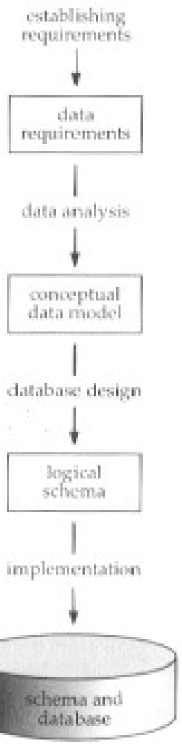
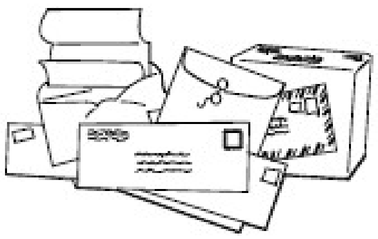
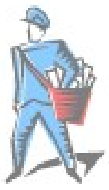
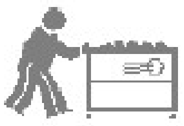
THE OPEN UNIVERSITY, TM422
PROJECT REPORT 2002

Resources Management Database For Royal Mail Coventry Delivery Office









Prepared by:
Omer Dawelbeit
T0986935

Abstract

Table of Contents

Chapter 1

Introduction

1.1 Project aims

The aims of this project can widely be divided into two main areas:

1- To develop a database for Coventry delivery office using the database development techniques outlined in M358, and to ensure that the database satisfies its client's requirements and maintains internal consistency.

2- To further explore and investigate how the new features in SQL99 can be used to implement postal addresses.

The delivery office database should satisfy, but is not limited to the following:

- Assists D.O managers (DOMs) to keep and maintain personnel records.
- Replace the current system, which consists of Microsoft Excel spreadsheets. The old system suffers from many problems such as inconsistencies and update anomalies and lack of security.
- Record the delivery office (DO) staff details, which include, but not limited to: Full names, addresses, contact numbers, pay numbers, dates of entry to the business, training courses attended, grades, education, job details, …etc.
- Assist the DOM to plan the office annual leave (holidays) for the staff, taking into consideration the maximum quota for the number of staff allowed to be off work for one week.
- Assist the DOM in resources planning and **duty coverage**.
- Help to keep and maintain a sick record for the office.
- Record customer complaints list, which include complaint type, job number and responsible postman pay number.
- Keep a list of the addresses covered by the delivery office, which assist the DOM in address management, e.g. adding new addresses or deleting demolished addresses.

The project also aimed at exploring in depth some database issues, this exploration required some literature search. The issues investigated in this project can be summarised as follows:

- The components of the query processor part of the DBMS and their operation. The project also briefly discusses how the query can be mapped by these components into an optimised sequence of lower level operations.

- The new features in SQL99, especially user defined types and how these can assist in implementing complex data that has subcomponents such as postal addresses. The project also looked at the importance and the structure of postal addresses.

1.2 Project description

1.2.1 Introduction

The database produced in this project is required by Royal Mail delivery office in Coventry. Royal Mail deals with collection, processing, distribution and delivery of mail in the UK. These stages collectively are called the pipeline, and the last stage in this pipeline is delivery. Usually a delivery function is carried out in local offices all throughout the country and they delivery to specific postcodes, e.g. CV6 area.

Currently the administration staff in the delivery office is using spreadsheets in the management of the day to day activities. However, the data in these spreadsheets is just used for storage and display purposes and can sometimes become invalid or inconsistent because of the lack of the meaning of the data, constraints, referential integrity, security and access control. The spreadsheet files are spread over a number of computers and usually contains partial data about the resources of the four delivery offices. These files are being created by the administration staff to help them carrying out their activities. When one of the administration staff is off work, it proved to be very hard for the person covering his/her job to locate or use these files.

For the above reasons the delivery office requires the development of a standard database system that can be used by all administration staff. The database system must satisfy the requirements outline in **Appendix A**, some of these requirements is to store and maintain lists of data such as employees, vehicles, walks and duties. Other requirements are to satisfy the daily and weekly operation of the delivery offices, which are mainly concerned with the **duty coverage**. The database is required to represent and maintain the integrity and the meaning of the data and also have the ability to be extended to support access controls and security.

The project also gave an overview of the structure and the operation of the query processor module of the DBMS. This module is investigated because of its importance in ensuring that the query language need only specify the result of a query not how it should processed. Besides, this module relives the user from query optimisation, a time-consuming task that is best handled by the query processor.

There was also a need to explore the current advances in SQL, and how these can be used to implement postal addresses. The importance of postal addresses is that they exist almost in all databases that hold personnel details. The way these addresses are implemented in a database affects the way they can be manipulated and processed. This means if the DBMS doesn't recognise the internal parts of the addresses, then extra processing is required on either the DBMS side or the application process side. The project tries to give a solution to this by implement postal addresses using SQL99 user defined types.

1.2.2 Procedure

The following list gives a brief outline of the techniques used for the development of the project:

- Project management techniques were used to organise and schedule a plan for the work on the project. TurboProject Express[1] Software was used to construct a Gantt chart for the project schedule.
- Literature search was carried out for the work on the main part of the project and for the further topics. This involved the use of various resources such libraries, databases, online resources, and so on.
- Relational database development steps outlined in M358 were used to develop the D.O database. These steps can be summarised as establishing requirements, data analysis, database design database implementation and finally database testing.
- Meetings were held with the client to establish a statement of data requirements (Appendix A)
- The statement of data requirements was used to produce a conceptual data model using entity relationships modelling (Appendix B)
- The conceptual data model was then transformed to a relation model, which is the specification of the logical schema (Appendix C).
- SQL Anywhere 5.0 (Copyright © Sybase Inc.) was used for the implementation of the relation model. This version of SQL is the one used by Sybase SQL Anywhere DBMS.
- InfoMaker 5.0 (Copyright © Sybase Inc.) was used as a direct entry tool to enter and execute the SQL statements used to create the D.O database.

[1] TurboProject Express Ver 2.02 Copyright © IMSIsoft 2000 (http://www.imsisoft.com/).

1.3 Review of Database Development Techniques

The activities carried out in this project for the database development are shown in Figure 1.1 below and can be summarised as follows (OU, M358, bk. 4, pp. 7):

- **Establishing requirements** involves consultation with, and agreement among, users as to wheat persistent data they want, expressed as a statement of data requirements.
- **Data analysis** starts with a statement of data requirements and produces [an E-R] conceptual data model, which is a formal representation of what data a database should contain, expressed in terms that are independent of how it may be realized.
- **Database design** starts with a conceptual data model and produces a specification of a logical schema [(relational model)].
- **Implementation** involves the construction of a database according to a given specification of a logical schema, which requires specification of an appropriate storage schema [using SQL].

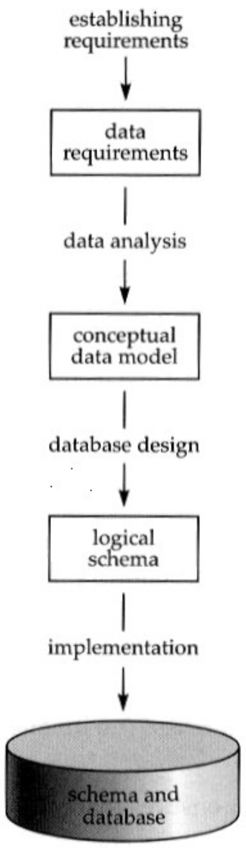


Figure 1.1 Model of database development (OU, M358, bk. 4, pp. 7)

Chapter 2

Literature Search

2.1 Introduction

A big part of this project required collecting information from external sources other than the course materials; this is why the literature search was a major activity throughout the project progress and the database development.

I've started the literature search while working on TMA01 to develop my project proposal, and that was mainly to collect some initial ideas, at that stage my search was not structured and I wasn't aware of the different sources and tools available.

During that first stage of my project I've searched different sources randomly without any search strategy, besides I did not know what exactly I was looking for. This was apparent when reflecting on the literature search in TMA02, and from my tutor comments I realised that I need a methodology for the literature search and also to develop more searching skills. A starting point was SAFARI [1] (Skills in Accessing, Finding, and Reviewing Information). The online course is organised into sections that cover the following areas:

- Understanding information
- Unpacking information
- Planning a search
- Searching for information
- Evaluating information
- Organising information

After finishing the online course, I had a very good idea about the basic principles and skills required to carryout an effective literature search, not only that but also the different sources of information such as library catalogues, bibliographic databases, and the World Wide Web.

I started by identifying the categories of the information I needed to collect. These categories are information in the database field and general information. Information in the database field was mainly required for the further topics. However, some of this information was also required for the main part of the project.

The next step was to identify the sources of information such as library catalogues, bibliographic databases and the World Wide Web. I've identified some of the digital bibliographic databases which are specialized in the Information Technology field and in particular databases. Some of these are maintained by:

- **ACM**: Association for Computing Machinery.
- **IIEEE**: Institute of Electrical and Electronics Engineers.
- **VLDB**: Proceedings of the International Conference on Very Large DataBases
- **DBPD**: Database Programming and Design

2.2 Literature search relevant to the main part of the project

The client supplied most of the supplementary material required for me to gain domain experience in the operation of the delivery office. This helped me to understand what needed to be represented in the database. On the other hand most of the literature required for the process of database development was from M358 course material. When searching for a specific subject I used the course index as a faster way of locating the information.

During the data analysis I was not sure of how to model postal addresses. There were two choices, either to model a whole address as an attribute or model an address at an entity. The first option would require the use of SQL procedures to manipulate different parts of the address when the database is implemented. However, in the second option the address sub-component can be easily used in queries.

At this stage I needed some information about the actual modelling of postal addresses in the **P**ostcode **A**ddress **F**ile[2] (**PAF**). I used the search facility on Royal Mail website[3] and located 'PAF 5.0 Digest' (Royal Mail 2000). This is a comprehensive guide that explains the structure of the PAF database and the format of the subcomponents of postal addresses. Based on the modelling of the addresses in the PAF as tables, I have decided to follow the same approach when modelling addresses in the D.O database.

The information provided in (Royal Mail 2000) about the length of each subcomponent also proved useful when constructing the relational model, and it was used as a guide when deciding the string length for the postcode, street and so on.

During the implementation activities I required some information about using some the SQL Anywhere statements and data types, (Sybase, Inc. 1996) proved very useful. This user guide was supplied in an electronic format and has a search facility. The guide offered comprehensive cover for SQL Anywhere syntax and usage. I used the guide all throughout the implementation activity to locate information such as SQL domains, triggers, user defined functions and procedures.

During the project lifetime I needed some discrete information, for example the definition of some database terms. I have used the World Wide Web for such information by trying to locate online dictionaries for computer and Information Technology. Using Yahoo search engine (www.yahoo.com) with keywords (such as 'IT dictionary') I came across Webopedia (http://www.webopedia.com) a very useful online dictionary for IT terms.

[2] **PAF** is a registered trademark of the Royal Mail (part of Consignia plc) and stands for Postcode Address File. It is a database containing all known addresses and Postcodes in the United Kingdom, including England, Scotland, Wales, Northern Ireland, Jersey, Guernsey, and the Isle Of Man (over 26 million addresses, 1.71 million Postcodes). It includes Small User Residential, Small User Organisation and Large User Organisation details.

[3] Royal Mail online: (http://www.consignia-online.com)

2.3 Literature search relevant to the two further topics:

I started the literature search for this part of the project by first deciding on the two further topics to explore further. The next step was to identify the information I was going to need before planning a search. Reading the relevant text from M358 and writing down a few points that need exploration helped to have a general idea of the sort of information required. The initial topics I decided to explore were:

- Distributed data, by further exploring and discussing the different techniques used such, client multi-server, distributed databases and replication systems.
- Modelling and processing of postal addresses in relational databases.

Both subjects are large and open ended considering the time and the scope for this project. According to this I decided to choose more specific topics. I have decided to investigate the query optimisation part of the DBMS and briefly assess the different search strategies used by the query optimiser. This choice is based on the importance of query processing in improving the performance of query execution in both centralised and distributed DBMS. On the other hand, I've decided to explore the new features in SQL99 that can be used to implement complex data with subcomponents such as postal addresses.

After deciding on the two topics, I considered the following sources of information, and using measures such as quality, relevance, objectivity and timelines to assess the information found:

- **Databases** (A way of storing, indexing, organising and retrieving information). Specialised databases contain information in summary form about books and journal articles.

- **Library catalogues**, are databases containing information, which relates to material located in a particular library.

- The **World Wide Web**

First I considered locating some good reference books that I could use for background reading. The first option was library catalogues, however some of these do no provide full book information and review. I used some well know online booksellers take as an example Barnes and Noble (www.bn.com) online bookstore offers not just search facilities to locate book, but also provide reviews, prefaces and table of contents for most references. Using keywords as *distributed data*, *distributed databases*, *databases*, *client multi-server systems* located a few books one of which was (Valduriez and Ozsu, 1999).

The book does not just explain the Distribute DBMS architecture, but also reviews computer networks, distributed database design, relational DBMS structure and summarises the query processing and the different techniques of optimising distributed queries. The book was relevant and useful for the query optimisation topic. The book explained how queries expressed in relational calculus get rewritten into relational algebra and organised into an operator tree (see Chapter 5).

(Gruber 2000) and (Elmasri and Navathe 2001) were valuable sources of information for the second topic. (Gruber 2000) includes a chapter that summarises the new features in SQL99. On the other hand, (Elmasri and Navathe 2001) gives chapters that discuss Object databases, Enhanced entity relations modelling and SQL99 new object features in details.

Articles published in journals were located using databases such as ACM digital database. Using search techniques, such as titles, subject area, and keywords usually produced some good articles. For example, some articles that I came across are (Kossmann, 2000) and (Eisenberg and Melton 1999).

(Kossmann 2000) discusses a recent research about the state of the art techniques used for distributed query processing and it concentrates on *structured* data (such as that found in relational or object-oriented databases) and on query languages for structured data (such as SQL or OQL). The article also gave a general architecture for query processing, which I've used as a starting point for the query processing topic. (Eisenberg and Melton 1999) included some practical examples using the new user defined types in SQL99.

The literature search I have done was useful, however this came to reality after I learnt about the information search by doing the SAFARI. I have learnt about the different types of information (grey, primary, secondary literature), and where each. Most importantly how to identify my needs and clarify my needs, and make a checklist for each source of information (databases, web pages, library catalogues), and devise a search strategy for each source, and after that assess the value and usefulness of the information found.

Chapter 3

Design Approach & Activities

3.1 Project Management Techniques

After developing the project proposal and setting out the project aims I went towards devising a schedule for the project. This was done using project management techniques such as Gantt charts. Project management can be thought of as the process that holds the project development activities together, and ensures that the project meets its targets and objectives.

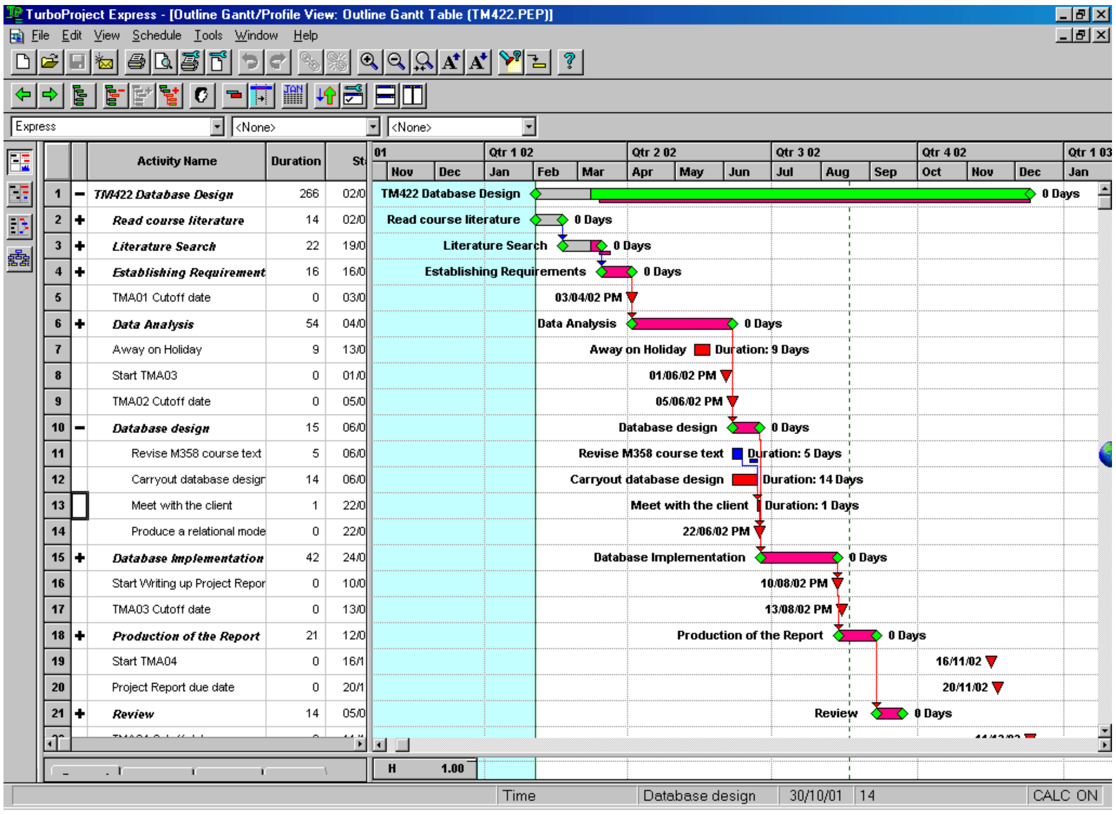
To devise a schedule for the project I divided the project into phases, tasks (activities), and key events. I've used the database development model (Figure 1.1) as a guideline for the different phases of the project. These phases can be summarised as:

- Read course literature
- Literature search
- Establishing requirements
- Data analysis
- Database design
- Database implementation
- Database testing
- Production of the Report
- Project review

Each one of those phases needed breaking down into tasks, so I realised in order to organise the project schedule, I was going to need a project scheduling and planning tool. I used TurboProject Express tool (Figure 3.1), this tool assisted me greatly in developing the project schedule and in realising the critical path and the different tasks forming it.

Using the tool, I have defined tasks and key events for each phase. For example, the development of a conceptual data model is a key event because it forms an interface between data analysis and database design when one is finished the other one can start.

