taken When a Key Field Is Modified</td>
<td>255</td>
</tr>
<tr>
<td>19.6 Enforcing Referential Integrity</td>
<td>256</td>
</tr>
<tr>
<td>19.7 Further Referential Constraint Syntax</td>
<td>257</td>
</tr>
<tr>
<td>19.8 Notes on Referential Constraints</td>
<td>258</td>
</tr>
<tr>
<td>19.9 The <code>ON DELETE</code> Clause</td>
<td>259</td>
</tr>
<tr>
<td>19.10 The <code>ON UPDATE</code> Clause</td>
<td>260</td>
</tr>
<tr>
<td>19.11 Oracle and Access/Visual Basic Limitations</td>
<td>260</td>
</tr>
</tbody>
</table>
19.1  **Reducing Errors by Knowing the Foreign Keys**

Whether a table structure is derived with entity relationship analysis or by normalisation, a knowledge of the key fields can help secure data integrity.

As well as its own key a table may have a field or fields that correspond to the key to another table....

.... this is called a *foreign* key.

When a record from a table is deleted or a key field is modified, it is essential to know which other tables contain the key field(s) as a foreign key....

.... **ALL** tables containing the corresponding foreign key will also need to be modified!

If these foreign key fields are left unchanged the database will lose its integrity.

If a record is to be deleted action must be taken to either:

1. Set the corresponding foreign key fields to null or default values.
2. Delete the records containing the foreign key values.
3. Prompt the user to enter another foreign key value.
19.2 Action To Taken When a Record Is Deleted

The action to be taken when a key field is deleted will depend on the tables and situation concerned.

Consider a company employee details table. The key field, the person ID, may be a foreign key in:

1. a table of previous jobs in other companies
2. a table of who drives each company car

Suppose a person record is to be deleted (he or she has left the company).

The previous job table entries for the corresponding foreign key value will only be relevant to that person.

ie. The corresponding previous job records can also be deleted.

For the company car table, the car will continue to exist, so the car record cannot be deleted!

If a car can exist without being allocated to a person (it becomes a pool car) the record in the car table can be set to null....

.... otherwise the car must be reallocated to another person.
19.3 Master-Detail Tables

For some entities, instances of that entity only exist because of an instance of another, different entity.

These are detail entities, each owing their existence to a master entity.

There are two types of master-detail relationships:

1. A detail entity table where each instance is associated with only one of the master entity.

   *Eg. A previous job table where each record corresponds to just one person entity record.*

   Instances of previous jobs are usually created when the associated person instance is created.

   Such detail table records are deleted when the corresponding master record is deleted.

2. A detail entity table where each instance may be associated with several instances of the master entity.

   *Eg. A job title table where each title may be associated with several person entity records.*

   A job title is created when a person instance is created only if the job title does not already exist.

   Such detail table records would be deleted when no corresponding master records remain.
19.4 Preventing a Deletion

Suppose the company's employee records also contain a table of who had worked on which projects in the last month.

This project table is of historical interest so the records would need to be kept ....

.... but the record would make no sense without the details in the original person record ....

.... so the person record cannot be deleted!

ie. The foreign keys in other tables may mean a record cannot be deleted!
19.5  **Action To Taken When a Key Field Is Modified**

If a record's key field is modified rather than deleted then the actions taken on corresponding foreign keys can be similar to the actions for deletion.

There is also the additional option of modifying the foreign keys in step with the original record.

Eg. Suppose the company employee record contained the key field of a company car record (rather than vice-versa as in the last example) ....

.... and suppose also the car registration number was the company car key field.

If the car is replaced, the registration number would change - and in this case the person record would need to have the same update.

Note: Modifying a tables key fields is questionable ....

.... *is it really correct to do so?*

In the above example it may be better to delete the old car's record and insert the new car as this more accurately reflects the real world action.

*To preserve integrity it may be better to prevent any modification to any key field in any table!*
19.6 **Enforcing Referential Integrity**

SQL/92 provides constraint facilities to enforce referential integrity.

ie. It ensures that every foreign key relates to a valid entry in the table it references.

In SQL/92 it is possible to specify the primary key for a table and foreign keys referring to fields in other tables.

This is done when creating the table:

Eg. `CREATE TABLE person ( idnum INTEGER PRIMARY KEY, name VARCHAR(20), deptno INTEGER REFERENCES dept(deptno) );`

It is not possible to delete a record in a table if there is any instance of a reference to it by a foreign key field in another table.

Eg. If any record in the `person` table above has a `deptno` with value 10, then the corresponding `dept` table record with `deptno` value 10 cannot be deleted.

But note - any error message generated will depend on the database system .... in Access/Visual Basic SQL **NO error message is given.**
19.7 Further Referential Constraint Syntax

The specification of primary and foreign keys can also be a base table constraint.

Eg.

```sql
CREATE TABLE person (
    fname CHAR(20),
    sname CHAR(20),
    deptno INTEGER,
    PRIMARY KEY (sname,fname),
    FOREIGN KEY (deptno)
        REFERENCES dept(deptno)
);
```

This is the only way of defining multiple field keys.

Like other constraints, referential constraints can be named and handled in the `ALTER TABLE` command.

Eg.

```sql
CREATE TABLE person (
    idnum INTEGER
        CONSTRAINT pkey PRIMARY KEY
    name CHAR(20),
    deptno INTEGER,
    CONSTRAINT fkey
        FOREIGN KEY (deptno)
        REFERENCES dept(deptno)
);
```

```sql
ALTER TABLE person DROP CONSTRAINT fkey;
```

```sql
ALTER TABLE person ADD
    CONSTRAINT newkey
        FOREIGN KEY(name)
        REFERENCES employee(ename);
```

Note in Access/Visual Basic SQL all constraints **must** be named.
19.8 Notes on Referential Constraints

1. There can only be one primary key for a table but there can as many foreign keys as required. There can be any number of foreign keys in different tables referencing the same column or columns in one table.

2. A PRIMARY KEY constraint automatically makes the designated fields both UNIQUE and NOT NULL.

3. Most database systems create a unique index on the fields of the primary key constraint. The primary key can then be dropped with the DROP INDEX command.

4. Before a foreign key can be created it is necessary to either:
   • Create a primary key constraint on the referenced column(s) in the referenced table.
   • Create a unique constraint or index on the referenced column(s) in the referenced table.