The project schedule played an important role in promoting a disciplined approached that I have followed throughout my work on the project, besides it enabled me to adjust the available time for each activity when others were delayed to stay within the deadline for the project. Alongside the project schedule I have also maintained a project log by recording the work I carried out on every work session. This helped me in writing the project report and reflecting back what I have done.


TurboProject Express - [Outline Gantt/Profile View: Outline Gantt Table (TM422.PEP)]

Express | <None> | <None>

		Activity Name	Duration	St
1	−	*TM422 Database Design*	266	02/0
2	+	*Read course literature*	14	02/0
3	+	*Literature Search*	22	19/0
4	+	*Establishing Requirement*	16	16/0
5		TMA01 Cutoff date	0	03/0
6	+	*Data Analysis*	54	04/0
7		Away on Holiday	9	13/0
8		Start TMA03	0	01/0
9		TMA02 Cutoff date	0	05/0
10	−	*Database design*	15	06/0
11		Revise M358 course text	5	06/0
12		Carryout database desigr	14	06/0
13		Meet with the client	1	22/0
14		Produce a relational mode	0	22/0
15	+	*Database Implementation*	42	24/0
16		Start Writing up Project Repor	0	10/0
17		TMA03 Cutoff date	0	13/0
18	+	*Production of the Report*	21	12/0
19		Start TMA04	0	16/1
20		Project Report due date	0	20/1
21	+	*Review*	14	05/0

01
Nov
Dec
Qtr 1 02
Jan
Feb
Mar
Qtr 2 02
Apr
May
Jun
Qtr 3 02
Jul
Aug
Sep
Qtr 4 02
Oct
Nov
Dec
Qtr 1 03
Jan
TM422 Database Design 0 Days
Read course literature 0 Days
Literature Search 0 Days
Establishing Requirements 0 Days
03/04/02 PM
Data Analysis 0 Days
Away on Holiday Duration: 9 Days
01/06/02 PM
05/06/02 PM
Database design 0 Days
Revise M358 course text Duration: 5 Days
Carryout database design Duration: 14 Days
Meet with the client Duration: 1 Days
22/06/02 PM
Database Implementation 0 Days
10/08/02 PM
13/08/02 PM
Production of the Report 0 Days
16/11/02
20/11/02
Review 0 Days
H 1.00

Time | Database design | 30/10/01 | 14 | CALC ON

Figure 3.1 *Screenshot of Project Express showing a Gantt chart for the project*

3.2 Database Development Techniques

The steps of database development outlined in subsection 1.3 were applied to develop the database. These steps followed the general model shown in Figure 3.2 below. This implied that any revision required for relations or tables to be applied to the conceptual data model. This change should also be propagated to the relational model that can be used to implement new tables. The same process is repeated until the database tables represent the data in an acceptable way.

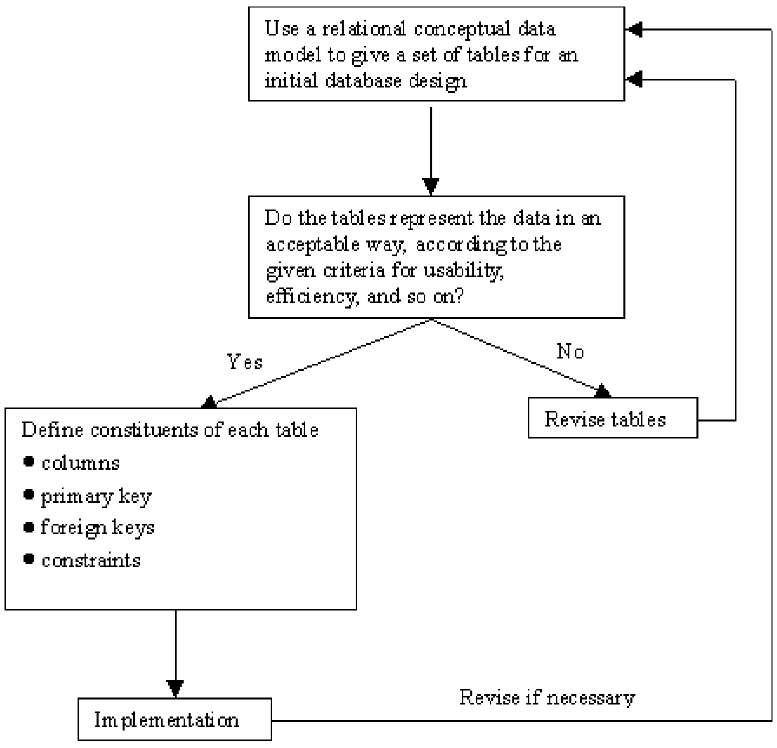

Use a relational conceptual data model to give a set of tables for an initial database design
Do the tables represent the data in an acceptable way, according to the given criteria for usability, efficiency, and so on?
Yes
No
Revise tables
Define constituents of each table
● columns
● primary key
● foreign keys
● constraints
Implementation
Revise if necessary

Figure 3.2 *Summary of steps for database design*

3.2.1 Establishing requirements

After a few meetings with the client to establish the data requirement I have compiled a statement of data requirements (Appendix A). During the work on the statement many problem and ambiguities arose, so I had to consult the client for further clarifications. Most of the client requirements were in a form of tables, spreadsheet and forms that were used in the old system, this material was of great use when writing the statement of data requirements. The statement of data requirements was checked with the client to ensure it fully represented the their requirements.

3.2.2 Data Analysis

The data analysis activities use the statement of data requirements to produce a conceptual data model (Appendix B).

Entity types and relationships

From the initial statement of requirements, I've carried out different activities to analyse the data and identify potential entity types and relationships taking these points as guidelines:

- An entity represents a subject that has meaning in a given context and about which there is a need to record data, which can be identified as nouns or noun phrases.
- A relationship is about the existence of a connection between subjects, and in the context of a statement of data requirement it is a verb or verbal phrase

In the management of the day-to-day activities the delivery office administration staff require a database to meet the following:

1- Coventry has four delivery offices each one of them is identified by its name and has a telephone number, a manager and a postal address.

2- Each delivery office has a number of walks, a walk is a collection of addresses, which are on the same road or are close to each other, each walk has a unique walk number, a type (town, rural, or bulks), a delivery method (van or on foot), notes (dog warnings, etc…) and a postal area that it covers.

3- The delivery office also has a number of vans each van is characterised by a unique number beside the DVLA registration number, a make, a model and the size…

Figure 3.3 *Part of the Delivery Office data requirements statement*

Consider Figure 3.3 as an example, the potential entity types can be identified as:

delivery office, administration staff, manager, postal address, walk, road, walk number, road, van…

By looking at some of these nouns I've dropped manager, road, walk number and road as entity types, because these are potential attributes for entity types. For example a manager is an attribute of a delivery office, road is an attribute of an address and walk number is an attribute of walk. On the other hand, the other nouns such as delivery office represent a subject for which data needs to be recorded i.e. name, manager, telephone number and an address. Administration staff are employees of the client, which is an entity type.

Although the statement mentioned that a delivery office has a number of walks, a walk is not an attribute of a delivery office, because information needs to be recorded about each walk, for example a walk number, and addresses covered by that walk. So this suggests a relationship between an office and its walks such as consists of.
This starts another argument about addresses, if they are an attribute of a walk. But because addresses are not atomic data structures I have decided to represent them as entity types. This

has many benefits, first because many entity types in my design would have an address attribute, it was beneficial to model address as a separate entity type and link it with other entity types using relationships to avoid duplication (other benefits were discussed in subsection 2.2).

After finishing the data analysis I had some entity types and relationships between them. The next step was to find identifiers for these entities. Luckily all the entity types have natural identifiers used by the client, such as employee number, vehicle registration number or walk number. Even addresses in my design have postcodes and building numbers, which are enough to identify a unique address. To summarise I didn't need to use any artificial identifiers.

At this stage of modelling I followed a simple approach, to identify main entity types, their attributes and the relationships between them, then decide if any of these entity types is an attribute or visa versa.

Entity subtypes

Entity subtypes were introduced to assist in understanding some complex situations such the Duty super type and its subtypes. I have also used entity subtypes in the case of Vehicle super type. Not using entity subtypes at this stage would have resulted in more entities and relationships and this will complicate the E-R model. For example, entity Employee would have three relationships with entities FullTimeDuty, PartTimeDuty and NonWalkDuty instead of just one

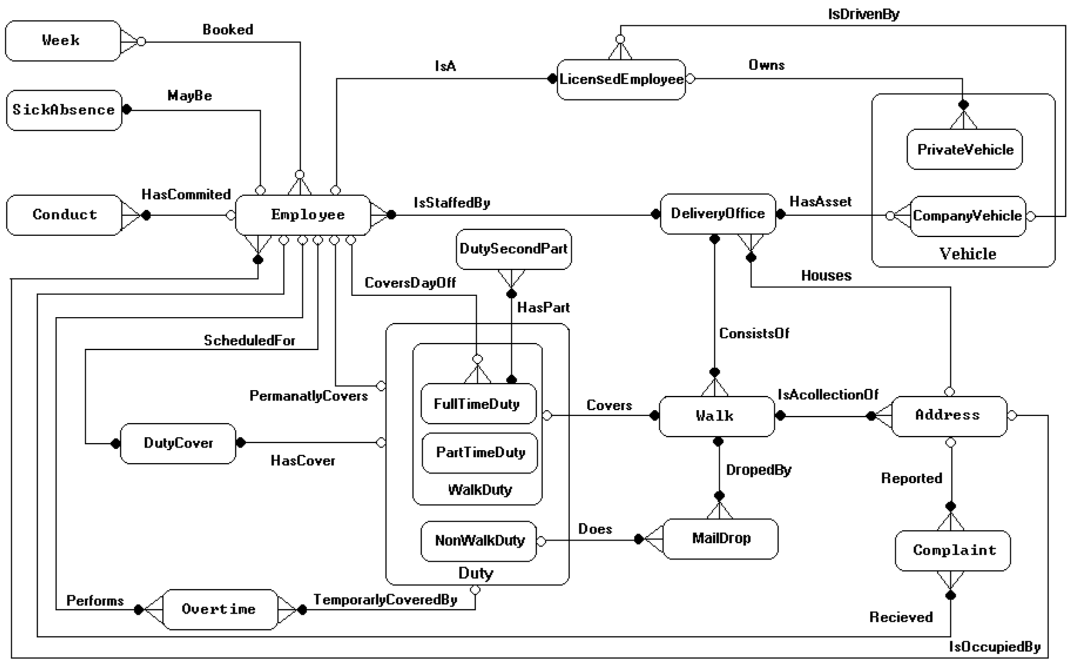


***Figure 3.4** (a) The Delivery Office E-R diagram*

Entity types

Employee (PayNo, Name, DateOfBirth, DateOfEntry, Grade, Phone, BadgeNo, Skills, OvertimeAvailability, Status)
DeliveryOffice (Name, TelNo, ManagerName)
Vehicle (RegNumber, Model)
PrivateVehicle (Colour)
CompanyVehicle (VehicleNo, MotDueDate, Size)
Conduct (Reference, Type, Notes, Date)
Week (WeekNo, Quota)
SickAbsence (PayNo, DateCommenced, Reason)
Walk (WalkNumber,Office, Type, DelMethod, Remarks, Status)
MailDrop (WalkNo, Office, DropPoint, Details)
Duty (DutyNo)
WalkDuty()
FullTimeDuty ()
PartTimeDuty (Hours)
NonWalkDuty(DutyDetails)
DutySecondPart (DutyNo, WeekNo, DayOff, SecondPart)
DutyCover (DutyNo, PayNo, Cause)
Overtime (DutyNo, EmployeeNo, Duration)
Complaint (ReferenceNo, Details, Date)
LicensedEmployee (LicenceNo, VehicleSize)
Address (PostCode, BuildingNumber, StreetName, Town, PostalArea)

Constraints

The total annual leave for a certain week must not exceed the quota for that week.

Only employees with a driver grade or postal higher grade can participate in the PermanatlyCovers relationship with NonWalkDuty.

Only full time employee can participate in PermanentlyCoveredBy with FullTimeDuty and only part time employees can participate in PermanentlyCoveredBy with PartTimeDuty.

An Employee participating in CoversDayOff must not be participating in PermanentlyCoveredBy

A duty can be covered by overtime or duty cover only if it does not participate into PermanatlyCovers and HasCover or the employee doing that duty is participating in the Has or MayBe relationships.

An employee can participate in PermanentlyCoveredBy with only FullTimeDuty or PartTimeDuty or NonWalkDuty.

Employee and duties participating in PermanentlyCoveredBy should belong to the same delivery office

An employee participating in Performs or ScheduledFor should not be absent form work.

An employee can only participate in the Performs relationship if the attribute OvertimeAvailability is 'Yes'.

Assumptions

Only current sick absences are recorded, no history is kept.

A company vehicle might not be a spare vehicle and not used for a specific Delivery Office.

A duty might be vacant, i.e. with no employee covering it.

An employee can be a spare, i.e. not covering any duty.

The overtime is updated daily, i.e. no history is kept.

The duty cover are updated weekly, no history is kept.

All full time duties are 40 hours.

Figure 3.4 *(b) The Delivery Office entity types, constraints and assumptions.*

m:n Relationships

The E-R diagram in Figure 3.4 (a) has an entity type that resulted from resolving n:m relationships this is Overtime. However, this was resolved in the E-R diagram because of the extra attributes that needed to exist such Duration, an attribute of Overtime (The delivery office conceptual data model is included in Appendix B). In the case of the Booked (Figure 3.5), it is not resolved at this stage because only the identifiers of Employee and Week need to be recorded and used to form the new identifier for Booked (WeekNo, PayNo).

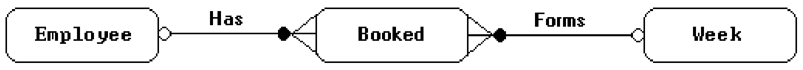


Figure 3.5 *Booked relationship*

Constraints

Constraints are represented in the E-R models in two ways (Figure 3.4):

- As a property of a modelling construct, such as participation conditions and unique attributes.
- As a description in the Constraints part of a model as shown in Figure 3.4(b).

There are a few things in the conceptual data model which are chosen for the design purpose, such as entity type LicencedEmployee because not every Employee can participate in Drives and Owns relationships, only Employees with driving licences can. Looking ahead at an Employee table, employees without driving licences will have Null entries in the column that corresponds to foreign keys referencing PrivateVehicle and CompanyVehicle, so using this new entity type eliminates this problem. The same approach was used with SickAbsence to avoid Null entries in the Employee Table.

3.2.3 Database Design

Developing a relational model:

After constructing the conceptual data model, I started developing the relational model for the delivery office (Appendix C). There following subsections discusses some issues about the design decisions:

Domains

All notes attributes were defined on the GeneralNotes domain. Notes are only descriptive text where no validation is possible and they are unlikely to be compared with each other.

Declaring relations

Entity subtypes concept is not supported by either relational models or SQL schemas. However, there are alternative ways to transform entity subtypes to tables in a logical schema (OU, M358, bk. 4 pp.96). One of these ways is to use relations that only represent the subtypes, and not any super types. For example, the entity type Duty and its subtypes (Figure 3.6) would be represented in the relational model as relations *FullTimeDuty*, *PartTimeDuty* and *NonWalkDuty*. These relations then give three tables, which I've named **full_time_duty**, **part_time_duty** and **non_walk_duty**.

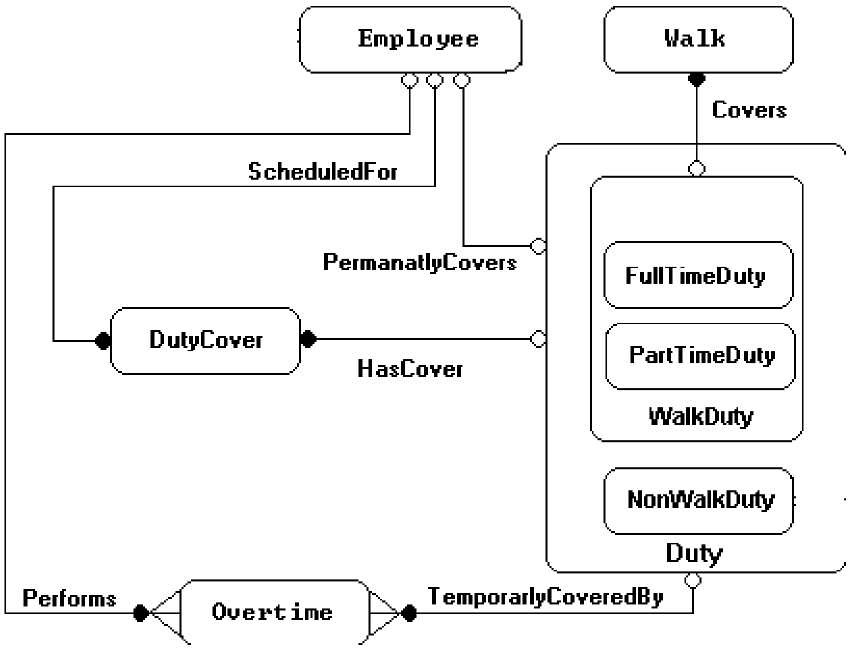


Figure 3.6 Part of the Delivery Office E-R diagram

Representing relationships

1:n relationships were represented using foreign keys, take for example the DeliveryOffice relation:

```
relation DeliveryOffice
    Name: DeliveryOfficeNames
    TelNo: TelephoneNumbers
    ManagerName: PersonNames
    BuildingNumber: BuildingNumbers
    PostCode: PostCodes
    primary key Name
    foreign key (BuildingNumber, PostCode) references Address not allowed null
    { represent mandatory participation with respect to HasAsset }
    constraint (project DeliveryOffice over name) difference (project CompanyVehicle over
    Office) is empty
```

The foreign key (BuildingNumber, PostCode) represents the relationship Houses, the mandatory participation condition from the DeliveryOffice side is expressed by not allowing the foreign key to be null. On the other hand, the mandatory participation condition on the referenced relation side is expressed using a general constraint, for example the participation condition on the DeliveryOffice side in regards to HasAsset.

With regards to 1:1 relationships, their representation has two steps. First, deciding which one of the two relations will have the attribute declared as a foreign key, and then declare that same attribute to be an alternate key. I've followed this technique to represent the following relationships:

IsA, Drives, MayBe and CoveredBy.

During implementation I had to revise the side on which the alternate key for 1:1 relationship representation resides. This was important because SQL Anywhere implies a not null for the UNIQUE definition. This implies a mandatory participation condition from the side including the foreign key. Take for example, the foreign key used in representing the Covers relationship is better located on the Walk side because the participation condition is mandatory, and this is expressed explicitly when declaring UNIQUE in SQL Anywhere. The same was used with regards to MayBe by placing the foreign key on the SickAbsence side.

Representing constraints

Participation conditions constraints in the Delivery Office relational model are either represented by the foreign key not allowed null on the referencing relation side, or by a general constraint form the referenced relation side. When the foreign key is part of the primary key for the referencing relation there is no need for the 'not allowed null entry' because this is already guaranteed using key constraints.

Other constraints located in the Constraints part of the conceptual data model were expressed using the relational algebra **constraint** expression if this was possible. If this was not possible the representation was left until the implementation stage to make use of triggers. For example the constraint on the holiday quotas can be maintained by a trigger that makes use of aggregate function count() in implementation stage.

3.2.4 Database Implementation

Introduction:

This section discusses how the specification of the delivery office logical schema (relational model shown in Appendix C) is used to implement a schema and its associated database. This implementation is done using SQL Data Definition Language (DDL) statements (Appendix D). The output is a specification of all the tables and their constraints in a database.

The delivery office database is implemented using Sybase SQL Anywhere Database Management system. This is a relational database system, and the database itself is stored on one or more disk drives, and consists of the following objects [Sybase, Inc. 1995, ch.2, pp.2]:

Database objects	**Description**
Tables	Hold the information in the database
Keys	Relate the information in one table to that in another
Indexes	Allow quick access to information in the database
Views	Are computed tables
Stored procedures	Hold queries and commands that may be executed by any client application (stored procedures are not available in the SQL Anywhere Desktop Runtime system)
Triggers	Assist in maintaining the integrity of the information in the database (triggers are not available in the SQL Anywhere Desktop Runtime system)
System tables	Hold the information about the structure of the database

Sybase SQL Anywhere makes use of SQL Anywhere Version 5.0 as a Data Definition Language (DDL) and Data Manipulation Language (DML). SQL Anywhere conforms to SQL92 Entry level standards and has some features which are not part of standard SQL. That is why in some situations I had to make decisions on how to implement some of the database definitions that can not be defined in the same way they were defined in the relational data model.

InfoMaker 5.0 was used as direct entry database tool by entering and executing SQL DDL and DML statements (see appendix D) used to create and populate the delivery office database in the Database Administration window. This is shown in Fig. 3.7 below as the direct entry database tool. InfoMaker also contains an Open Database Connectivity (ODBC[*]) interface, this enables InfoMaker to access data sources for which an ODBC driver exist.

[*] **ODBC** is a standard database access method developed by Microsoft Corporation. The goal of ODBC is to make it possible to access any data from any application, regardless of which DBMS is handling the data. ODBC manages this by inserting a middle layer, called a database driver, between an application and the DBMS. The purpose of this layer is to translate the application's data queries into commands that the DBMS understands. For this to work, both the application and the DBMS must be ODBC-compliant; that is, the application must be capable of issuing ODBC commands and the DBMS must be capable of responding to them.

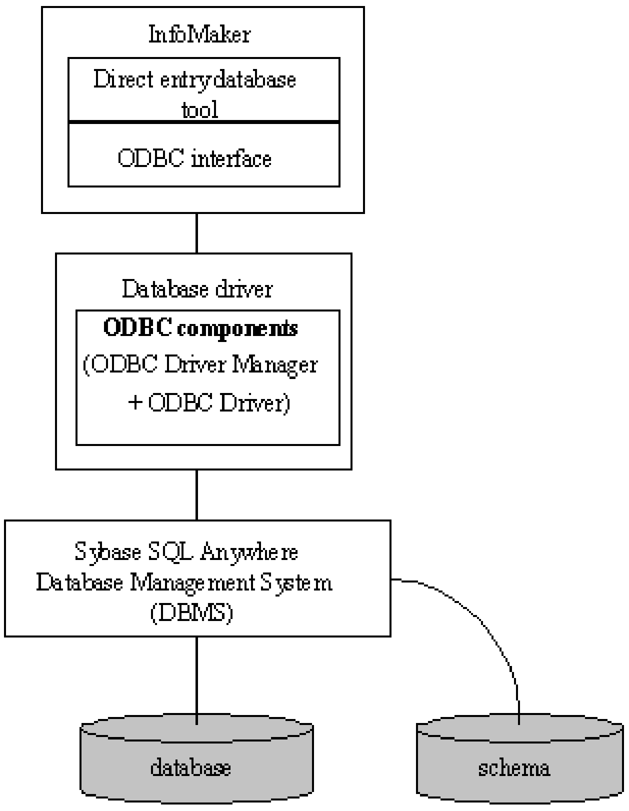

InfoMaker
Direct entry database tool
ODBC interface
Database driver
ODBC components
(ODBC Driver Manager + ODBC Driver)
Sybase SQL Anywhere Database Management System (DBMS)
database
schema

Figure 3.7 *Model showing use of InfoMaker as direct entry database tool for interaction with Sybase SQL Anywhere DBMS*

SQL Data Definition Statements

The naming convention being used for the SQL tables differs from the one that was used for the conceptual data model, for example according to the convention I've used, the attribute named PayNo is transformed to the column named **pay_no**. This is the common convention for SQL names, and arises because SQL does not distinguish between upper and lower case letters (unless enclosed in double quotes, such as “PayNo”), and underscore is the only non-alphanumeric character allowed in SQL names.

Domains

I’ve started the database implementation by defining a set of domains to provide a common definition that can be shared by a number of columns (see appendix D.1.1). This is particularly useful when a domain is defined as more than just a data type, including both constraints and default values, which has the benefit of ensuring consistency between columns. For example I've defined the following domain for delivery office column that exists in **company_vehicle**, **walk**, **delivery_office** and **employee** tables:

```
CREATE DOMAIN delivery_office_names AS VARCHAR(10)
    CHECK (@col IN ('south', 'north', 'east', 'west'));
```

SQL Anywhere supports both CREATE DATATYPE and CREATE DOMAIN statements. I’ve used the later because CREATE DOMAIN is the syntax used in the draft SQL/3 standard [Sybase, Inc. 1995, ch.27, pp.6]. When defining a CHECK condition on a user-defined data type in SQL Anywhere, any variable prefixed by @ is replaced by the name of the column when the CHECK condition is evaluated. For example the **work_status** data type accepts only 'ft' and 'pt' character strings:

```
CREATE DOMAIN work_status AS CHAR(2)
   NOT NULL
   CHECK(@col IN ('pt', 'ft'));
```

Any variable name prefixed with @ could be used instead of @col. Any column defined as **work_status** type accepts only 'ft' and 'pt' character strings unless it has a CHECK condition explicitly defined.

The attribute of the data type can be overridden if needed by explicitly providing attributes for the column. A column created on data type **work_status** with NULL values explicitly allowed does allow NULLs, regardless of the setting in the **work_status** data type.

Tables Definition

The next step was to define the database table, using the SQL CREATE TABLE statement (see appendix D.1.2). The general format of this statement in SQL Anywhere for a simple table is:

```
CREATE TABLE <table-name> (<column-definition [column-constraint …]>,…,
       PRIMARY KEY (column-name,…),
       [UNIQUE ( column-name, ... ),]
       [[NOT NULL] FOREIGN KEY [role-name] [(column-name, ... )]
       REFERENCES table-name [( column-name )] [ actions ],]
       [CHECK ( condition )]);
```

The non SQL terms used are defined as follows:

<u>column-definition</u>:

column-name data-type [**NOT NULL**] [**DEFAULT** default-value]

<u>default-value:</u>

This can be a string, a number, **AUTOINCREMENT**, **CURRENT DATE**, **CURRENT TIME**, **CURRENT TIMESTAMP**, **NULL**, **USER**, or a (constant-expression).

<u>column-constraint:</u>

CHECK (condition)

<u>action:</u>

ON [**UPDATE** | **DELETE**]...[**CASCADE** | **SET NULL** | **SET DEFAULT** | **RESTRICT**]

The Delivery Office relational model contains 19 relations (Appendix C) that needed to be transformed to SQL tables. The tables had to be created in a specific order; this is because of the referential integrity between different tables, besides some of the table constraints and referential integrity constraints were added later using the ALTER TABLE statement.

The general procedure I've used to create tables and constraints can be summarised in figure 3.8 below.

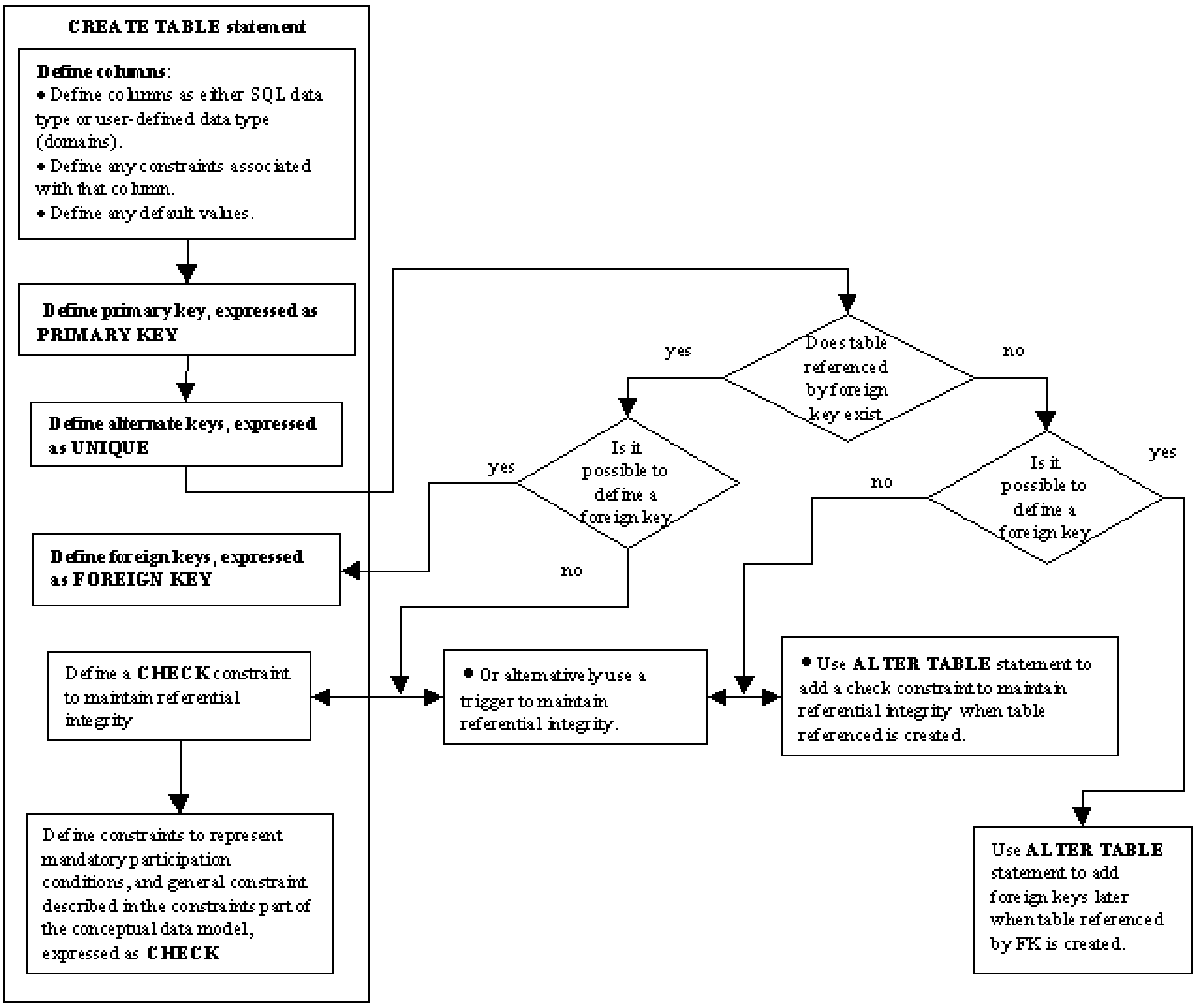

CREATE TABLE statement
Define columns:
• Define columns as either SQL data type or user-defined data type (domains).
• Define any constraints associated with that column.
• Define any default values.
Define primary key, expressed as PRIMARY KEY
Define alternate keys, expressed as UNIQUE
Define foreign keys, expressed as FOREIGN KEY
Define a CHECK constraint to maintain referential integrity
Define constraints to represent mandatory participation conditions, and general constraint described in the constraints part of the conceptual data model, expressed as CHECK
Does table referenced by foreign key exist
yes
no
Is it possible to define a foreign key
yes
no
Is it possible to define a foreign key
no
yes
• Or alternatively use a trigger to maintain referential integrity.
• Use ALTER TABLE statement to add a check constraint to maintain referential integrity when table referenced is created.
Use ALTER TABLE statement to add foreign keys later when table referenced by FK is created.

Figure 3.8. General steps for implementing tables and constraints used for the Delivery Office database

Defining columns

The first step towards defining table was to define the columns and any associated constraints such as 'NOT NULL'. The columns were either defined as SQL data type such as the **street_name** definition in **postal_address** table, or defined on a domain such as the **office** column on the same table:

```
CREATE TABLE postal_address
         (post_code post_codes,
          building_no building_numbers,
          street_name VARCHAR(20) NOT NULL,
            postal_area VARCHAR(20),
          town VARCHAR(14) NOT NULL,
          walk_no walk_numbers NOT NULL,
          office delivery_office_names NOT NULL,
          PRIMARY KEY (post_code, building_no))!
```

A CHECK condition can be applied to values in a single column, to ensure that they satisfy rules. These rules may be rules that data must satisfy in order to be reasonable, or they may be more rigid rules that reflect organization policies and procedures. For example, consider the constraint defined on the **overtime_availability** column in the **employee** table shown below. This definition limits the values for **overtime_availability** to 'yes' or 'no' character strings. I've also defined some default values for some of the columns such as the **grade** column with default value 'opg'. I've defined some columns as VARCHAR and others as CHAR data types, this was done to reflect a character string defined as CHAR has a fixed number of characters such as driving licence numbers which have a fixed length of 16 characters. On the other hand columns defined as VARCHAR such as the person names, which can vary in length, but not to exceed 20 characters.

```
CREATE TABLE employee
          (pay_no employee_numbers,
          employee_name person_names,
          date_of_birth DATE NOT NULL,
          date_of_entry DATE NOT NULL,
          grade VARCHAR(10) CHECK (grade IN ('opg','opgdriver','ex_phg')) NOT NULL DEFAULT
                  'opg',
          phone phone_numbers,
          badge_no VARCHAR(4),
          skills LONG VARCHAR,
          overtime_availability VARCHAR(3) CHECK (overtime_availability IN ('yes', 'no')),
          office delivery_office_names NOT NULL,
          house_no building_numbers,
          post_code post_codes,
          status work_status,
          PRIMARY KEY (pay_no),
          UNIQUE (badge_no),
          // IsStaffedBy
          FOREIGN KEY (office) REFERENCES delivery_office,
          // defined as not null in the domain definition which reflects
          // madatory participation condition with regards to postal_address & delivery_office
          // IsOccupiedBy
          FOREIGN KEY (post_code, house_no) REFERENCES postal_address )!
```

Declaring primary and alternate keys

The primary keys definition was a straightforward transformation of the primary keys from the relational model as shown in employee table above. Alternate keys were defined using the SQL UNIQUE constraints, however because SQL Anywhere imposes a NOT NULL on the UNIQUE constraints I had to make some decisions about the implementing the alternate keys which are allowed null in the relational model.

The only alternate keys declared in the relational model to represent the meaning of the data are VehicleNumber and BadgeNo, both declared as not null, so there was no problem implementing those. However the problem arose when implementing the alternate keys that resulted from 1:1 relationships (HasCover, Covers, Permanently Covers, MayBe and IsA as shown in Figure 3.9) representation in the relational model. When representing 1:1 relationships, one relation had the foreign key declared as an alternate key, however when the participation condition is optional the alternate key needed to be allowed null. Because all the 1:1 relationships in Delivery Office conceptual data model are mandatory from one side and optional from the other I've decided to modify the relational model so that the relation that declares the foreign key for 1:1 relationship is on the mandatory side. This automatically implied declaring the alternate key as not allowed null.

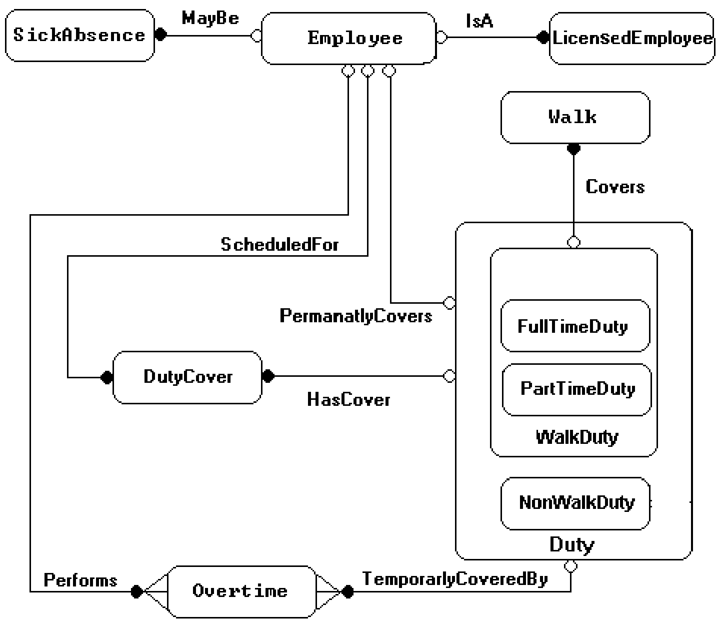


Figure 3.9 Part of the Delivery Office E-R diagram showing 1:1 relationships

However alternate keys defined in the FullTimeDuty, PartTimeDuty and NonWalkDuty relations were not implemented using a UNIQUE constraint, because this would mean the optional participation condition with regards to PermanentlyCovers would become mandatory. The uniqueness of these columns was maintained using triggers[4]. The following is a trigger that ensures that column **duty_holder** is unique in table **full_time_duty**:

```
CREATE TRIGGER add_modify_fulltime_duty
        BEFORE INSERT, UPDATE ON full_time_duty
        REFERENCING NEW AS new_full_time_duty
        FOR EACH ROW
        BEGIN
                DECLARE invalid_duty_details EXCEPTION FOR SQLSTATE '99999';
                IF ((new_full_time_duty.duty_holder IN (SELECT duty_holder FROM
                    full_time_duty))
                THEN
                        SIGNAL invalid_duty_details;
                END IF;
```

[4] SQL Anywhere provides triggers. A trigger is a procedure stored in the database that is executed automatically whenever the information in a specified table is altered. Triggers are a powerful mechanism for database administrators and developers to ensure that data is kept reliable.

```
          END!
```

Using a CHECK constraint was not possible in this case, because it is a static constraint that holds all the time, however a trigger is a dynamic constraint that is fired only when a table is modified. In this case we need to check that every new added or updated tuple does not contain a **duty_holder** value that exists on the table, thus ensuring the uniqueness of **duty_holder**.