5. If the column(s) in the referenced table are not specified then the primary key of the referenced table is assumed.

   Eg. CONSTRAINT fkey
       FOREIGN KEY deptno
       REFERENCES dept
19.9 The **ON DELETE** Clause

The SQL/92 default is to prevent the deletion of any row in a table if it has a field with a value referenced by a foreign key in another table.

Alternative actions can be specified, however, by adding one of the following to the foreign key definition:

- **ON DELETE CASCADE**
- **ON DELETE SET NULL**
- **ON DELETE SET DEFAULT**

Eg. 

```
CREATE TABLE person (  
  idnum INTEGER PRIMARY KEY,  
  name  CHAR(20),  
  deptno INTEGER  
  REFERENCES dept(deptno)  
  ON DELETE CASCADE  
);  
```

**CASCADE** means that when the delete is attempted on the row in the dept table referenced by the deptno field in the person table....

1. the delete in the dept table is permitted.

2. all rows in the person table that reference the deleted dept table row will also be deleted.

**SET NULL** also permits the delete in the dept table but then sets the corresponding foreign key fields in the person table to **NULL**.

**SET DEFAULT** is similar to **SET NULL** except the foreign key fields are set to their default values.
19.10 The ON UPDATE Clause

The **ON UPDATE clause** is similar to the **ON DELETE clause** except that it refers to the action taken when a referenced field is modified rather than deleted.

The SQL/92 default action is to prevent the modification, but this can be altered if any of the following are specified in the definition of the foreign key:

- **ON UPDATE CASCADE**
- **ON UPDATE SET NULL**
- **ON UPDATE SET DEFAULT**

**CASCADE** permits the update and causes the foreign keys to be updated in step with the referenced field.

**SET NULL** permits the update but then sets the corresponding foreign key fields to **NULL**.

**SET DEFAULT** is similar to **SET NULL** except the foreign key fields are set to their default value.

19.11 Oracle and Access/Visual Basic Limitations

- Oracle only implements the **ON DELETE CASCADE** option. The other **ON DELETE** options and the **ON UPDATE** options are not recognised.

- Access/Visual Basic SQL does not implement the **ON DELETE** or **ON UPDATE** options, but it is possible to achieve cascade deletes and updates by setting relation attributes.
CHAPTER 20

Designing the User Interface for Integrity

Contents

20.1 Reducing Human Error With Automatic Generation of Identifier Values ........................................ 262
20.2 Re-using Identity Numbers .............................................................................................................. 263
20.3 Automatic Generation of Master Identifiers in Detail Tables .......................................................... 264
20.4 Eliminating the Need to Remember Identifiers .............................................................................. 265
20.5 Reducing Input Error By Informing the User of Input Types ............................................................. 266
20.6 Reducing Input Error By Displaying Input Values .......................................................................... 267
20.7 Reducing Input Error By Checking Input Values ............................................................................. 268
20.8 Reducing Input Error By Restricting Input To a Limited List of Values ........................................... 269
20.9 Password Protection ......................................................................................................................... 270
20.1 Reducing Human Error With Automatic Generation of Identifier Values

It is very common to represent entities in a table with a unique identification value.

It is usually better if these identifiers are numeric.

Why?

1. Numeric values usually take up less storage space, and they are often repeated as foreign keys in other tables.

2. Comparing numeric values is usually a faster operation than comparing character strings.

3. It is easier to automatically generate the next free number when inputting a new record.

If there is an automatic generation of a new identity number the scope for human error is eliminated.

In order to automatically generate identity numbers it is common to have an extra table in the system with a name such as "identifiers".

This table will only have one record, with a field for every identifier required showing either the last used or the next free identifier value.
20.2 Re-using Identity Numbers

Question.: When a record is deleted how can an automatic system re-use the spare identifier?

Answer: It must not do so!

Such re-use is an open invitation for mistakes to happen....

.. identity numbers should remain unique for all time!

Eg. Suppose bank were in the habit of re-using an account number after someone closed their account ....

.... then one day a cheque, written by an original account number owner, is lost in the post but turns up much later ....

.... the new account owner will not be too pleased!

An identity number field may be defined to have only a limited number of digits - so it is conceivable the system will run out of numbers ....

.... but, it is far better to redefine the field to hold more digits than risk the potential confusion of re-using identity numbers.
20.3 **Automatic Generation of Master Identifiers in Detail Tables**

The generation of input "form" screens of many database management systems is based round forms that correspond to underlying tables.

Repeated attributes of an entity are normally put into a separate "detail" table which contains the key to the "master" table as a foreign key.

Eg. An employee master table may have an associated detail table of past positions held.

Input of a new "master" entity will therefore require input via several screen "forms", one for the master and one for each detail table.

To reduce human error on input the system should:

1. **Automatically** present the user with the each detail screen in turn when the master screen input is finished.

2. **Automatically** fill in the foreign master key fields in each detail table.

3. Ideally, allow the user to swap backwards and forwards between the master and detail screens.

This will allow the detail screens to be treated as an extension of the master screen.
20.4 Eliminating the Need to Remember Identifiers

Users of many database systems can be seen using a pen and paper to write down values read from the screen that they will need to enter later.

Eg. A company’s employee records contain the addresses of each person. One employee gives his name and reports a change of address.

The database system operator may have to:
1. Select a search option.
2. Search through the records to find the employee's ID number.
3. Write down the ID number.
4. Select a modify option.
5. Enter the ID number of the record to modify.
6. Change the address.

This use of pen and paper is error prone and a sure sign of a poorly designed database system.

The system should have been designed to either:
1. Allow the user to enter modify mode directly from the search mode to modify the found record.
2. Allow the employees name to be used as an alternative to entering the ID number to identify the record to modify.

However it is done, all need for operators to remember data should be eliminated.
20.5 Reducing Input Error By Informing the User of Input Types

This can be achieved in a number of ways:

1. Providing Context Sensitive Help

Help for the current input field can be made available on part of the screen, or in a separate small window,
either: *automatically*, when the field is entered
or: *on request*, with a special key combination or button.

This is particularly useful to inform the user of the input type, such as the format for date input.

2. Identifying Mandatory Input Fields

Most input forms have some fields that must be filled in for the form to make sense and other fields that may be left blank.