Declaring foreign keys

As shown in Figure 3.8, when defining tables foreign keys were either defined as part of the tables definition if the referenced table existed or defined later after the referenced table was created using ALTER TABLE statement. A column defined as a foreign had DEFAULT NULL or NOT NULL constraint according to the participation condition from the referencing table side. For example a DEFAULT NULL for the foreign key represents an optional participation condition (as shown in the example below for **licenced_employee** table), on the other hand NOT NULL represents a mandatory one.

```
CREATE TABLE licensed_employee
          (license_no license_numbers,
           vehicle_size vehicle_sizes DEFAULT NULL,
           pay_no employee_numbers NOT NULL,
           vehicle_no vehicle_reg_numbers DEFAULT NULL,
           PRIMARY KEY (license_no),
           UNIQUE (pay_no),
          // IsA
           FOREIGN KEY (pay_no) REFERENCES employee,
          //IsDrivenBy
           FOREIGN KEY (vehicle_no) REFERENCES company_vehicle)!
```

As mentioned before it's not possible to define a foreign key **duty_no** in the **duty_cover**, **over_time** and **walk** tables to reference to reference the three duty tables. This is because a foreign key can only references one table. One way around this is to use a CHECK clause to maintain referential integrity. I've declared a CHECK condition on the **duty_cover** table for this purpose as follows:

```
ALTER TABLE duty_cover
          ADD CHECK (
                              (duty_no IN (SELECT duty_no FROM part_time_duty))
                              OR
                              (duty_no IN (SELECT duty_no FROM full_time_duty))
                              OR
                              (duty_no IN (SELECT duty_no FROM indoors_duty)))!
```

The same check was added to the **over_time** table, but not to the **walk** table because besides maintaining the referential integrity the constraint needs to ensure that **walk** tuples with column **status** equal 'ft' reference full_time_duty. And tuples with column **status** equal 'pt' reference **part_time_duty**. This requires the use of the SQL IF construct, thus I've used the following trigger to enforce this sophisticated CHECK conditions:

```
CREATE TRIGGER add_modify_walk
          BEFORE INSERT, UPDATE ON walk
          REFERENCING NEW AS new_walk
          FOR EACH ROW
          BEGIN
                    DECLARE invalid_walk_details EXCEPTION FOR SQLSTATE '99999';
```

```
                IF (new_walk.status = 'PT')
                THEN
                        IF (NOT (new_walk.duty_no IN (SELECT duty_no FROM
                            part_time_duty)))
                        THEN
                                SIGNAL invalid_walk_details;
                        END IF;
                END IF;

                IF (new_walk.status = 'FT')
                THEN
                IF (NOT (new_walk.duty_no IN (SELECT duty_no FROM
                         full_time_duty)))
                        THEN
                                SIGNAL invalid_walk_details;
                        END IF;

                END IF;
        END!
```

Summary of database tables

I've created the following tables[5] by executing the CREATE TABLE statement shown in Appendix D.1.2:

booked(week_no, pay_no)
company_vehicle(reg_no, vehicle_no, mot_due_date, vehicle_size, model, office_name)
compliant(reference_no, post_code, building_no, pay_no, details, date)
conduct(reference_no, conduct_type, notes, staff_no, date)
delivery_office(office_name, telephone_no, manager_name, building_no, post_code)
duty_cover(duty_no, cover, reason)
duty_second_part(duty_no, rotation_week, day_off, second_part)
employee (pay_no, employee_name, date_of_birth, date_of_entry, grade, phone, badge_no, skills, overtime_availability, office, house_no, post_code, status)
full_time_duty(duty_no, duty_holder, day_off_cover)
licensed_employee(license_no, vehicle_size, pay_no, vehicle_no)
mail_drop(walk_no, office, drop_point, details, duty_no)
non_walk_duty(duty_no, pay_no, duty_details)
over_time(duty_no, pay_no, duration)
part_time_duty(duty_no, duty_holder, duty_hours)
postal_address(post_code, building_no, street_name, postal_area, town, walk_no, office)
private_vehicle(reg_no, model, color, license_no)

[5] Rather than giving the complete CREATE TABLE or CREATE VIEW statements for each table and view, it is simpler to specify the columns of a table or a view in the same way used for the attributes of entity types and relations.

sick_absence(pay_no, date_commenced, reason)
walk(walk_number, office_name, walk_type, delivery_method, remarks, status)
week(week_no, quota)

Extra Tables

I've also created the following extra tables that doesn't exist in the conceptual data model:

current_week(week_no)
duty_cover_info(duty_no, reason, cover_type)

The table **current_week** should have only one row and only accessed through the following procedure (see Appendix D.1.7):

```
// A procedure to set the current week in the finantial year
CREATE PROCEDURE set_week(IN weekno SMALLINT)
        BEGIN
           UPDATE current_week
           SET week_no = weekno;
        END!
```

This table stores the number of the current week in the financial year. This single value is then used by different procedures, functions and triggers to query the database. To ensure this works, access to table **current_week** needs to be restricted and the only access allowed should be through the set_week() procedure.
The table **duty_cover_info** is a summary table used to store some data that involves some complex derivation carried out by the process_duty_cover() procedure (see appendix D.1.7). Again because the data in this table needs to be processed and joined with other tables access to it needs to be restricted and only allowed through the process_duty_cover().

The database also contained some extra tables called system tables. These tables hold the information about the structure of the database (database schema).

Defining constraints

The general constraints defined for the Delivery Office database can be summarised as follows:

- Constraints used to represent mandatory participation conditions; at one end expressed as NOT NULL for a foreign key and at the other end as more complex CHECK clause. These were defined as follows for tables **delivery_office** and **walk**:

```
// represent mandatory participation with respect to IsStaffedBy
ALTER TABLE delivery_office
        ADD CHECK (office_name IN (SELECT office FROM employee))!

// represent mandatory participation with respect to ConsistsOf
ALTER TABLE delivery_office
        ADD CHECK (office_name IN (SELECT office_name FROM walk))!

// represent mandatory participation with respect to HasAsset
ALTER TABLE delivery_office
        ADD CHECK (office_name IN (SELECT office_name FROM company_vehicle))!
```

```
// represent mandatory participation with respect to IsAcollectionOf from Walk side
ALTER TABLE walk
        ADD CHECK
                (EXISTS (SELECT * FROM postal_address
                        WHERE (postal_address.walk_no = walk.walk_number)
                                AND (postal_address.office = walk.office_name)))!
```

- Constraints described in the constraints part of the conceptual data model (see appendix B.3). I've used triggers when inserting or updating some of the tables. These triggers combine all the constraints that affect the same table together. This is because triggers have more advantages over the CHECK clause, first they can handle very sophisticated constraints involving the IF construct, second they can dynamically update or perform an operation on a table. For example, here is the trigger that ensures the total number of tuples of table **booked** doesn't exceed the quota for a specific week:

```
// ensures that annual leave doesn't exceed the quota
CREATE TRIGGER add_annual_leave
        BEFORE INSERT, UPDATE ON bookedholiday
        REFERENCING NEW AS new_annual_leave
        FOR EACH ROW
        BEGIN
                DECLARE leave_exceed_quota EXCEPTION FOR SQLSTATE '99999';
                IF ((SELECT COUNT(pay_no) FROM bookedholiday WHERE
                                bookedholiday.week_no = new_annual_leave.week_no)
                                >= (SELECT quota FROM week WHERE week.week_no =
                                new_annual_leave.week_no)) THEN
                        SIGNAL leave_exceed_quota;
                END IF;
        END!
```

The following is the trigger that maintains all the general constraints that affects the **part_time_duty** table:

```
CREATE TRIGGER add_modify_parttime_duty
        BEFORE INSERT, UPDATE ON part_time_duty
        REFERENCING NEW AS new_part_time_duty
        FOR EACH ROW
        BEGIN
                DECLARE invalid_duty_details EXCEPTION FOR SQLSTATE '99999';
                IF (
                   //// ensures that duty_holder is unique
                   (new_part_time_duty.duty_holder IN (SELECT duty_holder FROM
                   part_time_duty))

                   //// ensures duty_holder does not exist in full_time_duty or  non_walk_duty
                   OR
                   (new_part_time_duty.duty_holder IN (SELECT duty_holder FROM
                   full_time_duty))
                   OR
                   (new_part_time_duty.duty_holder IN (SELECT pay_no FROM
                    indoors_duty))

                   //// ensures that duty_no is a primary key across the subtypes of duty
                   OR
                   (new_part_time_duty.duty_no IN (SELECT duty_no FROM full_time_duty))
                   OR
                   (new_part_time_duty.duty_no IN (SELECT duty_no FROM indoors_duty))

                    //// ensure only part time employees can participate in PermanentlyCovers
                   with part_time_duty
                   OR
                   ((SELECT status FROM employee WHERE employee.pay_no =
                    new_part_time_duty.duty_holder) <> 'PT')

                   //// ensure only employees and duties from the same office can participate
                   in PermanentlyCovers
                   OR
                   ((SELECT LEFT (employee.office,1) FROM employee WHERE
                   employee.pay_no   = new_part_time_duty.duty_holder)
                                   <> (SELECT LEFT (new_part_time_duty.duty_no,1) ) ) )
                THEN
                        SIGNAL invalid_duty_details;
                END IF;
        END!
```

The only constraint that was defined in a procedure is the constraint on the **over_time** table. This constraint ensures only staff available for **over_time** can perform it, however, according to the client this is a changing, because on some delivery offices staff might be persuaded to do overtime if the operation requires. Besides the constraints on the over_time table may vary from office to office such as controlling the duration of the overtime. I've created the procedure add_over_time() that can be used for this purpose. All access to overtime table should be restrict and controlled through procedures that contains the constraints which are not part of the table and might change with time. Different procedures can be tailored for different overtime restriction requirements.

On the other hand constraints that ensure the meaning of the data are included in a trigger defined on the **over_time** table as follows:

```
  CREATE TRIGGER add_modify_over_time
          BEFORE INSERT, UPDATE ON over_time
          REFERENCING NEW AS new_over_time
          FOR EACH ROW
          BEGIN
                  DECLARE invalid_overtime_details EXCEPTION FOR SQLSTATE '99999';

                   //// Make sure overtime is used to cover a vacant or absent duty
                   IF (NOT (new_over_time.duty_no IN (SELECT duty_number FROM
                   absent_duties_after_cover)))
                   THEN
                            SIGNAL invalid_overtime_details;
                   END IF;

                   //// Make sure the employee performing the overtime is not absent from work
                   IF ((new_over_time.pay_no IN (SELECT pay_number FROM
                   sick_employees))
                               OR
                               (new_over_time.pay_no IN (SELECT pay_no FROM
                                                         bookedholiday
                                     WHERE
```

```
                        week_no = (SELECT week_no FROM current_week))))
                THEN
                        SIGNAL invalid_overtime_details;
                END IF;
        END!
```

Defining views

I've defined a set of views for the daily most used queries by the database users. Views have advantages over using base tables directly, these can be summarised as follows:

- usability, because specific data required by users can be defined as a view, thus providing a simplified way to access the data;
- flexibility, because a view enables changes to be made to base tables without affecting the users' view of data, providing data independence;
- access control, because defining users' privileges for views, rather than base tables, allows more precise control on the data made available to users.

Table 4.2 below summarises the Delivery Office database views and a brief description their purpose. The actual view definitions are shown in Appendix D.1.4

Table 4.2 *The Delivery Office database views and their descriptions*

View columns	**Description**
employee_car (employee_name, licence_no, reg_no, model)	List of all employee who have private vehicle and the vehicle details
sick_employees (name, office, pay_number, grade, phone, address)	List some for sick employees including full address which makes use of the full_address () function
all_duties (duty_no, duty_holder, status)	List of all duties their status and holders
employee_duty_walk (name, office, pay_number, grade, duty_number, walk_type, delivery_method, status)	List details of all duties, their holders and walks
vacant_absent_duties (duty_number, reason, status)	List details of vacant duties and duties covered by absent staff
absent_duties_after_cover (duty_number, reason, status)	From the list above only list those duties which are not covered by duty cover

spare_employees (name, pay_number, grade, office, status)	List some of the details of employees not covering any duty
display_duty_walk_info ("Name", "Office", "Pay_number", "Grade", "Duty_number", "Walk_type", "Delivery_method", "Status", "Reason", "Cover_type")	This is the duty coverage summary list, the view is carried by joining employee_duty_walk with duty_cover_info
where_employees_live (name, office, pay_number, grade, phone, address)	List some the details of all employees and their addresses, also makes use of the full_address () function
available_spare_employees (name, pay_number, grade, office, status)	List some details for all spare employees who are not absent from work
absent_employees (pay_no)	List the pay numbers of all absent employees

Defining functions & procedures

I've also defined some functions and procedures to be used to process the data stored in the Delivery Office database. SQL Anywhere supports procedures and user-defined functions. These can include control statements that allow repetition (LOOP statement) and conditional execution (IF statement and CASE statement) of SQL statement. These procedures and functions are then stored in the database for use by all application.

Using database stored procedures and function have advantages over using external ones written in other programming languages. These benefits can be summarised as follow:

- **Standardization**: The action to be performed on the data is coded and stored in one place, and can be called by any application. Any new changes to this action need only be made in one place.
- **Efficiency**: Stored procedures provide efficiency over user process procedures because only the results need to be sent to the user process and not all data.
- **Security**: Stored procedure can be use to restrict or define the action performed on the data by all applications.

The Delivery Office database procedures and functions are summarised in Table 4.3 below. The actual definitions are included in Appendices D.1.6 and D.1.7.

Table 4.3 *The Delivery Office database procedures & functions and their descriptions*

Procedure or function heading	**Description**
FUNCTION duty_to_office()	Takes a duty number and returns the delivery office to which the duty belongs.
FUNCTION full_address()	Takes a postcode and a building number and returns a full address as a character string
PROCEDURE set_week()	Sets the current week in the financial year, this should the only access to table current_week
PROCEDURE add_over_time()	Takes some values and enter then into the over_time table. This should be the only ADD access to the

	over_time tables. Also used to enforce the changing over_time requirements.
PROCEDURE process_duty_cover()	Used to populate the duty_cover_info and set the cover_type column.

Table creation order

I've used the steps shown below in creating the Delivery Office database. I needed to organise these steps and execute them in an order that ensures all referenced tables in foreign keys definitions or in general constraints are available and populated. These steps include table definition, definition of constraints, definition of referential integrity and the population of the tables. Each table was created with primary and alternate key definition if that didn't involve the use of a trigger and was immediately populated after creation, and because most of the constraints such as triggers didn't exist at that time, I ensured all data entered is consistent and valid with regard to the constraints to be defined. This was important because when triggers are used, the database may already contain inconsistent data that goes unnoticed. Triggers only come to action when they are created and they enforce conditions only when data is added or modified in contrast to static constraints that hold all the time.

- Create all domains
- Create the **postal_address** table without defining the foreign key than references **walk**.
- Create the delivery_office table including foreign key, but without defining the general constraints that represent the mandatory participation conditions with regards to HasAsset, IsStaffedBy and ConsistsOf.
- Create the employee table including foreign keys.
- Create the walk table including foreign keys and the constraint that represent the mandatory participation condition with regards to IsAcollectionOf.
- Alter table postal_address, add foreign key to reference walk.
- Create table compay_vehicle including foreign keys.
- Create table licensed_employee including foreign keys.
- Alter table delivery_office, define the constraints that represent the mandatory participation conditions with regards to HasAsset, IsStaffedBy and ConsistsOf.
- Create table private_vehicle including foreign keys.
- Create table complaint including foreign keys.
- Create table conduct including foreign keys.
- Create table week.
- Create table booked including foreign keys.
- Create table sick_absence including foreign keys.
- Create table non_walk_duty including foreign keys, with defining the trigger.

- Create table full_time_duty including foreign keys, with defining the trigger.
- Create table part_time_duty including foreign keys, with defining the trigger.
- Create table duty_cover including foreign keys, with defining the trigger.
- Create table over_time including foreign keys, with defining the trigger.
- Create table duty_second_part including foreign keys.
- Create table mail_drop including foreign keys.
- Create table current_week.
- Create table duty_cover_info.
- Create the following triggers: add_annual_leave, add_modify_fulltime_duty, add_modifiy_duty_cover, add_modify_over_time, add_modify_sick_absence, add_modifiy_part_time_duty, add_modify_non_walk_duty, add_modify_walk
- Create database functions and procedures.
- Create database views.