It is possible to prevent the user putting a blank field into the database by specifying the field as **NOT NULL**

But ... it is less confusing for the user if the form:
1. Clearly identifies mandatory fields eg. by the use of colour.
2. Prevents the user moving to another field until an entry has been made.
20.6 *Reducing Input Error By Displaying Input Values*

1. Defaults

Default input values save time and help to reduce input errors, particularly on large forms or forms that are going to be filled in many times over.

SQL/92 provides a default facility to ensure a suitable default value is always entered into the database if the field is unspecified.

But... it is also useful to display the default in the input field on the screen so that:

1. The user knows the value of the default
2. The user can choose whether to overwrite the default or not.

2. Input Verification Fields

It is common for a user to enter an identification number to select a record for modification or deletion ....

.... but numeric values are easily miss typed.

In all such cases the user should be given an immediate display of other associated information, such as a name, to allow the user to verify the input.
20.7 Reducing Input Error By Checking Input Values

1. Input Field Type Checking

The database field types will prevent a user entering a value of completely the wrong type, such as a letter in a number field.

The application builder may also allow more sophisticated checking ....

Eg. Only allowing the letters and digits in the right places when inputting a national insurance number.

It is more user friendly to notify the user of the error immediately on exit from the input field rather than when the whole form has been completed.

2. Input Field Range Checking

This facility only allows sensible values to be input into a field....

Eg. Restricting an employee age field to be in the range 16 to 65.

The SQL/92 CHECK constraint can allow such checks to be made when putting the data in the data table....

But.... It is more user friendly to notify the user of the error immediately on exit from the input field rather than when the whole form has been completed.
20.8 **Reducing Input Error By Restricting Input To a Limited List of Values**

A textual input field often has only a limited number of valid input values.

Eg. "Manager" or "Clerk" could be valid in a person's job grade field but not "Boss".

Valid input is usually restricted to:

- *either* a constant list of values.
- *or* a list of values from a data table.

The `CHECK` constraint could be used to give an error if the input does not match one of the valid values.

But .... a more user friendly facility is to provide a pop-up (or pull-down) menu to allow the user to select the required value from a list.

Such a pop-up menu could be:

- *either* automatic, popping up on entering the field.
- *or* optional, popping up when the user requests it by pressing a particular key or clicking a button.

Note: The menu facility need not be compulsory....

.... many expert typists prefer a system which allows them the option of typing in values by hand, as for them it is quicker and more accurate to do so.
20.9 Password Protection

Password protection facilities may be provided as part of a Database Management System.

This will enable an application to be built to:

(1) Restrict access to authorised users.

(2) Give different users different levels of access.

Restricting certain users or categories of users to "read only" access can prevent unintended corruption of data by inexperienced personnel.
CHAPTER 21

Multi-User Systems

Contents

21.1 Problems With Multi-User Systems 1: Inter-Related Changes 272
21.2 Packaging Changes Into Transactions With SQL 273
21.3 Problems With Multi-User Systems 2: Simultaneous Changes By Different Users 275
21.4 Database Locking Systems 276
21.5 Automatic Locking During an SQL Transaction 277
21.6 Locking With the SELECT FOR UPDATE Command 278
21.7 Access/Visual Basic Transactions 279
21.8 Whole Table Locks 280
21.9 Problems With Multi-User Systems 3: The Need for a "Snapshot" View 281
21.10 Problems With Multi-User Systems 4: Security Against Unauthorised Access 282
21.11 Database Systems Without Built-in Security 283
21.1 **Problems With Multi-User Systems 1:**

*Inter-Related Changes*

Changes made to a system often come in "packages" that only make sense taken all together.

For example, a transfer of money from one account to another will involve two database changes:

- subtracting the money from one record
- adding it to the other.

ie. Two separate SQL `UPDATE` statements.

Even if this is done automatically within a single procedure on the client computer it will still take a finite time....

.... there is still time for another user to access the data in the meantime.

This danger increases with more complicated transaction packages.

To avoid this problem SQL has a means of making all the changes appear to other users to happen altogether....

.... this is called *transaction processing.*
21.2 Packaging Changes Into Transactions With SQL

A package of changes to a database using SQL is considered to be part of a single "transaction"....

.... each transaction can consist of several inserts, updates and deletions.

The transaction starts with the first table change....

....it ends when the COMMIT command is issued, it is only then that the changes are made permanent and the actual data changes are made to the table.

Until then ....

1. The changes are only visible to the user making the changes, other users see the table in its original unaltered state.

2. The user can wipe out the changes for that transaction by issuing a ROLLBACK command.

This packaging into transactions preserves integrity by making related changes appear to happen together giving a consistent view of the data to other users.

Eg. The transfer of money from one account to another will make the two database changes appear to other users to happen instantly.
21.3  **Problems With Multi-User Systems 2:**

**Simultaneous Changes By Different Users**

A user may often read a value from the database then use that value to later change the database.

Eg. Suppose a banking system worked as follows:

First: Data is read from the database and displayed in fields on the screen
Then: The user changes the screen values
Finally: The user finishes by clicking a "save" button to update the database values.

But two bank clerks, Alice and Bert, read the same bank account total in quick succession.

Alice adds £100 to her copy.... Bert subtracts £50 from his.

Alice updates the account .... no problem.
Bert updates the account .... and wipes out the £100!

(Alternatively - if Bert had updated first then his £50 withdrawal would have been lost.)

*What is needed is for the record to be locked when it first read so that no other user can update or lock that record until the changes are complete.*
Another example:

Suppose a database program automatically provides an ID number when a new record is inserted.

The next available ID number is stored in a special database table.

First: The program reads the next unused ID number from the database and uses this as the ID number in the new record inserted.

Then: It adds 1 to the next available ID number stored in the database.

But if Alice and Bert both insert new records at approximately the same time.

Alice reads the next available ID number from the database ..... and Bert does the same.

Alice adds one to the next available ID number .... and Bert does the same.

The next available number is now 2 more than before
.... but the new records inserted by Alice and Bert have the same ID!

*Again a lock is needed from the time the next available ID number is read until it updated.*
21.4 Database Locking Systems

Most database management systems provide for some form of multi-user access, often this access is through a network.

To allow more than one user to make changes it is necessary to have an automatic system of locking out other users when any change is being made.

Different locking facilities may be provided depending on the database management system used and what action the user is taking.

How much of the system is locked can be:

- All data tables in the database system.
- A single table.
- A group of records in a table.
- A single record in a table.
- A single field of a single record in a table.

How long a table is locked can be:

- Until the user finishes accessing the system.
- While the user is working on the table.
- While the user works on the individual record.
- Until the user gives a command to "commit" the changes.

The type of locking can be to:

- Allow others "read only" access to the data.
- Prevent any access by other users to the data.
21.5 Automatic Locking During an SQL Transaction

Whenever a change is made to a table using the SQL commands **INSERT**, **UPDATE** or **DELETE** the affected rows are locked until the transaction is completed.

The extent of the lock depends on the database system.

Eg. Oracle and Access/Visual Basic both use a page locking system ....

.... which typically locks about 20 rows in the table.

This lock means that

1. Other users can only read the locked records .... they cannot change them until the transaction is completed.

2. If other users do read the locked records they always get the pre-change, pre-lock values.

The lock is released when the user issues a **COMMIT** or **ROLLBACK** command (or other command with an implicit commit such as the **CREATE TABLE** command).

This helps preserve integrity by preventing two users from changing the same data at the same time.
21.6 Locking With the `SELECT FOR UPDATE` Command

The SQL automatic lock only starts when the first change is made to the database ....

.... it does not solve the problem when changes are based on values that are first read from the database.

Many database systems, such as Oracle, extend the SQL `SELECT` command to lock rows in a table by adding the extra `FOR UPDATE` clause in the form:

```
SELECT ......
......
FOR UPDATE OF column1, column2, ... [NOWAIT];
```

eg. 
```
SELECT ename, sal, comm
FROM emp
FOR UPDATE OF sal, comm NOWAIT;
```

This starts a transaction with a lock such that only the user can then update or delete the selected rows until the transaction is completed.

The optional extra keyword `NOWAIT` prevents the system waiting while another user finishes their changes if it is already locked.

There is obviously no need to use the `SELECT FOR UPDATE` command in a single user system.
21.7 Access/Visual Basic Transactions

Transactions in Access/Visual Basic are activated in a non standard way under program control with the Visual Basic command:

\[ \text{<database_variable>}.\text{BeginTrans} \]

eg. \text{mydb}.\text{BeginTrans}

There is no \text{COMMIT} or \text{ROLLBACK} command in Visual Basic/Access SQL, instead the equivalent Visual Basic commands must be used:

ie. \[ \text{<database_variable>}.\text{CommitTrans} \]
and \[ \text{<database_variable>}.\text{Rollback} \]

eg. \text{mydb}.\text{CommitTrans} \\
\text{mydb}.\text{RollBack}

If these commands are not used Visual Basic/Access SQL \textit{automatically commits} after every command.

There is also no \texttt{SELECT FOR UPDATE} command in Visual Basic/Access SQL .... a record can be locked for the period between a \texttt{SELECT} and an \texttt{UPDATE} by:

1. Starting the transaction with the Visual Basic \texttt{BeginTrans} command.

2. Locking the record with a dummy update such as:

\begin{verbatim}
UPDATE emp SET empno=empno WHERE ename = 'SMITH';
\end{verbatim}

3. Selecting the record and performing the updates as required.
21.8 Whole Table Locks

For complex action involving a whole table, many database management systems such as Oracle, provide a non SQL/92 standard table lock which be applied with the LOCK TABLE command:

\[
\text{LOCK TABLE <tablename> IN EXCLUSIVE MODE;}
\]

Only one user can apply this lock to a table at one time. Other users can still read the data but they will see it in its pre-change state.

Other, weaker locking modes may also be available.

The lock is released when the user issues a COMMIT or ROLLBACK command (or other command with an implicit commit).

The LOCK TABLE command does not exist in Access/Visual Basic SQL.

Instead the following non standard methods are offered:

1. Whole or part tables or groups of tables may be locked using member functions of Dynaset and Table variables.

2. An entire database .MDB file can be locked using extra parameters to the OpenDatabase function.
21.9 Problems With Multi-User Systems 3: The Need for a "Snapshot" View

Sometimes it is necessary to collect data from several tables in order to prepare a detailed report or perform a complex calculation.

Eg. An auditor may wish to examine all of a company's financial data which may be spread across many tables ....

.... but updates by other users made between the auditor's examination of the different tables could completely upset the figures.

To solve this problem some databases provide the SQL command \texttt{SET TRANSACTION READ ONLY;} which

1. must be the first command in a transaction.

2. prevents any command except \texttt{SELECT} from being used in that transaction.

3. provides a snap shot of the whole system at the time of the command ....

all selects for the remainder of the transaction will show data as it was when the \texttt{SET TRANSACTION READ ONLY} command was issued.

N.B. This command is not available in Access/Visual Basic SQL .... A whole database lock would need to be used instead.
21.10 Problems With Multi-User Systems 4: Security Against Unauthorised Access

Many database management systems provide a means of giving individuals usernames.

Each table is "owned" by its creator - other users cannot access it without permission from the owner.

In SQL permission is given using the following:

```
GRANT access-list ON table-name
   TO user-list;
```

where the access-list is either `ALL` or any combination of:

- `ALTER`
- `DELETE`
- `INDEX`
- `INSERT`
- `SELECT`
- `UPDATE`
- `UPDATE(column-name)`

and the user-list can be either `PUBLIC` to give access to all users, or a list of user names.

Eg.

```
GRANT ALL ON MY_TABLE TO PUBLIC;
```

```
GRANT SELECT, INDEX, UPDATE(SAL, COMM)
   ON EMP TO OPS$corjd, OPS$corpk;
```

N.B. It is not possible to grant another user access to use the `DROP TABLE` command.

The `GRANT` command can also be extended with:

```
WITH GRANT OPTION
```
The **WITH GRANT OPTION** clause allows the other users you have granted access to use the `GRANT` command to pass the permission on to a third party:

Eg.  
```
GRANT SELECT, INDEX, UPDATE(SAL, COMM) 
    ON EMP TO OPS$corjd, OPS$corpk 
    WITH GRANT OPTION;
```

The SQL command to remove access privileges is:

```
REVOKE access-list FROM user-list;
```

Eg.  
```
REVOKE SELECT, UPDATE(COMM) 
    ON EMP FROM OPS$corjd;

REVOKE ALL ON MY TABLE FROM PUBLIC;
```

21.11 **Database Systems Without Built-in Security**

Some database systems (such as Visual Basic) do not support the `GRANT` and `REVOKE` commands.