Populating Tables

The tables were populated using INSERT INTO statement (Appendix D.2.1). Consider the following example for the **deliver_office** table:

```
INSERT INTO delivery_office VALUES  ('West','02476557263','G Sandeep','40','CV1 1AA')!
INSERT INTO delivery_office VALUES  ('South','02476557264','D Snowdon','40','CV1 1AA')!
INSERT INTO delivery_office VALUES  ('South','02476557264','D Snowdon','40','CV1 1AA')!
INSERT INTO delivery_office VALUES  ('South','02476557264','D Snowdon','40','CV1 1AA')!
```

Now we can display the delivery_office table by executing the following SELECT statement:

```
SELECT * FROM delivery_office!
```

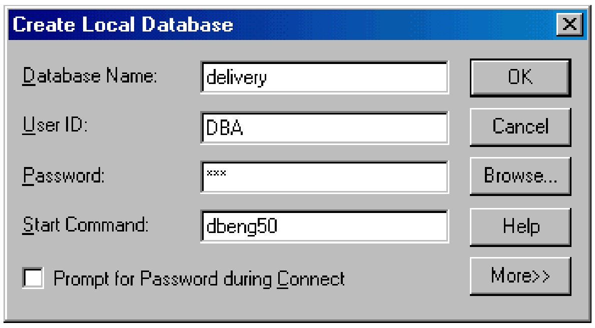
The resulting table is shown in table 4.1 below. The data of the Delivery Office database base tables is included in Appendix E. And the SQL INSERT INTO statements used to populate the tables is shown in Appendix D.2.1.

Table 4.1 delivery_office Table

office_name	telephone_no	manager_name	building_no	post_code
East	02476557261	Jon Campbell	40	CV1 1AA
North	02476557262	Steve Moore	40	CV1 1AA
West	02476557263	G Sandeep	40	CV1 1AA
South	02476557264	D Snowdon	40	CV1 1AA

Database files and indexes

For the delivery office database, I've created a local database by using InfoMaker. I've done this by opening the database widow and selecting the option, Create Database. In the resulting dialogue box, entitled Create Local Database (Figure 4.10), I've entered the database name as 'delivery' this is also the file name used to store the database. The default user ID and password are 'DBA' (Database Administrator) and 'sql' respectively.


Create Local Database

Database Name:	delivery	OK
User ID:	DBA	Cancel
Password:	***	Browse...
Start Command:	dbeng50	Help
☐ Prompt for Password during Connect		More>>

Figure 4.10 Create Local Database dialogue box

When a database is initialised, it is composed of one file (database_name.db). This first database file is called the root file. All database objects and all data are placed in the root file. For many databases, it is convenient to keep the database as a single file.

Each SQL Anywhere database file has a maximum size of 2 GB. To divide large databases among more than one file the user can create a new database file, or dbspace, using the CREATE DBSPACE statement. A new dbspace may be on the same disk drive as the root file or on another disk drive. The user must have DBA authority to create new database files [Sybase Inc. 1995 ch.15, pp.3].

When creating the Delivery Office database, the following two files were created

- delivery.db
- delivery.log

The file 'delivery.log' is used to store all changes to the database in the order that they occur. Inserts, updates, deletes, commits, rollbacks, and database schema changes are all logged. The transaction log is called a forward log file.

An index is a column or set of columns you identify for the purpose of improving database performance when searching for the data specified by the index. The user can index a column or set of columns if information from the columns will be needed frequently. Primary and foreign keys are special examples of indexes. However for the Delivery Office database no indexes were defined, this is because the database tables are small so it's quicker to search all the rows of the tables than first search for an index entry and then retrieve just the rows required.

Chapter 4

Results & Discussion

4.1 Client requirements

One of the outcomes of the work carried out in this project is the Delivery Office database. The development steps (Figure 4.2 in subsection 4.2.2) followed in this project aimed at ensuring that the database has the following important properties:

- **completeness** in satisfying user requirements ensures that users can access the data they want;
- maintaining **integrity** of the data, including both consistency (no contradictory data) and correctness (no invalid data);
- having **flexibility** to change, that database can be changed to satisfy user requirements without excessive effort;
- enabling execution **efficiency** ensures that users do not have to wait for long times when accessing the data**;**
- providing **usability** ensures the ease of use of the data.

Most of the validation done was to ensure that the database maintains the integrity of the data, i.e. all the constraints function as required by the client. Constraints were either implemented using CHECK clauses or using triggers, and they were tested by trying to update the database in a way that violated the constraints. Most of the testing went towards ensuring that the tables resulted from the implementation of the Duty super type represent the meaning of data. This was not easy because the concept of entity subtypes is not supported in the relational schema model.

In the following subsection we discuss how the database was tested and validated inline with the above properties to ensure that it meets the client's data and processing requirements.

4.1.1 Data requirements

The client data requirements were included in the statement of data requirements (Appendix A). The database was developed according to those requirements, now it's the time to ensure that the database satisfies those requirements.

The client required a list to maintain a record of its physical assets employees, vehicles, walks, delivery offices and addresses. Besides records to maintain other administration tasks such as a record of annual leave, sick leave, customer complaints and conduct of employees. The following sample queries were used to ensure the database meets its requirements:

Purpose: Display all employees that work in the South delivery office
Query: **SELECT** * **FROM** employee **WHERE** office = 'South'!
Results:

pay_no	employee_name	date_of_birth	ate_of_ent	grade	phone	badge_no	skills	ime_availa	office	house_no	post_code	status
23771	M Thomas	22/10/65	24/11/98	OPGDriver		1601	sign langua	yes	South	6	CV2 4BA	FT
77890	N George	02/02/80	08/05/00	OPG	02476885965	5532		no	South	105	CV6 5AH	FT
77564	A Ahmed	25/06/62	09/01/80	OPG	07885641365	8542		no	South	89	CV6 5AH	FT
58470	S Michael	22/07/71	20/06/99	OPG	02476885414	2001		no	South	50	CV2 4JO	FT
55355	P James	23/05/76	08/10/98	OPG	02476100123	5103		yes	South	20	CV3 5FK	PT
96742	B Muhsin	25/07/77	03/09/00	ex_PHG	02476947586	3356		yes	South	93	CV3 5FK	FT
20903	B Chandler	29/10/78	10/07/01	OPG	07985252751	1976		yes	South	322	CV3 2ZX	FT

Purpose: Display the names, pay numbers & licence numbers of all employees with driving licences.
Query: **SELECT** employee_name, license_no, employee.pay_no **FROM** employee, licensed_employee **WHERE** employee.pay_no = licensed_employee.pay_no!
Results:

employee_name	license_no	pay_no
M John	JOHNA862088MA3ER	86961
E Magee	MAGEE563232ER3OM	99856
S Mohammed	MOHAM982365SM8TY	88974
E Francis	FRANC981234EN1ZX	10088
M Thomas	THOMA122334MT8BN	23771
M Philip	PHILI657890MK0OI	78541
M Rushton	RUSHT563123MH7WE	87961
Y Jones	JONES501389YA8NM	12458
J Singh	SINGH096354JS4CV	19700
J Wilcox	WILCO455668JM1QW	54213
O Steven	STEVE908765OD4TY	99471
S Fox	FOXXY444908SP0TN	31692
J McCanze	MCCAN012799JK5XY	10023
N George	GEORG111090NJ9LP	77890
A Ahmed	AHMED480546AM1TU	77564
S Michael	MICHA009871SR6BG	58470

Purpose: Display the names, addresses and phone numbers of all employees that work in the North delivery office in alphabetical order.

Query: **SELECT** name, phone, address **FROM** where_employees_live **WHERE** office = 'North' **ORDER BY** "name"!
Results:

name	phone	address
A Ali	02476854124	65, The Avenue, Brownshill Green, COVENTRY, CV5 3SC
A Eaton	02476441230	89, Benedictine Road , Earlsdon, COVENTRY, CV4 4KB
A Wilford	02476737684	3, The Monks Croft, Alderman's Green, COVENTRY, CV2 4BA
E Francis	02476884736	69, The Avenue, Brownshill Green, COVENTRY, CV5 3SC
J McCanze	02476778830	5, Stoney Road , Upper Stoke, COVENTRY, CV2 1KY
R Sims	02476453788	1, Swifts Corner, Little Heath, COVENTRY, CV6 5RT
S Mohammed	07988614214	2, Swifts Corner, Little Heath, COVENTRY, CV6 5RT

Purpose: Display the names, pay numbers and offices of all absent employees.
Query: **SELECT** employee_name, employee.pay_no, office **FROM** employee, absent_employees **WHERE** employee.pay_no = absent_employees.pay_no!
Results:

employee_name	pay_no	office
S Wesley	21652	East
R Sims	24661	North
S McDonald	25600	East
P James	55355	South
S Mohammed	88974	North
B Muhsin	96742	South

Purpose: Display the names licence numbers and car details of all employees who own cars.
Query: **SELECT** * **FROM** employee_car!
Results:

employee_name	license_no	reg_no	model
E Magee	MAGEE563232ER3OM	M505 RAU	Ford Escort
M Thomas	THOMA122334MT8BN	T485 NBV	Vauxhall Vectra
O Steven	STEVE908765OD4TY	CK02MNA	Peugeot 406
Y Jones	JONES501389YA8NM	DN51JKL	Honda Civic
S Fox	FOXXY444908SP0TN	X981 SHR	Nissan Micra
J McCanze	MCCAN012799JK5XY	D123 TEW	Ford Fiesta
S Mohammed	MOHAM982365SM8TY	L459 ZCV	Rover Tomcat
M Philip	PHILI657890MK0OI	P45 ERT	Renault Clio
A Ahmed	AHMED480546AM1TU	J543 NAZ	BMW 320i SE

Purpose: Display the company vehicles details for all vehicles that need to have an MOT before December 2003.
Query: **SELECT** * **FROM** company_vehicle **WHERE** ((**MONTH**(mot_due_date) < 12) **AND** (**YEAR**(mot_due_date) < 2003)) **OR** ((**MONTH**(mot_due_date) < 12) **AND** (**YEAR**(mot_due_date) = 2003))!
Results:

reg_no	vehicle_no	mot_due_date	vehicle_size	model	office_name
X467 GFD	CV1	12/08/03	150	FORD	

T765 VBC	CV3	07/11/02	250	FORD	North
Y543 RAD	CV5	27/11/02	150	RENAULT	South
Y786 MUY	CV6	10/09/03	50	VAUXHALL	North

4.1.2 Data processing requirements

The main data processing requirement for the D.O. is to be able to identify daily the duties that are not covered, and to use staff to cover them on a temporary basis or on overtime. This is the main task for the D.O. administration staff and is called **duty coverage**. The **duty coverage** is done on a weekly and a daily basis (Appendix A). In this subsection we summarise of the steps carried out to test the D.O. database against **duty coverage** requirements. Figure 5.1 shows the activities in terms of the D.O database that need to be carried out weekly for **duty coverage**.

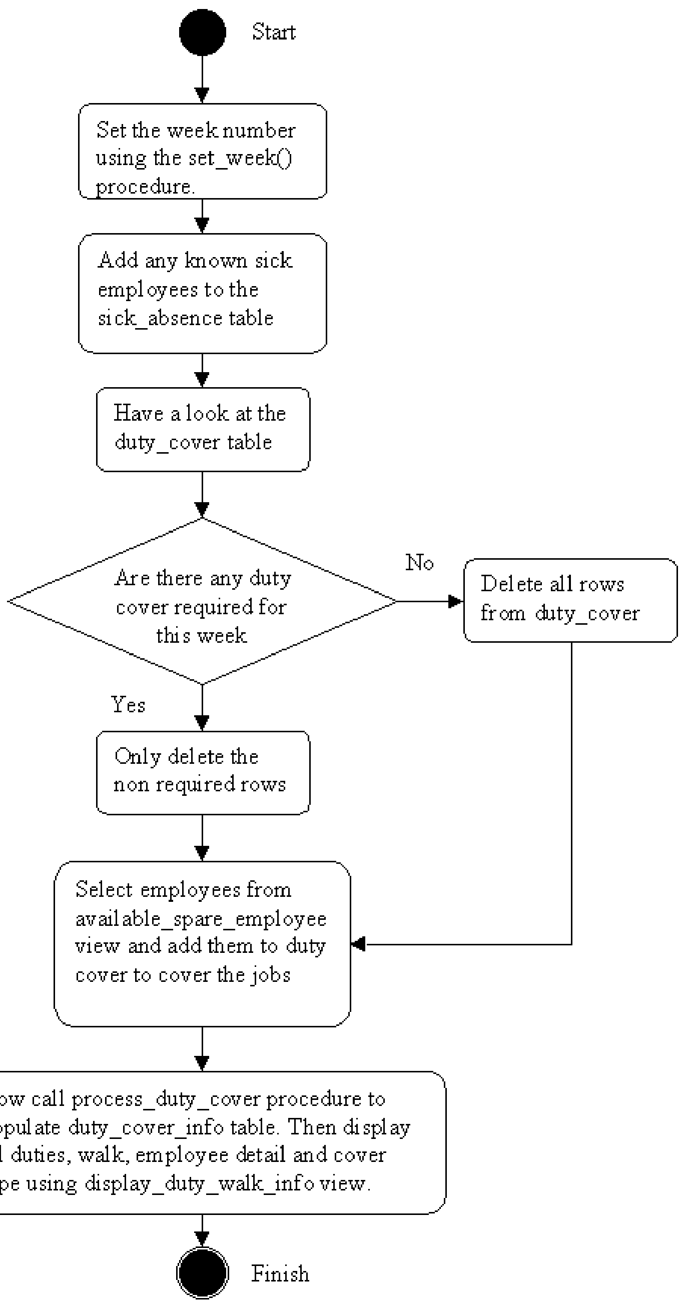

Start
Set the week number using the set_week() procedure.
Add any known sick employees to the sick_absence table
Have a look at the duty_cover table
Are there any duty cover required for this week
No
Delete all rows from duty_cover
Yes
Only delete the non required rows
Select employees from available_spare_employee view and add them to duty cover to cover the jobs
Now call process_duty_cover procedure to populate duty_cover_info table. Then display all duties, walk, employee detail and cover type using display_duty_walk_info view.
Finish

Figure 5.1 Activity Diagram for the weekly operation of the Delivery Office database

Weekly operation for duty coverage

The following in an example of the steps that need to be carried out weekly on the D.O database (Appendix 2.2):

1- Use the procedure set_week() to set the current week number of the financial year as follows:

```
CALL set_week(2)!
```

2- Add any known sick employees to the **sick_absence** table, then display it:

```
INSERT INTO sick_absence VALUES('25600','28/10/02','Stomach bug')!
SELECT * FROM sick_absence!
```

pay_no	date_commenced	reason
24661	10/09/02	Cold and flu
96742	10/10/02	Bad back
25600	02/10/28	Stomach bug

3- Display the **duty_cover** table used to assign temporary staff to duties:

SELECT * **FROM** duty_cover!

duty_no	cover	reason
S007	58470	VAC
S004	20903	VAC
N007	24001	VAC
E001	19119	AL

Have a look at the vacant and absent duties as follows:

SELECT * **FROM** vacant_absent_duties!

duty_no	reason	status
N002	AL	PT
S002	AL	PT
S005	SL	ID
N001	SL	PT
S004	VAC	FT
S007	VAC	FT
N007	VAC	ID
E007	VAC	PT

Duty 'E001' is not present in the table anymore, which means the duty holder has come back from annual leave (AL) this week. This can be checked by looking at the **booked** table and displaying the name and duty holder pay number of duty 'E001':

SELECT * **FROM** booked!

week_no	pay_no
2	55355
2	88974
3	54213
1	21652

SELECT name, pay_number, duty_number **FROM** employee_duty_walk **WHERE** duty_number = 'E001'!

name	pay_number	duty_number
S Wesley	21652	E001

From the two tables above it's clear that 'S Wesley' was on holiday last week (week 1). Now the tuple containing 'E001' is not required anymore in the **duty_cover** table. For the other vacant 'VAC' duties in the **duty_cover** table the cover can carryon to this week, so now the last row can be deleted as follows:

DELETE FROM duty_cover **WHERE** duty_no = 'E001'!

4- Now empty the **over_time** table:

```
DELETE FROM over_time!
```

5- Now call the procedure process_duty_cover() to populate **duty_cover_info** then display **duty_cover_info**:

```
CALL process_duty_cover()!
SELECT * FROM display_duty_walk_info!
```

name	office	pay_number	grade	duty_number	walk_type	delivery_method	status	reason	cover_type
	East			E007			PT	VAC	🟨
Y Jones	East	12458	OPG	E003	Town	Walk	FT		
P Smith	East	20034	ex_PHG	E005			ID		
S Wesley	East	21652	Opg	E001	Town	Walk	FT		
M Philip	East	78541	OPG	E002	Town	Walk	PT		
M Rushton	East	87961	OPG	E004	Bulk	Van	PT		
E Magee	East	99856	OPGDriver	E006			ID		
	North			N007			ID	VAC	duty cover
J McCanze	North	10023	OPG	N003	Bulk	Van	FT		
E Francis	North	10088	OPGDriver	N006			ID		
R Sims	North	24661	OPG	N001	Town	Walk	PT	SL	🟨
A Ali	North	63263	ex_PHG	N005			ID		
A Wilford	North	75632	OPG	N004	Rural	Cycle	FT		
S Mohammed	North	88974	OPG	N002	Town	Walk	PT	AL	🟨
	South			S004	Bulk	Van	FT	VAC	duty cover
	South			S007			FT	VAC	duty cover
M Thomas	South	23771	OPGDriver	S006			ID		
P James	South	55355	OPG	S002	Town	Walk	PT	AL	🟨
A Ahmed	South	77564	OPG	S003	Rural	Cycle	FT		
N George	South	77890	OPG	S001	Town	Walk	FT		
B Muhsin	South	96742	ex_PHG	S005			ID	SL	🟨
D McRobbert	West	11543	OPG	W003	Town	Walk	FT		
J Singh	West	19700	OPG	W002	Town	Walk	FT		
S Fox	West	31692	OPG	W004	Bulk	Van	FT		
J Wilcox	West	54213	OPG	W001	Town	Walk	PT		
M John	West	86961	OPGDriver	W006			ID		
O Steven	West	99471	ex_PHG	W005			ID		

6- By looking at columns **reason** and **cover_type** in the **duty_cover_info** table above we can see there are still 5 duties to be covered (highlighted with yellow). Let us see if there are any spare employees available:

```
SELECT * FROM available_spare_employees WHERE pay_number NOT IN (SELECT cover
FROM duty_cover)!
```

name	pay_number	grade	office	status
C Crane	19119	OPG	West	FT

The remaining duties to be covered can be checked as follows:

```
SELECT * FROM absent_duties_after_cover!
```

duty_number	reason	status

N002	AL	PT
S002	AL	PT
S005	SL	ID
N001	SL	PT
E007	VAC	PT

'C Crane' can be assigned to one of these duties as follows:

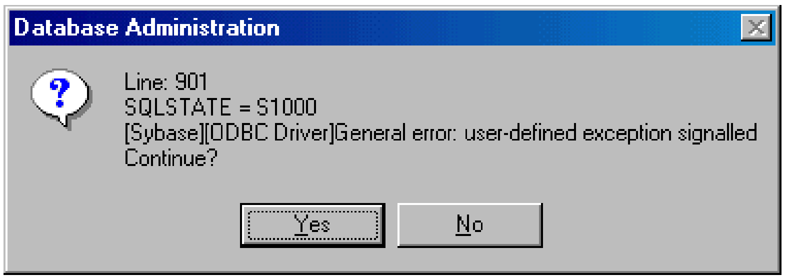
```
INSERT INTO duty_cover VALUES ('N002', '19119', 'AL')!
```

Daily operation for duty coverage

Now the database is ready for the daily operation that adds overtime to cover the remaining absent or vacant duties. The trigger defined on the over_time table will enforce that overtime should be used to cover a vacant or absent duty and the employee performing the overtime is not absent from work. Consider this example to validate the add_modify_over_time trigger:

```
CALL add_over_time('S002','55355',3)!
CALL add_over_time('E001','23771',3)!
```

When executing both statements failed to update the over_time table and InfoMaker displayed the dialog box shown in Figure 5.2 below. The first statement failed because employee with number '55355' exists in the **booked** table, and the second one failed because duty number 'E001' is not vacant or absent.