*These systems rely on the usernames and filename protection of the operating system for security against unauthorised access.*

Permission for other users to access database files must be given using the operating system commands - Eg. "filer" on the PC.
CHAPTER 22

Database Application Development

Contents

22.1 The Failure of Software Development .................................................. 285
22.2 An Alternative Approach for Database Application Development: Prototyping ................................................. 286
22.3 The Benefits of Prototyping .................................................................. 287
22.4 Prototyping With Database Management Systems .................................. 288
22.5 Prototyping and Database Design ............................................................ 289
22.6 Example Issues that Affect the Database Design ................................. 290
22.7 How Is Prototyping Done? ....................................................................... 292
22.8 Using a 4th Generation Development System - *Is This Really Prototyping?* ......................................................... 293
22.9 Prototyping Costs for Database Applications ......................................... 294
22.10 Prototyping Methods ............................................................................ 295
22.11 Advantages of Prototyping ................................................................... 298
22.12 Disadvantages of Prototyping ............................................................... 300
22.13 A Real Life Prototyping Experience ...................................................... 301
22.1 The Failure of Software Development

Statistics from DTI quoted in Software Engineering Solutions seminar:

For large software systems (>50K lines of code):

- Only 1% finished on time.
- On average they are one year late.
- On average they cost twice the original estimate.

Half the worlds software staff are involved in maintenance work....

.... at least half of this effort is non productive.

Most new software:

- Fails to meet aspirations of the sponsors.
- Fails to be accepted by users.
- Does not survive its anticipated life.

Is there no better way?
22.2 **An Alternative Approach for Database Application Development: Prototyping**

*What is prototyping?*

Prototyping is the process of transforming an idea into a model for purposes of developing, testing and communicating that idea.

*How does it work?*

It gives a visual representation that can be examined, and a working model of the interface of an application that can be tried out.

A pure prototype:

- is **not** the first version of the end product.
- is a throw away model built purely to try out an idea.
- is often referred to as a *rapid prototype*.

*Why is it done?*

Prototyping provides an effective method for generating feedback about what is good and what is bad about an idea.

It is often the only really effective way of doing so.
22.3 The Benefits of Prototyping

Prototyping can help in the following areas:

• Testing the feasibility of a solution idea
• Further developing a solution idea
• Communicating that idea to others
• Market analysis
• Testing a users reaction to a system
• Ergonomic and productivity testing
• Dividing the work into segments for different developers and for phased development
• Trying out alternative solutions
• Writing user and training manuals

It is particularly useful for the trying out ideas for the user interface .......

...... a notoriously difficult area to predict accurately as what seems right in theory can be quite impractical in practice.
22.4 Prototyping With Database Management Systems

Most database applications are built using:

either: the database management system's own built in application development system.

Eg. Visual Basic for Applications in Access

or: an independent development system that can interface to a database.

Eg. Visual Studio using Visual Basic or C#

or: a Web browser front end that interfaces to a database through an intermediate program.

Eg. Firefox, Chrome or Internet Explorer interfaced with a program written in Java, PHP or Pearl.

These allow a fast development of the user interface.

.... this is important for database systems because:

1. The database system itself may provide a fast and convenient means of directly generating default screens for handling a database table.

   *ie. The database helps produce the user interface*

2. The interface gives important clues about the data structure.

   *ie. The user interface helps design the database*
22.5 Prototyping and Database Design

The above is a typical screen that may be developed as a prototype for testing by a future user of an application.

Obviously it could uncover issues of usability such as:

- How do I abandon input of a persons details?
- How do I obtain help on using the system?

but

*it can also uncover issues that directly affect the database table design.*
22.6 **Example Issues that Affect the Database Design**

1. The user could point out that there is more than one person with the same name.

   This identifies a need for a ID number as an additional attribute.

2. The user could ask for new ID numbers to be generated automatically.

   This indicates the need for a single row table just to hold the last used ID number to enable the ID numbers to be generated.

3. The user may ask whether the surname should be put in first and whether first names or initials are required.

   This could lead to further enquiries identifying a need for separate surname and first name table fields.

4. The user may ask what format the dates should be in, which may reveal the information that dates in both the 20th and 21st centuries are to be stored.

   This indicates the full 4 digit year must be stored in any date attribute fields.
5. The user could object to filling in a long job title or department name and ask to enter a shortcode instead.

   This identifies a need for separate Department and Job Title entity tables.

6. The user could ask for the salary screen field to be filled in automatically as it can be calculated from the grade and job title.

   This indicates that the salary should not be in the Person table, but in a table with the job shortcode and the grade as a multiple field keys.

7. The user could complain there is no way of entering a person with more than one degree.

   This indicates a need for a separate Degree Qualification entity table.

8. The user may try to leave some fields blank.

   This may indicate that some attributes are optional and perhaps should be made into separate entities to avoid too many null database fields.
22.7 How Is Prototyping Done?

To be more specific, the question is really:

*How can a prototype be developed which provides a meaningful simulation of the real thing, and yet is produced so cheaply, in terms of time and effort, that a worthwhile saving is made on the whole project?*

With hardware it is relatively easy:

   Development costs are small compared with mass production.

With software it is much more difficult:

   Mass production is very cheap compared with the high costs of development.

*What is a reasonable cost for a prototype?*

This will depend on:

- How effective it is in generating useful feedback.
- How much, if any, is reusable in the end product.

The high cost of software maintenance could justify spending up to a third of the development cost on throw away software....

.... but for a modern database management system the reusable interface means the cost is rarely so high.
22.8 Using a Database Management System for Development - *Is This Really Prototyping?*

With a modern application development system, such as in a database management system, the screen design and layout of any prototype can be later used for the final system ....

.... *can this be called prototyping?*

Strictly speaking the answer is "no".

- It is not a pure prototype as it will not be thrown away

    **but ....**

- often it is not the first version either as it may contain some throwaway code, eg. to place sample values in the screen fields.

_It does not matter whether the prototype is pure or some hybrid ...._

    .... *the important principle is that an early interface model is produced that can generate user reaction and feedback.*

_With these modern interface development systems there is so little cost when producing a prototype there is really no excuse for not doing so!_
22.9 **Prototyping Costs for Database Applications**

A prototype of a database application developed with a database management system will consist of:

1. **The interface screens**
   
   These have **no cost** as they are reused
   
   *plus*.  
   - they are produced quickly
   - they can generate useful feedback

2. **Some working code reusable in the end product**

   This has **no cost** as it is reused ....
   
   *but*.  
   - it can take a long time to produce
   - a prototype looses value if it is not produced quickly

   ie. Reusable, working code should be kept to a minimum in a prototype

3. **Some code to show samples of how the system would work.**

   This **has a cost** as it is thrown away ....
   
   *but*.  
   - it can often be produced very quickly
   - it can generate useful feedback

   ie. Throw away sample code is worthwhile if produced quickly
22.10 Prototyping Methods

How can a prototype be produced quickly with minimum cost?

1. Produce only part of the system

   Is it possible to partition the program such that a prototype can be built for part of the system?

   A prototype will usually be very useful even if it has nothing but the interface ....

   .... this is the part of the program most likely to generate useful feedback.

   Most database management systems have quick and easy to use facilities for building the user interface based on the table structure.

2. Use a reduced database

   A program may be greatly reduced if an in-memory database set up by hand, is used instead of a vast disk based database.

   Three, two or even one database record can be sufficient to show how the system would work.

   It may be possible to dispense with parts of the database altogether, if simpler data relations will suffice for demonstration purposes.
3. **Simplify the development process**

   *ie. Sacrifice quality and reliability.*

   The prototype is a *throw away* product, it does not need to be robust.

   Even if the interface screens are to be kept the underlying code is often thrown away.

   A hacked together program may be sufficient for a demonstration or user test.

   *But . . .* if several prototype versions are eventually required, modifications from one prototype to another may be difficult.

4. **Simplify data handling and error checking**

   The prototype is merely to show the user what will happen when the right and the wrong input is used.

   It need handle only one example of correct input, and one example for each input error action.

   Eg. It could treat "1/2/1997" as valid and treat *any* other date as an invalid date entry.
5. **Ignore user input detail**

   The modes of operating a system can still be shown even if the user text input is disregarded.

   A search could always "locate" the record for John Smith regardless which name the user requested.

6. **Use alternative software to generate output**

   Much of the user's feel of a program comes from it's output .... .... so a sample of the output can generate useful feedback.

   For example:

   *Output onto paper* - Use any word processor, drawing package or any other convenient software to create sample reports.

   *Output to the screen* - Even if it cannot do a thing, a sample screen or two created by any convenient software can generate feedback.

7. **Use a modified version of another program**

   A modification of similar application in a different context, even on a different computer, may be sufficient to generate user feedback.
22.11 Advantages of Prototyping

1. It is an method of extracting from the customer the application requirements that affect the database design and system operation which is:
   - Effective and more reliable than other methods
   - Often quicker than other methods

2. It is an effective means of checking the database design and system operation, identifying:
   - Misunderstandings between the customer and developer
   - Errors in the customer's specification
   - Omissions
   - Difficult to use features

3. Alternatives can be tested and compared using different try-it-and-see prototypes.

4. The early 'tangible' nature of a prototype:
   - Gives greater developer satisfaction
   - Pleases managers who like to see "progress"
   - Impresses customers
   - Helps communicate the developer's ideas to the customers.
5. Customer participation in the design process:
   • Helps the customers decide what they want
   • Helps the developer to know what the customers actually need
   • Helps the developer to know any customer eccentricities
   • Increases customer loyalty to the end product

6. The prototype can serve as a specification for the full production system which:
   • Is easily communicated to a team of developers without ambiguity
   • Enables developers to identify sections for splitting up the work between team members
   • Allows technical authors to immediately start on the documentation of user and training manuals
22.12 **Disadvantages of Prototyping**

1. *The prototype takes too much time and effort.*

   **But** for modern database management systems ....
   
   1. the speed which prototypes can be produced
   2. the ability to re-use much of the prototype in the end product (especially the screens)

   mean that prototyping will nearly always be cost effective.

2. *The prototype is confused with the real thing.*

   This could mean either:

   (a) The limitations of the prototype are taken as a reflection of the final product.

       ie. An overall bad impression is given.

   (b) The customer is pleased with the prototype and wants to use it.

       The developer may then be under pressure to further develop the prototype ....

       .... but later developments are based on a design which was full of compromises and never intended for further work.

       *ie. It becomes a bodge!*
22.13 **A Real Life Prototyping Experience**

Scene: A prototype has been shown which the customer likes so much the customer wants to use it.

Developer: *No you can't!*

Customer: *It's mine - I paid for it.*

D: *You can't, I'm not ready.*

C: *I'm going to - I can see it does what I want.*

D: *It's not documented!*

C: *I'll learn!*

D: *I won't be able to develop it - it'll all change!*

C: *It works at it is doesn't it?*

D: *Yes, but only bits of it - it's not finished.*

C: *Oh - why not?*

D: *It's a prototype!*

C: *But I can do xxx, yyy, and zzz can't I?*

D: *Yes* (.... mistake!)

C: *OK then I'll use that.*

D: *OK you can use the prototype to get used to using the buttons and to find out about how the system is used within Windows ....

.... it may all change ....

.... do not 'go live' until I have developed the full system, only run it as a duplicate system.*

C: *OK* (.... a narrow escape!)
# INDEX

## A

<table>
<thead>
<tr>
<th>Term</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALL SQL clause</td>
<td>197,204,282-283</td>
</tr>
<tr>
<td>ALTER TABLE SQL Command</td>
<td>187-188,193,238-239,</td>
</tr>
<tr>
<td></td>
<td>243,257,282</td>
</tr>
<tr>
<td>ANY SQL clause</td>
<td>197,201</td>
</tr>
<tr>
<td>Application Development</td>
<td>7,142,284-301</td>
</tr>
<tr>
<td>AS SQL clause</td>
<td>151,180,186,204,210</td>
</tr>
<tr>
<td>Attributes</td>
<td>11-18,24,30,53-54,60-61,65,72,</td>
</tr>
<tr>
<td></td>
<td>74,117-122,143</td>
</tr>
<tr>
<td>AVG SQL group function</td>
<td>161</td>
</tr>
</tbody>
</table>