Database Administration

```
Line: 901
SQLSTATE = S1000
[Sybase][ODBC Driver]General error: user-defined exception signalled
Continue?
```

Figure 5.2 InfoMaker General error dialog box

The daily database operation is mainly entering any sick employees in the sick_absence table and adding overtime to the over_time table. Consider the following example to cover the five remaining duties mentioned in step 6 above:

```
CALL add_over_time('S002','23771',3)!
CALL add_over_time('S005','23771',5)!
CALL add_over_time('N001','10023',3)!
CALL add_over_time('E007','99856',3)!
```

To see the duty coverage summary we can call the procedure process_duty_cover() to populate **duty_cover_info** then display **duty_cover_info**:

```
CALL process_duty_cover()!
SELECT * FROM display_duty_walk_info!
```

name	office	pay_number	grade	duty_number	walk_type	delivery_method	status	reason	cover_type
	East			E007			PT	VAC	Overtime
Y Jones	East	12458	OPG	E003	Town	Walk	FT		
P Smith	East	20034	ex_PHG	E005			ID		
S Wesley	East	21652	Opg	E001	Town	Walk	FT		
M Philip	East	78541	OPG	E002	Town	Walk	PT		
M Rushton	East	87961	OPG	E004	Bulk	Van	PT		
E Magee	East	99856	OPGDriver	E006			ID		
	North			N007			ID	VAC	duty cover
J McCanze	North	10023	OPG	N003	Bulk	Van	FT		
E Francis	North	10088	OPGDriver	N006			ID		
R Sims	North	24661	OPG	N001	Town	Walk	PT	SL	Overtime
A Ali	North	63263	ex_PHG	N005			ID		
A Wilford	North	75632	OPG	N004	Rural	Cycle	FT		
S Mohammed	North	88974	OPG	N002	Town	Walk	PT	AL	duty cover
	South			S004	Bulk	Van	FT	VAC	duty cover
	South			S007			FT	VAC	duty cover
M Thomas	South	23771	OPGDriver	S006			ID		
P James	South	55355	OPG	S002	Town	Walk	PT	AL	Overtime
A Ahmed	South	77564	OPG	S003	Rural	Cycle	FT		
N George	South	77890	OPG	S001	Town	Walk	FT		
B Muhsin	South	96742	ex_PHG	S005			ID	SL	Overtime
D McRobbert	West	11543	OPG	W003	Town	Walk	FT		
J Singh	West	19700	OPG	W002	Town	Walk	FT		
S Fox	West	31692	OPG	W004	Bulk	Van	FT		
J Wilcox	West	54213	OPG	W001	Town	Walk	PT		
M John	West	86961	OPGDriver	W006			ID		
O Steven	West	99471	ex_PHG	W005			ID		

Figure 5.3 Summary of employees, duties, walks and coverage

4.1.2 Summary

We have seen in this section how the database was tested and validated to ensure the data is consistent with the client requirements. The testing was carried out in three stages. The first stage was to ensure the integrity of the database including both consistency and correctness. This stage went towards testing the constraints defined on the columns and tables.

Simple CHECK clauses to constraint the values of columns didn't need thorough testing because SQL automatically check the values of the columns when table are updated. However, constraints that involve complex CHECK clause and triggers needed very extensive testing to ensure that these integrity constraints represent the constraints in the conceptual data model.

Basically the test for this stage was by trying to violate these constraints and observe the outcome. When a constraint definition was successful, SQL displayed either an integrity constraint violation or general error dialog box. However when the constraint failed the database was updated and nothing was displayed. In those situations I revised the SQL syntax to correct the error.

The second stage was to ensure that the database satisfies the client's data requirements. The testing in this stage consisted mainly of general queries that made use of the base tables and views. The strategy followed in this test was to cover all the data requirements recorded in 'statement of the data requirements' (see appendix A), and some other queries the client is likely to use. For this purpose I've defined some SQL functions that can be used to enhance the data produced by general queries. For example, the **full_address()** function was used in different queries to produce tables with columns of full addresses.

The third stage was to ensure the database meets the client data processing requirements. The main client's data processing requirement is the **duty coverage** process. The main activities carried out in this process are shown in Figure 5.1. These activities were tested using the daily and weekly **duty coverage** scenarios outlined in subsection 5.1.2. These scenarios represented weekly and daily database operation that aims at producing the data shown in Figure 5.3.

Finally, the testing carried out in the above three stages ensured that the D.O. database meets its client requirements and maintains satisfactory internal consistency.

4.2 Review of the further topics

4.2.1 Overview of Query Processing

4.2.2 Implementation of Postal Addresses using SQL-99 User Defined Types

One of the extra subjects investigated in this project was the implementation of postal address using SQL99 user defined types. This topic looked at postal addresses as complex data that have internal parts of interest. The topic also discussed the need for some data type to represent postal addresses and assist in easily accessing their internal parts.

To better understand the nature and structure of postal addresses I had to carry out some literature search to learn more about postal addresses. The postal addresses structure outlined in Appendix F, was very useful when studying the internal structure of postal addresses and deciding the string length required to represent those internal parts. For example, here in the UK the postcode can be represented as SQL data type CHAR(7). I've also learnt that PAF is a relational database that has tables containing addresses information, and having relationships with each other maintained using foreign keys.

The topic summarised the levels of conformance for SQL standards and how SQL99 used a different approach of performance by using a core functionality that is a prerequisite for any sort of conformance. This is a superset of Entry SQL92, so implementations can go directly from Entry SQL92 to Core SQL99. Next, I learnt about the new relational and object oriented features in SQL99.

I've then considered the new data type **ROW** and how it can be used to implement columns or tables of postal addresses. And moved on to investigate the User Defined Types (UDT) in SQL99 and how types of postal addresses can be created, instantiated and extended using inheritance. UDTs follow the object-oriented concept in programming, by providing constructor to initialise object when they are created. Then I've considered the problem of violating the first normal form by implementing **ROW** types or UDTs as columns, and how it's an advantage to represent them as tables. This is logical, especially considering how both tables and classes can map to entities in the ER model.

Finally, I've seen how the change in direction of traditional relational DBMSs towards complex data required a change in their basic relational model. This change requires the extended the relational model by incorporating a variety of features that make it object relational. I've also learnt that this change presented some technical problems, one of which is query optimisation when there is an unknown number of UDTs instead of a narrow range of data types.

Chapter 5

Overview of Query Processing

5.1 Introduction to query processing:

The success of relational database technology in data processing is due, in part, to the availability of nonprocedural languages (i.e., SQL), which can significantly improve application development and end-user productivity. By hiding the low-level details about the physical organisation of the data, relational database languages allow the expression of complex queries in a concise and simple fashion. In particular, to construct the answer to the query, the user does not precisely specify the procedure to follow. This procedure is actually devised by a DBMS module (Figure 5.1), usually called a query processor. This relives the user from query optimisation, a time-consuming task that is best handled by the query processor, since it can exploit a large amount of useful information about the data (Valduriez and Ozsu.1999, pp. 188).

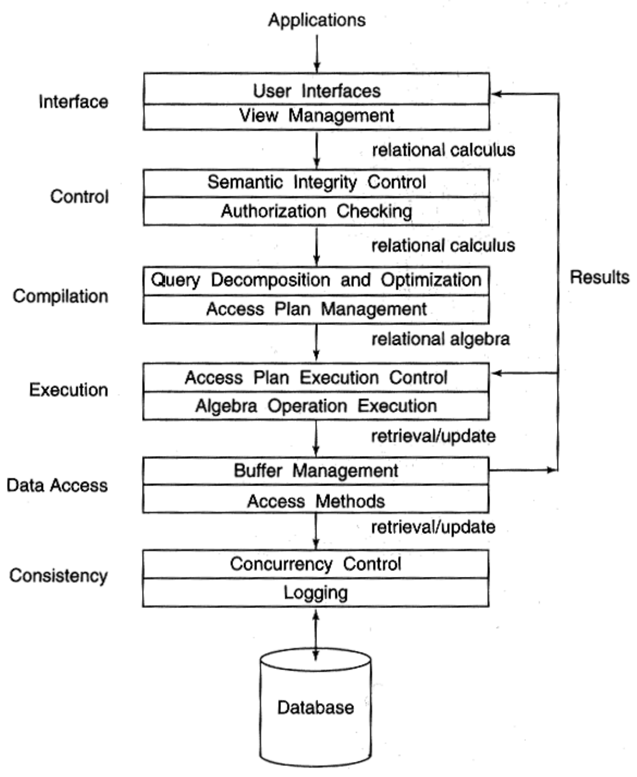


Figure 5.1 *Functional layers of a relational DBMS (Valduriez and Ozsu.1999, pp. 50).*

The objective of query processing layer (Figure 5.1) [..] is to transform a high-level query on a [centralised or] distributed database (seen as a single database by the users) into an efficient execution strategy expressed in a low-level language on the local databases. An important aspect of query processing is query optimisation (Valduriez and Ozsu.1999, pp. 192). Query optimisation is necessary for queries that are specified in a high-level query language such as SQL. This is because SQL queries are more declarative in nature. They specify what the intended results should be, rather than identifying the details of how the result should be obtained.

Because many execution strategies are correct transformations of the same high-level query, the one that optimises (minimise) resource consumption should be retained (Valduriez and Ozsu.1999, pp. 192). Some good measures of resource consumption are:

- *total cost* which is the sum of all times incurred in processing the operations of the query at various locations in a network,
- *response time*, which is the time elapsed for executing the query.

5.2 The Query Processor

The query processor receives an SQL query as input, translates and optimises this query in several phases into an executable query plan, and executes the plan in order to obtain the results of the query. If the query is an interactive ad hoc query (dynamic SQL), the plan is directly executed by the query execution engine and the results are presented to the user. If the query is a *canned* query that is part of an application program (embedded SQL), the plan is stored in the database and executed by the query execution engine every time the application program is executed (Chamberlin et al. 1981; cited in Kossmann, 2000).

5.3 Architecture and operation of a Query Processor

The general architecture for a query processor is shown in Figure 6.1 below. This architecture can be used for any kind of database system including centralized, distributed, or parallel systems (Kossmann. 2000). Components of the query processor briefly described below.

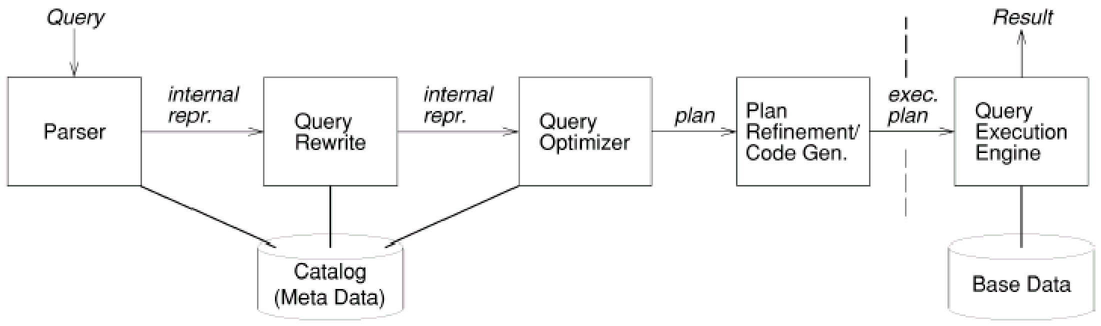


Figure 5.2 *Phases of query processing*
(Haas et al. 1989; cited in Kossmann. 2000).

5.3.1 The Parser

The parser checks the query syntax to determine whether it is formulated according to the syntax rules (rules of grammar) of the query language. The query must also be validated, by checking that all attribute and relation names are valid and semantically meaningful names in the schema of the particular database being queried (Elmasri and Navathe, 2001, pp. 585). All the information needed by the parser is stored in the catalog.

After the parsing and validating, an internal representation of the query is then created, usually as a tree data structure called a **query tree**. It is also possible to represent the query using a graph data structure called a **query graph**. For example, Figure 6.2 shows a query tree corresponding to the relational algebra expression for Query 1 on the D.O database below:

Query 1:

```
SELECT A.employee_name, B.license_no, C.reg_no, C.model
   FROM employee AS A, licensed_employee AS B, private_vehicle  AS C
   WHERE (A.pay_no = B.pay_no) AND (B.license_no = C.license_no)!
```

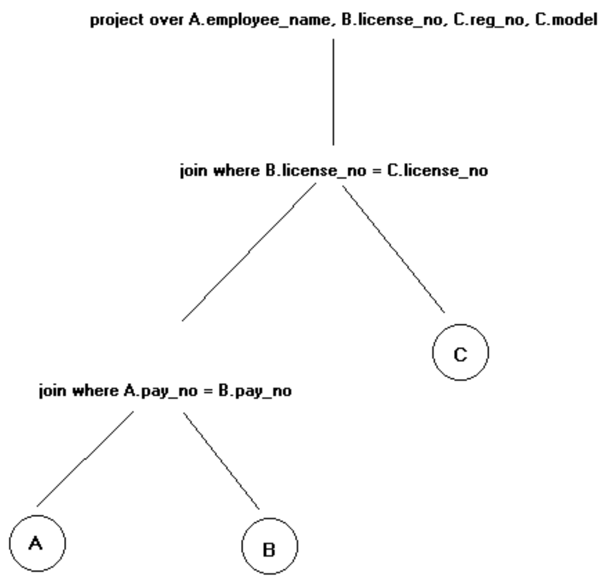

project over A.employee_name, B.license_no, C.reg_no, C.model
join where B.license_no = C.license_no
join where A.pay_no = B.pay_no
A
B
C

Figure 5.3 *Query tree corresponding to the relational algebra expression for Query 1*

The **query tree** is also called an **operator tree**. And is used to represent the relational algebra query graphically for the sake of clarity. The operator tree has leaf nodes which represent relationships stored in the database, and a non leaf nodes that is an intermediate relation produced by a relational algebra operator. The sequence of operations is directed from the leaves to the root, which represents the answer to the query.

5.3.2 The catalog

The catalog stores all the information needed in order to parse, rewrite and optimise a query. It maintains the s*chema* of the database (i.e., definition of tables, views, user-defined types and functions, integrity constraints, etc.), the *partitioning schema* (i.e., information about what global tables have been partitioned and how they can be reconstructed), and *physical information* such as the location of copies of partitions of tables, information about indices, and statistics that are used to estimate the cost of a plan. In most relational database systems, the catalog information is stored [..] [in tables called **system tables**] (Kossmann. 2000).

5.3.3 Query Rewrite

Then the **query rewrite** transforms a query in order to carry out optimisations that are good regardless of the physical state of the system. (e.g., the size of tables, presence of indices, locations of copies of tables, speed of machines, etc.). Typical transformations are the elimination of redundant predicates, simplification of expressions, and un-nesting of sub-queries and views (Kossmann. 2000). This component also rewrites the query in relational algebra. This is typically divided into the following two steps:

1- Straight-forward transformation of the query from relational calculus (SQL queries) into relational algebra.

2- Restructuring of the relational algebra query to improve performance.

The transformation of a tuple relational calculus query into an operator tree can easily be achieved as follows. First, a different leaf is created for each different tuple variable (corresponding to a relation). In SQL, the leaves are immediately available in the FROM clause. Second, the root node is created as a project operation involving the result attributes.

These are found in the SELECT clause in SQL. Third, the qualification (SQL WHERE clause) is translated into the appropriate sequence of relational operations (select, join, union, etc.) going from the leaves to the root [(see Figure 6.2 and Query 1)] (Valduriez and Ozsu.1999, pp. 210).

5.3.4 The query optimiser

The query optimiser finds the optimal execution strategy for a query. However, in some case the chosen execution plan is not the optimal (best) strategy, it is just a reasonably efficient strategy for executing the query. Finding the optimal strategy is usually too time-consuming except for the simplest of queries and may require information on how the files are implemented and even on the contents of the files. This information may not be fully available in the DBMS catalog.

The query optimiser is usually seen as three components: a search space, a cost model, and a search strategy (Figure 6.3). The *cost model* predicts the cost of a given execution plan. [..] (Valduriez and Ozsu.1999, pp. 232).