## B

<table>
<thead>
<tr>
<th>Term</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>BETWEEN SQL clause</td>
<td>155</td>
</tr>
<tr>
<td>BOOLEAN data type</td>
<td>185</td>
</tr>
<tr>
<td>Boyce</td>
<td>101</td>
</tr>
<tr>
<td>Boyce-Codd normal form</td>
<td>87,100-101,104-105,122-123,137</td>
</tr>
<tr>
<td>BYTE data type</td>
<td>185</td>
</tr>
</tbody>
</table>

## C

<table>
<thead>
<tr>
<th>Term</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Calculated values</td>
<td>161-164,114,215,217</td>
</tr>
<tr>
<td>CASCADE SQL clause</td>
<td>259-260</td>
</tr>
<tr>
<td>Case Studies</td>
<td>70-84,127-139</td>
</tr>
<tr>
<td>CHAR data type</td>
<td>184</td>
</tr>
<tr>
<td>Chasm trap</td>
<td>42</td>
</tr>
<tr>
<td>CHECK SQL clause</td>
<td>241-246,268-269</td>
</tr>
<tr>
<td>Client</td>
<td>144</td>
</tr>
<tr>
<td>Codd</td>
<td>86</td>
</tr>
<tr>
<td>Column Headers</td>
<td>151,154</td>
</tr>
<tr>
<td>COMMIT SQL Command</td>
<td>193,273,277,277,280</td>
</tr>
<tr>
<td>Compressed indexes</td>
<td>226</td>
</tr>
<tr>
<td>CONSTRAINT SQL clause</td>
<td>243,245</td>
</tr>
</tbody>
</table>
Constraints  240-260,268-269
Cost  292,294
COUNT  SQL group function  161-162
CREATE ASSERTION  SQL Command  193,244
CREATE INDEX  SQL Command  193,227,248,282
CREATE TABLE  SQL Command  184-187,193,238-239,
               241,243,245,256-257
CREATE VIEW  SQL Command  193,210
CROSS JOIN  SQL clause  174-175

D

Data dictionary  10,53
Database  143
DATE  data type  185
De-normalisation  220-222
DEFAULT  SQL clause  238-239
Default values  238-239,251,267
DELETE  SQL Command  191,193,277,282
Dependent entities  29,30,44
Dependent fields  92-99,101,105-109,136
Derived fields  114
DESC  SQL clause  157-158
Detail entities  29,30,253,264
Determinant fields  92
Diagrams  5,19-23,26-27,30,32-35,37-38,76,80
Discovered entities  48-51,60,81,121,124
Disk access  233
DISTINCT  SQL clause  159,161,201,217
Domain-key normal form  87
DOUBLE  data type  184
DROP ASSERTION  SQL Command  193,244
DROP INDEX  SQL Command  193,227,258
DROP TABLE  SQL Command  188,193,282
DROP VIEW  SQL Command  193,217
### E

<table>
<thead>
<tr>
<th>Term</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Either-or relationships</td>
<td>37, 61-64</td>
</tr>
<tr>
<td>Entities</td>
<td>11-20, 24-29, 48-54, 60-62, 65-67, 72, 74-75, 81, 83, 114, 117-121, 124-125</td>
</tr>
<tr>
<td>Entity relationship modelling</td>
<td>5, 6, 9-84, 86, 115-126</td>
</tr>
<tr>
<td>EXCEPT SQL clause</td>
<td>206-207</td>
</tr>
<tr>
<td>EXISTS SQL clause</td>
<td>199-201</td>
</tr>
</tbody>
</table>

### F

<table>
<thead>
<tr>
<th>Term</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fan trap</td>
<td>40-41, 79</td>
</tr>
<tr>
<td>Fifth normal form</td>
<td>87, 109-111, 114, 119, 123, 125</td>
</tr>
<tr>
<td>First normal form</td>
<td>87-90, 105, 117, 131, 133, 136, 138</td>
</tr>
<tr>
<td>FOR UPDATE SQL clause</td>
<td>166-167, 278-279</td>
</tr>
<tr>
<td>FORIEGN KEY SQL clause</td>
<td>257-260</td>
</tr>
<tr>
<td>Foreign keys</td>
<td>56, 63-64, 250-260</td>
</tr>
<tr>
<td>Fourth generation systems</td>
<td>141-142, 288-289, 292-294</td>
</tr>
<tr>
<td>Fourth normal form</td>
<td>87, 106-108, 110, 133</td>
</tr>
<tr>
<td>FROM SQL clause</td>
<td>150, 167</td>
</tr>
<tr>
<td>FULL JOIN SQL clause</td>
<td>176-177, 179</td>
</tr>
</tbody>
</table>

### G

<table>
<thead>
<tr>
<th>Term</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>GRANT SQL Command</td>
<td>193, 282-283</td>
</tr>
<tr>
<td>GROUP BY SQL clause</td>
<td>161-164, 167, 215, 217</td>
</tr>
<tr>
<td>Group data</td>
<td>161-164, 167, 215, 217</td>
</tr>
</tbody>
</table>

### H

<table>
<thead>
<tr>
<th>Term</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>HAVING SQL clause</td>
<td>164, 167, 232</td>
</tr>
<tr>
<td>Help</td>
<td>266, 289</td>
</tr>
<tr>
<td>Hidden entities</td>
<td>48-51, 60, 81, 121, 124</td>
</tr>
</tbody>
</table>

### I

<table>
<thead>
<tr>
<th>Term</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Identifier values</td>
<td>262-265, 290</td>
</tr>
<tr>
<td>IN SQL clause</td>
<td>155-156, 167, 198, 201</td>
</tr>
<tr>
<td>Indexes</td>
<td>223-227,248,258</td>
</tr>
<tr>
<td>INNER JOIN</td>
<td>SQL clause</td>
</tr>
<tr>
<td>INSERT</td>
<td>SQL Command</td>
</tr>
<tr>
<td>INTEGER</td>
<td>data type</td>
</tr>
<tr>
<td>Integrity</td>
<td>220-222,235-283</td>
</tr>
<tr>
<td>INTERSECT</td>
<td>SQL clause</td>
</tr>
<tr>
<td>INTO</td>
<td>SQL clause</td>
</tr>
</tbody>
</table>

**J**

Joining Tables | 102-103,137,168-182,212,219,230-231 |

**K**


**L**

LEFT JOIN | SQL clause | 176-177,179 |
LIKE | SQL clause | 155-156 |
LOCK TABLE | SQL Command | 280 |
Locking | 274-280 |
LONG | data type | 185 |

**M**

Maintenance commands | 183-193,225 |
Mandatory fields | 245-246,266 |
Master entities | 29,253,264 |
MAX | SQL group function | 161 |
MEMO | data type | 185 |
Memory use | 219,234 |
Merging tables | 112-113,220-222 |
MIN | SQL group function | 161 |
MINUS | SQL clause | 207 |
MONEY | data type | 185 |
Multi-user systems | 272-283 |
Multi-value dependencies 105-109,134
Multi-way relationships 45-47,51,80,124-125
Multiple keys 13,53,93,100,121-123

N

NATURAL JOIN SQL clause 170,175
NOT NULL SQL clause 155-156,224,245-247,258,266
Notation alternatives 23,32,35
Normalisation 5,6,85-139,221-223,236
NOWAIT SQL clause 278
Null fields 55,57,94,190,245-247,251-252,266,291

O

Obligatory relationships 33-35,54-58,77
ODBC 144
ON SQL clause 167,171-172,282-283
ON DELETE SQL clause 259-260
ON UPDATE SQL clause 260
Optimisation 47,55,57,66-67,94,97,119,

Optional relationships 33-36,55-58,62,77
ORDER BY SQL clause 157-158,162,167,205

P

Passwords 270
Performance 219-234
Pop-up menus 269
PRIMARY KEY SQL clause 256-258
Processes 21,296
Prototyping 286-301
Pull-down menus 269
PUBLIC SQL clause 282-283
Index

Q
Queries  103,148-182,186,190,194-207,223,226,
228-232,278-279,281-283

R
READ ONLY SQL clause  281
REAL data type  184
Records  143
Redundant relationships  40-43,79
REFERENCES SQL clause  256-260
Referential integrity  250-260
Reflective entities  26
Relation  143
Relationships  20-24,26-37,40-51,54-64,73,76-77,
79-80,84,121,124-125
REVOKE SQL Command  193,283
RIGHT JOIN SQL clause  176-179
ROLLBACK SQL Command  193,273,277,280

S
Second normal form  87,93-96,118,122,131,133,136,138
Security  211,282-283
SELECT SQL Command  103,148-182,186,190,194-207,
228-232,278-279,281-283
Sequence of entities  66-67,114
Server  144
SET SQL clause  192
SET ARRSIZE SQL Command  234
SET TRANSACTION SQL Command  281
SMALLINT data type  184
SOME SQL clause  197
Speed  218-234,298,300

SQL Clauses:  167
ALL  197,204,282-283
ANY 197,201
AS 151,180,186,204,210
BETWEEN 155
CASCADE 259-260
CHECK 241-246,268-269
CONSTRAINT 243,245
CROSS JOIN 174-175
DEFAULT 238-239
DESC 157-158
DISTINCT 159,161,201,217
EXCEPT 206-207
EXISTS 199-201
FOR UPDATE 166-167,278-279
FOREIGN KEY 257-260
FROM 150,167
FULL JOIN 176-177,179
GROUP BY 161-164,167,215,217
HAVING 164,167,232
IN 155-156,167,198,201
INNER JOIN 170-173,176,180-182,201
INTERSECT 206-207
INTO 165,167,186,190
LEFT JOIN 176-177,179
LIKE 155-156
MINUS 207
NATURAL JOIN 170,175
NOT NULL 155-156,224,245-247,258,266
NOWAIT 278
ON 167,171-172,282-283
ON DELETE 259-260
ON UPDATE 260
ORDER BY 157-158,162,167,205
PRIMARY KEY 256-258
PUBLIC 282-283
READ ONLY 281
REFERENCES 256-260
RIGHT JOIN 176-179
SET 192
SOME 197
TOP 160
UNION 202-207
UNIQUE 226-227,247-249,258
USING 170,175
VALUES 190
WHERE 103,155,164,167,176-177,191-192,195,201,224,228-232
WITH GRANT OPTION 282-283

SQL commands: 146
ALTER TABLE...ADD 187,193,238-239,243,257,282
ALTER TABLE...DROP 188,193,243,257,282
ALTER TABLE...MODIFY 188,193,238,243,282
COMMIT 193,273,277,280
CREATE ASSERTION 193,244
CREATE INDEX 193,227,248,282
CREATE TABLE 184-187,193,238-239,241,243,245,256-257
CREATE VIEW 193,210
DELETE 191,194,277,282
DROP ASSERTION 193,244
DROP INDEX 193,227,258
DROP TABLE 188,193,282
DROP VIEW 193,217
GRANT 193,282-283
INSERT 190,193,195,239,277,282
LOCK TABLE 280
REVOKE 193,283
ROLLBACK 193,273,277,280
SELECT 103,148-182,186,190,194-207,228-232,278-279,281-283
SET ARRAYSIZE 234
SET TRANSACTION 281
TABLE 148,157
UPDATE 187,192-193,272,277,279,282-283

SQL conditions 155,228-229,241-244
SQL constants

**SQL data types:**

- BOOLEAN 185
- BYTE 185
- CHAR 184
- DATE 185
- DOUBLE 184
- INTEGER 184
- LONG 185
- MEMO 185
- MONEY 185
- REAL 184
- SMALLINT 184
- TIME 185
- TIMESTAMP 185
- VARCHAR 184

**SQL group functions:**

- AVG 161
- COUNT 161-162
- MAX 161
- MIN 161
- SUM 161-162

**SQL names**

**SQL templates**

**SQL/92 standard**

**Subqueries**

**SUM SQL group function**

T

**TABLE SQL Command** 148,157
Text Attributes 17-18,88,119-120,291
Tables 10-12,53-67,83-84,87-139,147
Table alias names 180,200
Index

Third normal form 87,96-101,118-120,121-123, 132-133,136,138
Three-way relationships 45-47,51,80,124-125
TIME data type 185
Time slots 67,114
TIMESTAMP data type 185
TOP SQL clause 160
Transactions 272-273,277-279
Tuple 143

U

UNION SQL clause 202-207
UNIQUE SQL clause 226-227,247-249,258
UPDATE SQL Command 187,192-193,272,277,279, 282-283
USING SQL clause 170,175

V

VALUES SQL clause 190
VARCHAR data type 184
Views 103,137,208-217

W

WHERE SQL clause 103,155,164,167,176-177,191-192, 195,201,224,228-232
WITH GRANT OPTION SQL clause 282-283