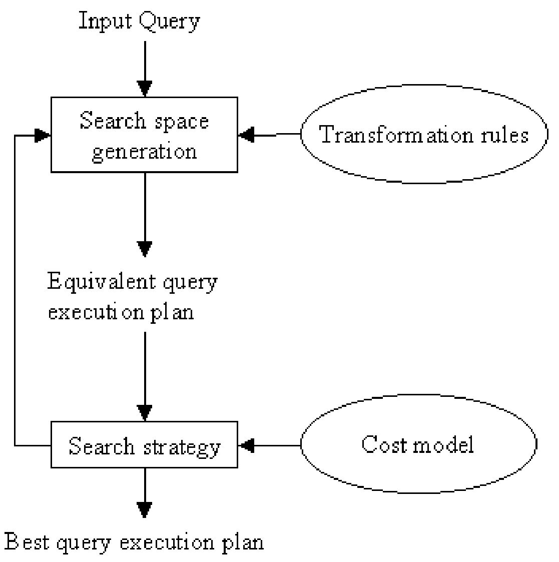


Figure 5.4 *Query Optimization Process (Valduriez and Ozsu.1999, pp. 230)*

5.3.5 Search Space

The search space is the set of alternative execution plans to represent the input query. These plans are equivalent, in the sense that they yield the same result but they differ on the execution order of operations and the way these operations are implemented, and therefore on performance. The *search space* is obtained by applying transformation rules, such as those for relational algebra. Because query execution plans are typically abstracted by means of operator trees, for a given query, the search space can thus be defined as the set of equivalent operator trees.

5.3.6 Search Strategy

The *search strategy* explores the search space and selects the best plan, using the cost model. The search strategies can widely be divided into two classes:

- **Exhaustive search**: where potentially the whole space is examined, and
- **Heuristic search**: where some 'heuristics' or knowledge acquired through experience is used to restrict the search to a smaller space.

The most popular search strategy used by most of the commercial database products in their query optimiser is the dynamic programming, which is almost exhaustive. The advantage of this strategy is that it assures the 'best' of all plans is found. It incurs an acceptable optimisation cost (in term of time and space) when the number of relations in the query is small (Valduriez and Ozsu.1999, pp. 232). However, the disadvantage of this strategy is that it has exponential time and space complexity (Kossmann. 2000). The time and space required takes the general form of A^n (where n is the number of relations in the query), which means that the time and space can grow very large in the case of complex queries. In particular, in a distributed system, the complexity of this strategy is prohibitive for many queries (Kossmann. 2000).

Alternative strategies that follow the heuristic search approach are the randomised strategies. They avoid the high cost of optimisation, in terms of memory and time consumption, but do not guarantee the best of all plans (Valduriez and Ozsu.1999, pp. 232).

5.3.7 The Plan

A plan specifies precisely how the query is to be executed. Plans are represented as trees. The nodes of a plan are operators, and every operator carries out one particular operation (subsection 6.1.6). The nodes of a plan are annotated, indicating, for instance, where the operator is to be carried out in case of distributed query processing. Figure 5.5 below shows the execution plan produced by InfoMaker when executing Query 1 above.

```
Estimate 4 I/O operations (best of 6 plans considered)
Scan private_vehicle AS C sequentially
Estimate getting here 9 times
Scan licensed_employee AS B using primary key
for rows where license_no equals private_vehicle.license_no
Estimate getting here 9 times
Scan employee AS A using primary key
for rows where pay_no equals licensed_employee.pay_no
Estimate getting here 9 times

Subquery1:
Estimate 1 I/O operations
Scan company_vehicle using primary key
for rows where reg_no equals expr
Estimate getting here 1 times
```

Figure 5.5 *SQL statements execution plan*

5.3.8 Plan Refinement/Code Generation.

This component transforms the plan produced by the optimizer into an *executable plan*. In some systems, plan refinement also involves carrying out simple optimizations, which are not carried out by the query optimizer in order to simplify its implementation.

5.3.9 Query Execution Engine.

This component has the task of running the query code, whether in complied or interpreted mode, to produce the query result. This component also generates an error message if a runtime error results.

5.4 Summary

In this section the architecture of the query processor (Figure 5.1) was briefly described. This architecture can be used for both centralised and distributed query processing. We also seen how query traverse through the various component of Figure 5.1. First, the query in SQL is checked and validated by the parser using the information stored in the Catalog (system tables) After that an internal representation is generated in a form of query tree that corresponds to the relational calculus of the query.

This query tree is then transformed into an operator tree by the Query Rewrite component. This operator tree has leaf nodes which represent relationships stored in the database, and a non leaf nodes that is an intermediate relation produced by a relational algebra operator. This operator tree is then used by the Query Optimiser to generate a search space. The search space can be defined as the set of equivalent operator trees that generates the same input query.

The search space can be examined for the best execution plan in terms of the cost model. This examination can be done using either exhaustive or heuristic search strategies. We also considered an example of exhaustive search using the dynamic programming strategy. Which, has an advantage of finding the best solution, and one of its disadvantage is that time and space increase exponentially with the query complexity. Randomised strategies were considered as an alternative to dynamic programming. Randomised strategies avoid the high cost of optimisation, but do not guarantee the best of all plans.

Finally we've seen how plan generated by the query optimiser is refined and used to generate the execution code that is used by the Query Execution Engine.

Chapter 7

Implementation of Postal Addresses using SQL-99 User Defined Types

7.2.1 Introduction

Extending the ability of the DBMS and its query processor to do more things will extend the ability to meet the demands placed upon it. This reduces the amount of the data that needs to be transferred from the DMBS to the user process. As a result, each client or user process needs to do less work to meet its requirement (Figure 5.1).

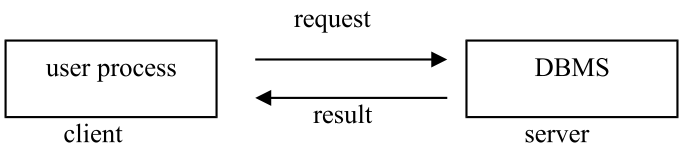


Figure 7.1 *A simplified view of client-server processing*

Take as an example, a company that maintains a mailing list that holds the account numbers, names and addresses of its customers. These data are held in a customer table, which contains thousands of tuples as follows:

Table 7.1: Customer Table

Account Number	Customer Name	Address
001234	J Smith	12 Warwick row, COVENTRY, CV5 3RR
901159	P McDonald	56 Cowley Road, DONCASTER, DN7 2SD
562098	A Singh	90 Cross Avenue, LONDON, N7 2WW
⋮		

The address column could merely be defined as a character type:

```
CREATE DOMAIN postal_addresses VARCHAR(100);
```

and then create the table as follows

```
CREATE TABLE Customer
     (account_number INTEGER,
      customer_name VARCHAR(12),
      address postal_addresses,
      PRIMARY KEY account_number);
```

Now imagine that the company wants to send some sale catalogues to all customers living in Doncaster. Transferring the whole table to the user process, which then searches the data for all the customers living in Doncaster, can do this. However this is quite inefficient in case that the customer table is large, and that the user process resides on a computer other than the one containing the table by sending the whole table across the network.

Another option would be to use the DBMS build-in functions or user-defined functions that resides in the database alongside the data, or use external functions written in other programming languages and can be called by the DBMS. In all three cases, a function may use one or more parameters of a predefined data type but it can return only one value of a given data type. This raises the demand for some structured data types that can be manipulated as an object. Objects that can be stored in variables, passed as arguments to routines and returned as return values from function invocations and more than that can provide access to its internal parts. These data types can be used to represent complex data such as postal addresses that have subcomponents of interest (street, town, post code, …etc.).

In this section we briefly discuss the new features provided in SQL3 that supports User Defined Data types (UDTs), and how these UDTs can be used to implement complex data such as postal addresses.

7.2.2 Introduction to postal addresses

Postal addresses are an example of complex data because they have internal attributes such as streets, towns, counties and postcodes. These internal attributes have structure of interest and values that need to be stored, retrieved and processed [Appendix F].

Nowadays it's a legal requirement for everyone who wants to open a bank account, buy a car, or shop online to have a postal address. So postal addresses are stored in many database systems, such as mailing lists, credit records, bank accounts, insurance records, …etc. to mention a few. And to complicate the issue further a postal address might have a correct format and at the same time be invalid. This is because a complete postal address represents a property or a post box, which is a physical location that can be located and visited by the postal staff.

Consider the following addresses,

50 Hampton Road Foleshill COVENTRY CV6 5GE WEST MIDLANDS UK	1 Paradise Way Cowley OXFORD OX4 1DL OXFORDSHIRE UK
(a) A valid postal address	**(b) An invalid postal address**

Figure 6.2 An example of a valid and invalid postal addresses

While the format of both addresses is correct, address (b) is invalid because it does not represent a real address. From the discussion above it's quite clear that there a need to validate postal addresses before storing them to ensure the integrity of the database. Checking these postal addresses against a standard database provided by the postal system carries out this validation. Here in the UK, Royal Mail provides what is called the PAF (Post code Address File) which contains over 26 million UK addresses. These addresses are stored in a relational format and sold by Royal Mail. In the United States a similar database called AIS (Address Information System) is provided by the US postal service.

7.2.3 SQL Standards and levels of conformance

SQL standard is defined jointly by the *ISO* (International Organisation for standardisation) and *ANSI* (the American National Standards Institute). [..]. Here is the list of the major ANSI / ISO SQL standards:

- **SQL86:** This provided a minimum functionality that all existing products had in common. Hence, it basically standardised syntax that was somewhat at variance.

- **SQL89:** This made only one major change to SQL86, which was to add support for mechanisms to enforce foreign key relationships.

- **SQL92:** This was a major update of the standard. Instead of simply certifying and standardising the overlap among existing products, the ISO specified future growth for the language, including much functionality that had not yet been implemented by anyone, at least not in the way specified. Since conforming to this standard is rather ambitious, the ISO specified three distinct levels of conformance: Entry, Intermediate, and Full [Gruber, 2000, pp. 21-22]. At the time of writing most commercial database vendors are still at Entry level, with some features from higher levels thrown in. For example SQLAnywhere conforms to SQL92 Entry level standards, but have features from higher levels such as cursors.

- **SQL99:** This is the new standard [..], it further extends SQL92 to include integration with object-oriented approaches, programmatic extensions, and other features.

SQL92 was structured into three levels of conformance so that implementers could conform to it gradually. As it turns out, most implementers have got stuck at Entry level because they are primarily interested in focusing on proprietary features. Hence, SQL99 takes a different approach. It uses a Core functionality that is a prerequisite for any sort of conformance. This is a superset of Entry SQL92, so implementations can go directly form Entry SQL92 to Core SQL99. Beyond that functionality are several types of enhanced conformance. Implementations that support Core can also support any, all, or none of the enhanced levels. Core conformance also requires that the DBMS support either Embedded SQL or directly coded modules [Gruber, 2000, pp. 474].

7.2.4 SQL99 and the support for complex data

SQL99 new features can be crudely partitioned into: relational features and object-oriented features.

Relational features are features that relate to SQL's traditional role and data model. These features are often divided into about five groups:

- New data types
- New predicates
- Enhanced semantics
- Additional security
- Active database (provided through *triggers*)

***Object oriented features*:** Some of the features that fall into this category were first defined in the SQL/PSM (Persistent Stored Modules) standard published in late 1996-specifically, support for functions and procedures invocable form SQL statements. SQL99 enhances that capability, called SQL-invoked routines, by adding a third class of routine know as *methods* [Eisenberg and Melton, 1999, pp. 134].

SQL/PSM, refers to procedural extensions to SQL that make it computationally complete and therefore suitable for the development of full applications. More commonly, these extensions will be used to create more sophisticated standard operations [such as functions and procedures] that can be invoked by any application that can access the database [Gruber, 2000, pp.475]. Some proprietary products already support this, such as SQLAnywhere DBMS.

Other object oriented features include the support for UDTs (user-defined datatypes). Users can now define their own data types by sub-classing the given types or by creating new structured types that behave like objects in the object oriented world. Along with defining the type, the user can specify functions that can be applied to the type or methods that are included in it. These functions can be used in SQL statements and the values they return referenced in predicates [Gruber, 2000, pp.479].

7.2.5 Implementation of postal addresses

We are going to consider some of SQL99 features that can be used to implement complex data, taking postal addresses as an example. This can be done in two ways; first by using the new build-in data type **ROW**, or by declaring a Structured User Defined Type (UDT).

A new data type supported in SQL99 is the Row data type, a composite type that contains one or more fields. The complete rows can be stored in variables, passed as arguments to routines and returned as return values from function invocations. It's possible to create tables whose tuples are of a particular row type, this gives database designers the additional power of storing structured values in single columns of the database. A row type may be defined using the syntax:

```
CREATE ROW TYPE row_type_name (<component declarations>);
```

In the case of postal addresses a row type can be created in two ways:

1- Create a row type with a name assigned to it as follows:

```
CREATE ROW TYPE postal_address_t
    (building_number CHAR(4),
     street CHAR(80),
     locality CHAR(35),
     post_town CHAR(30)
     post_code CHAR(7)
    );
```

Then a table of postal addresses can then be declared based on the row type declaration with the building number and the postcode as a primary key as follows:

```
CREATE TABLE postal_address OF postal_address_t
    (PRIMARY KEY (building_number, post_code));
```

We can then use SQL INSERT statement to populate the table as follows:

```
INSERT INTO postal_address
VALUES('89', 'Nuffield Road', 'Foleshill', 'Coventry',
       'CV6 5DD');
```

2- Create a table with some columns containing row values. This can be used to create a table that holds both customers' details such as first name, surname, date of birth, account numbers and their addresses we can use the following CREATE TABLE statement:

```
CREATE TABLE customers
    (account_no INTEGER,
     full_name ROW (first_name CHAR(20),
                    surname CHAR(20)),
     date_of_birth DATE,
     postal_address ROW (building_number CHAR(4),
                         street CHAR(80),
                         locality CHAR(35),
                         post_town CHAR(30)
                         post_code CHAR(7)),

     PRIMARY KEY account_no
    );
```

To populate the table we can use the SQL INSERT statement as follows:

```
INSERT INTO customers
VALUES(10024, ('John', 'Smith'), 19/08/76, ('89',
       'Nuffield Road', 'Foleshill', 'Coventry', 'CV6
       5DD'));
```

The question now is how can we reference components of tuples in order to use them in queries. [To answer this], SQL 3 uses a double dot notation to build path expressions that refer to the components of tuples [Elmasri and Navathe, (2001),pp. 454]. For example the, the query below retrieves the account numbers of customers living in Sheffield from the **customers** table.

```
SELECT customers.account_no
FORM customers
WHERE customers.postal_address..post_town = 'Sheffield';
```

In SQL3 a construct similar to class definition is provided whereby the user can create a named user-defined type with its own behavioural specification and internal structure; it is know as an **Abstract Data Type (ADT)**. The general form of an ADT specification is:

```
CREATE TYPE <type-name> (
     list of component attributes with individual types
     declaration of EQUAL and LESS THAN functions
     declaration of other functions (methods));
```

An ADT is a database object and therefore resides in a schema like a table or a view. To create an ADT we can use the above CREATE TYPE statement; to destroy one we can use DROP TYPE. Here is an example of CREATE TYPE to implement a postal address:

```
CREATE TYPE general_postal_address_t AS
    (building_number CHAR(4),
     street CHAR(80),
     post_town CHAR(30),
     NOT INSTANTIABLE,
     NOT FINAL
    );
```

Since it has several attributes, this is a structured UDT. The reason the UDT is declared as NOT FINAL is that we intend to inherit from this type and make it a superclass; the reason we did not make it instantiable is that we intend to use it as an *abstract*[6] superclass. Now we can create a post code UK address or a foreign address by inheriting from the abstract class `general_postal_address_t` as follows:

- A post code UK address

```
CREATE TYPE uk_postcode_address_t UNDER
general_postal_address_t  AS
    (locality CHAR(35),
     post_code CHAR(7),
     INSTANTIABLE,
     FINAL
    );
```

- United States Address:

```
CREATE TYPE us_zip_address_t UNDER
general_postal_address_t  AS
    (state CHAR(2),
     zip_code CHAR(5),
     INSTANTIABLE,
     FINAL
    );
```

Any complex types of postal addresses for example BFPO addresses or other foreign addresses can be created using inheritance.

Now let us consider the support SQL3 gives towards instantiating and processing these UDTs. To instantiate (create instances) a class, we would use a constructor method that is automatically created by SQL3. Here is an example:

[6] **abstract class** is a class that is not intended to have instances and in which some or all of its *methods* may be undefined. Abstract classes are used to define common structure for its subclasses

```
DECLARE postal_address uk_postcode_address_t;
SET postal_address = uk_postcode_address_t();
```

This statement makes use of the partial procedural features set that is part of the Basic Object Support (BOS) [in SQL3], specifically the use of variables, which, under BOS, can be of UDT datatypes. The DECLARE statement creates a variable to hold our new instance. The constructor for the UDT is uk_postcode_address_t(). Therefore the SET statement creates a new uk_postal_address_t object and stores it in the stud1 variable [Gruber, 2000, pp.490].

UDTs can function either as column values or as entire tables. In the former case, they violate the first normal form [..]. In any case, mapping UDTs to tables is a bit more logical, especially considering how both tables and classes can map to entities in the ER model [Gruber, 2000, pp.490].

The syntax to CREATE a typed table is rather simple:

```
CREATE TABLE post_addresses OF
             uk_postcode_address_t;
```

The question now is how to access the attributes of a UDT. To answer this SQL3 automatically creates what is called mutator and observer methods. Consider the following example:

```
SET postal_address =
          uk_postcode_address_t.building_no('2a');
SET postal_address =
          uk_postcode_address_t.street('Severn Way');
SET postal_address =
          uk_postcode_address_t.locatlity ('Grimsby');
SET postal_address =
          uk_postcode_address_t.post_code('DN1 3KL');
SET postal_address =
          uk_postcode_address_t.post_town('DONCASTER');
```

The mutators such as uk_postcode_address_t.post_code() and uk_postcode_address_t.post_town() are used to the values of the attributes of the object stored in post_address. The build-in mutator methods are named after the attributes they affect. To put the information held in post_address into the table post_addresses, we would use an INSERT statement, like this:

```
INSERT INTO post_addresses
     VALUES(post_address.building_no,
        post_address.street, post_address.post_town,
        post_address.locality,
        post_address.post_code
        );
```

To retrieve values from attributes we use the observer method. An observer method automatically takes the name of the attribute on which it works. The DBMS tells the difference between a mutator and an observer by looking at how the method is used. Consider the following example to retrieve the post towns from the post_addresses table:

```
SELECT post_address.post_town
       FROM  post_addresses;
```

7.2.5 Summary

In the above sections we have seen how the new features in SQL3 can be used to implement complex data such as postal addresses. We have also seen examples of postal address UDTs functioning either as column values or as entire tables. In the former case, they violate the first normal form because tables having columns defined as UDT may be multi-valued such as in postal addresses where a street and a town depends on the postcode and the building number not on the primary key for the table. However, some observers recognise this as just another "decomposable" data type [Eisenberg and Melton ,1999]. In any case, mapping UDTs to tables is a bit more logical, especially considering how both tables and classes can map to entities in the ER model

Finally, we have seen how the new features in SQL3 moved towards supporting complex data by adopting object-oriented approaches such as UDTs. On the other hand traditional RDBMSs (Relational Database Management Systems) based on SQL2 standards support complex queries and simple data (Figure 5.3). This is true because although SQL2 supports date and time types, this can be simple data compared with to the complex UDTs that can be defined under SQL3.
To move towards complex data, complex query (Figure 5.3), basic relational model of the RDBMSs need to be extended by incorporating a variety of features that make it object relational. The new DBMSs that belong to this category are called ORDBMSs (Object-Relational Database Management Systems). This change in direction presents a number of technical problems. For example query optimization changes when there may be an unknown number of UDTs instead of the narrow range of numbers, characters and date.

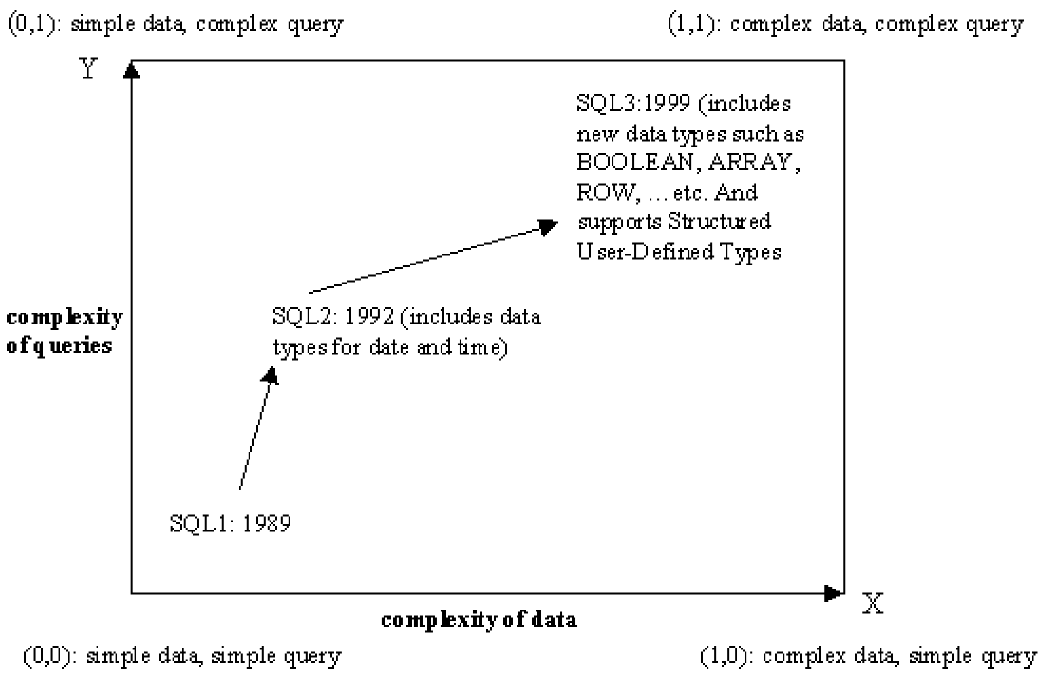

(0,1): simple data, complex query
(1,1): complex data, complex query
Y
SQL3:1999 (includes new data types such as BOOLEAN, ARRAY, ROW, … etc. And supports Structured User-Defined Types
complexity of queries
SQL2: 1992 (includes data types for date and time)
SQL1: 1989
X
complexity of data
(0,0): simple data, simple query
(1,0): complex data, simple query

Figure 6.3 A change of direction for SQL to handle complex data

Chapter 8

Conclusion

8.1 Review of the project activities

This project followed the general activities used for software development shown in Figure 7.1 below. The five activities of establishing requirements, analysis, design, implementation and testing are the 'backbone' of the software development process. Project management and quality management are the two additional activities that hold the process of development together.

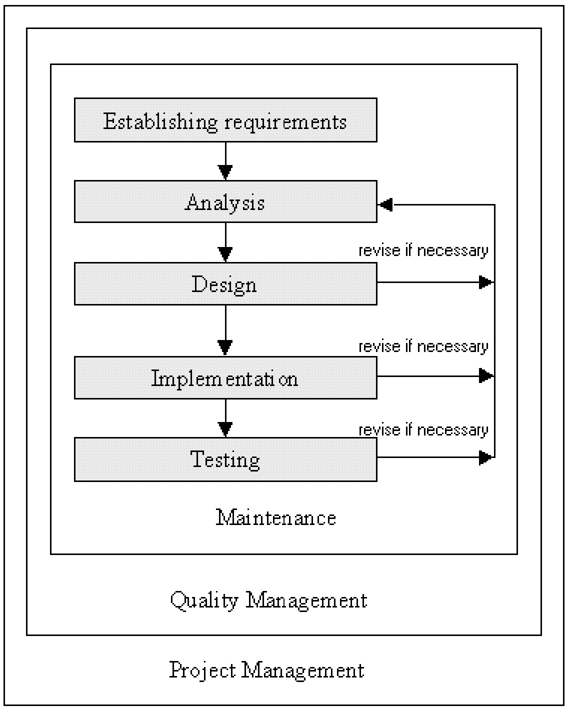


Figure 8.1 General iteration model for software development

The work on the project started with developing a project proposal, and based on this proposal I've used some of the project management techniques to develop a plan for the work on the project. This plan divided the work into phases and each phase included a number of tasks, and to monitor progress key events such as TMAs, and milestones such as the production of conceptual data model were identified. Based on these phases, tasks and key events a project schedule was constructed using a Gantt chart (Figure 4.1).

The database development side of project phases were based on activities outlined in **Figure 4.2**; these phases were establishing requirements, data analysis, database design, implementation and testing. The process followed for the design is shown in **Figure 4.6**, this process involved revising the outcome of each activity and use any changes to modify the conceptual data model and propagate the change down the development process.

The database development activities started with establishing the client requirements and the outcome of it was the statement of data requirement (Appendix A); this statement summarised the client data and data processing requirements. The statement was also checked with the client to ensure nothing was left out. Using the statement of data requirements I started the data analysis activity which resulted in the production of the conceptual data model

(Appendix B). One of the major decisions I took was the use entity subtypes to understand the data better.

Using the initial conceptual data model I've developed the relational data model which is the specification of the logical schema (Appendix C). The activities I undertook were to represent entities, relationships and constraints. Relations resulted from entities, however some of the relations resulted from resolving m:n relationships. Relations used to represent entity subtypes were based on the sub entities and not the super ones. Foreign keys were used to represent relationships, however in the case of relationships between entities and super-types it was not possible to use a foreign key definition. This is because a foreign key can only represent one relation. In these case the referential integrity was maintained using a general constraint.

The implementation stage required transforming the relations into SQL tables using CREATE TABLE statements. During the implementation stage some decisions had to be made due to the implementation of SQL Anywhere; for example, the UNIQUE definition in SQL Anywhere require a NOT NULL too. In contrast the relational model specified some of the alternate keys as not null. This affected two issues, the first design issue is on which table the alternate key used to represent 1:1 relationships need to be placed. The second implementation issue was the use of triggers and CHECK clause to implement alternate keys.

Triggers were heavily used in the implementation activity to represent dynamic and sophisticated constraints. Functions and procedures were used to implement the client's data processing requirements. InfoMaker 5.0 was used as a direct entry tool to execute the D.O. SQL DDL and DML (Appendix D).

The final stage was testing the database to ensure it both satisfies the client's requirements and maintain an internal consistent state. The database integrity was tested invalid and inconsistent data to ensure all triggers and table constraints containing queries functions as required. On the other hand the client's data requirements were tested by running some SELECT queries (subsection 4.2.1). Some of these queries were used to select data from base tables and views (Appendix E), others were used to select data that was either explicitly expressed in the requirements or likely to be executed by the client. In either case the tests ensured the database completeness.

The client's data processing requirements were tested by using daily and weekly scenarios of the **duty coverage** process. The weekly activity diagram for the D.O. database is shown in **Figure 5.1**. The activity diagram was used to write a set of queries that involved the use procedures and functions. The result of these queries was the summary provided in **Figure 5.3**.

The work carried out in this project demonstrated the use of database development techniques to provide a solution to a real problem. Not only that but also gave an example of how ambiguity can arise during the process of data analysis and how these can be resolved by further discussing them with the client. The project also shown that during database development some decisions might need to be made based on many factors. These decisions can be assessed against any alternatives, and choosing the approach that provide more advantages in terms of efficiency, usability, flexibility, and so on.

The use of entity subtypes as a powerful way of representing and understanding complex situations was explored in this project. The project proved that in some situations the implementation of entity subtypes involves quite complex constraints which required the use of triggers. Although the complexity involved in implementing entity subtypes, it might be an advantage over not using them and ending with a situation where data can't be understood.

The project also gave an example of modelling and implementing postal addresses based on the **postal_address** table in the D.O. database. This demonstrated an approach in which postal addresses can be stored as a table that can be referenced from other tables using **building_no** and **post_code** as foreign key. This approach can be used in practice with DBMSs that does not support UDTs.

This approach was preferred over other techniques such as representing postal address as a string or including the address columns in other tables because it has two advantages. First, it provides a way where the address sub components can be accessed directly in queries without the need for complex functions. Second, this approach avoids the transitive dependency introduces when including all the **postal_address** columns in other tables that have primary keys other than **building_no** and **post_code**.

The project looked briefly at the new features in SQL99 and how these can be used to implement postal addresses. Features such as **ROW** data type and **UDT**s were briefly described and backed with practical examples. This work represents a valuable foundation that can be used to change the way entity subtypes are implemented. By providing **UDT**s that acts as super classes, subclasses can inherit structure and behaviour and this way entity subtypes can be directly implemented as subclasses of **UDT**s. These **UDT**s can then be used as a table in the database. For example, the **GeneralPostalAddress** entity and its subtypes in the E-R diagram shown in **Figure 7.2**, can be implemented using the types general_postal_address_t, uk_postcode_address_t, us_zip_address_t described in subsection 6.2.5.

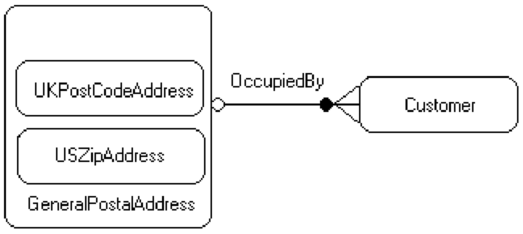


Figure 8.2 Cusomer and GeneralPostalAddress E-R diagram

8.2 Suggestions for Further Work

8.2.1 Access control for the D.O database

The capability of the D.O database can be extended if time allows, to include access control for the Delivery Office database. This will involve the identification of all database users and their access rights (privileges). Because the only creator of tables in the database is the DBA, he or she can grant privileges. This allows specified users to perform certain actions on the database. The DBA can also add new users to the database by using SQL Anywhere GRANT

CONNECT statement. For example, the following SQL statement can be used to add a new user to the D.O database, with user ID north_admin and password welcome:

```
GRANT CONNECT TO north_admin IDENTIFIED BY welcome;
```

Now the DBA can grant privileges on tables, views or procedures using the following SQL statement:

```
GRANT <privileges> ON <table/view/procedure> TO <user>;
```

<privileges> includes the following:

EXECUTE (the only privileges for procedures), ALTER, DELETE, INSERT, REFERENCES, SELECT, UPDATE and ALL

Privileges can be granted on views and procedures instead of base tables for extra and tailored security, this may be to limit access to portions of a table in the case of views or to strict the manner in which a table can be modified in case of procedures.

For the D.O database extra views can be defined to select only the data relevant to one of the delivery offices. The DBA can then grant access on these manager and administration staff of that delivery office. For example, the following SQL statements creates a view that selects only some the information for the North delivery office employees from the employee table and grants access on it to the administration staff of that office:

```
CREATE VIEW north_employees
        ("name", pay_number, grade, phone, "address", skills, overtime_availability, status) AS
        SELECT employee_name, pay_no, grade, phone, full_address(house_no, post_code), skills,
                overtime_availability, status
         FROM employee WHERE office = 'North';
```

```
GRANT SELECT ON north_employees TO north_admin;
```

8.2.2 An application process for the D.O database

For the D.O database to be used by users who doesn't know SQL an **application process** needs to be developed. This application process can provide a Graphical User Interface (GUI) to make accessing and using the database easy. Not only that but the application process can also provide added security and control. An application process can be developed using any of the commercial **Integrated Development Environment** (IDE) such as Borland Delphi or Borland C++ Builder.

Both IDEs offer database visual components library that can be used to develop an **application process** to access the databases. These database components have default behavior that enables them to perform useful functions with little or no programming.

The IDE Component palette provides two types of database component that can be used to connect to the database, retrieve information, update the database or execute stored procedures:

- The *Data Access components* contain objects that simplify database access by encapsulating database source information, such as the database to connect to, the tables in that database to access, and specific field references within those tables.

- The *Data Control components* contain data-aware user interface components for displaying database information in forms. Data Control components are like standard user interface components, except that their contents can be derived from or passed to database tables.

References & Bibliography

References:

[1] Valduriez, P. and Ozsu, M. (1999) *Principles of Distributed Database Systems,* 2nd Edition, New Jersey, Prentice Hall.

[2] Gruber M. (2000) *Mastering SQL*, Sybex International.

[3] Elmasri, R. and Navathe, S. (2001) *Fundamentals of Database Systems*, 3rd Edition, Addison-Wesley.

[4] Eisenberg, A. and Melton J. (1999) 'SQL: 1999, Formerly Known as SQL3' ACM SIGMOD Int. Conf. on Management of Data **28**, 1, pp. 131–138.

[5] Kossmann, D. (2000) 'The State of the Art in Distributed Query Processing' ACM Computing Surverys **32**, 4, pp. 422–469.

[6] Royal Mail (2000) 'The Complete PAF Digest Issue 5'
Available online at:
http://www.royalmail.com/docContent/other/Downloadable_Files/PAF_Digest_Issue_5_0.pdf

[7] Sybase, Inc. (1996) Sybase SQL Anywhere User's Guide Ver. 5.0, Copyright © 1991-1996 by Sybase, Inc. and its subsidiaries. All rights reserved. Printed in the United States of America.

[8] OU (1998) *'Relational Datbases'* M358 course text, The Open University, Walton Hall

[9] Haas, L., Freytag, J. C., Lohman, G., And Pirahesh, H. 1989. 'Extensible query processing in starburst'. ACM Sigmod Int. Conf. on Management of Data (Portland, OR, USA, May), pp. 377–388.

[10] Chamberlin, D., Astrahan, M., King, W., Lorie, R., Mehl, J., Price, T., Schkolnik, M., Selinger, P., Slutz, D., Wade, B., And Yost, R. 1981. 'Support for repetitive transactions and ad hoc queries in System R' *ACM Transactions on Database Systems* ***6***, 1 (March), pp. 70–94.

Bibliography

[1] Skills in Accessing, Finding, and Reviewing Information (SAFARI) course online, the Open university
http://sorbus.open.ac.uk/safari_ou/contents.htm

[2] Field, M. and Keller, L. (1998) *Project Management* , Thomson Learning

[3] Manola, F. and Sutherland, J. (2002) 'SQL3 Object Model' available online at:
http://www.objs.com/x3h7/sql3.htm
Accessed November 2002

Appendices

APPENDIX A The Delivery Office Statement of Data Requirements

In the management of the day-to-day activities the delivery office administration staff require a database to meet the following:

1- Coventry has four delivery offices each one of them is identified by its name and has a telephone number, a manager and a postal address.

2- Each delivery office has a number of walks, a walk is a collection of addresses, which are on the same road or are close to each other, each walk has a unique walk number, a type (town, rural, or bulks), a delivery method (van or on foot), notes (dog warnings, etc…) and a postal area that it covers. The delivery office also has a number of vans and each van is characterised by a unique number beside the DVLA registration number, a make, a model and the size. Besides the MOT due date is also required. The delivery office is staffed by a number of employees responsible for sorting and delivering the mail. Every employee is uniquely identified using a pay number. The employee name, date of birth, date of entry to the business, grade, phone number, address, badge number, skills, permanent duty and overtime availability need also be recorded. Each employee covers a duty or a drop and can perform overtime to cover a duty. Full time & par-time duties usually has a unique number, first post. Full time duties have second part which can be performing second post or performing indoors mail sorting (ASAP), they also have scheduled day off (in a 6 days week with a rotating day off) and the name of the employee covering the day off. This rotation is governed by six weeks rotation table 1 below. A duty covers one walk, or other indoors activities. An Employee might have a driving licence and owns a private car in which case the driving licence number need to be recorded and the vehicle details too (Registration number, make, model, colour). A driver employee does drops; every walk have at least one drop (drops bags). If a duty is not covered by employees (vacant) or if the employee is absent (holiday or sick) it's covered by overtime. Employees may also feel sick and not attend to work, in which case the pay number of the absent employee, the date commenced and the reason of sickness need to be recorded. This record need only be kept while the employee is off sick, past sick absences are stored separately. A conduct record is kept for each employee, which might be counselling notes, a reprimand, or attendance procedure warning, in either case some notes need to be kept. Customer can complaint about the service being carried out by employees, each complaint has a unique reference number and is from a specific address, notes of the complaints need to be kept too. Each employee can book annual leave by selecting weeks from the current year. Each week is identified uniquely by a week number and is limited by a quota.
Additional data requirement will include, but are not limited to:
- The ability to identify each day which duties are covered and which aren't
- Keeping record of overtime and provide a list of the walks covered by overtime.

- A list of the office annual leave record.
- A list of all the sick employees, and the ones on holiday.
- A list of the conduct record and complaints again staff members.

APPENDIX B The Delivery Office Conceptual Data Model

B.1 Entity types

Employee (PayNo, Name, DateOfBirth, DateOfEntry, Grade, Phone, BadgeNo, Skills, OvertimeAvailability, Status)

DeliveryOffice (Name, TelNo, ManagerName)

Vehicle (RegNumber, Model)

PrivateVehicle (Colour)

CompanyVehicle (VehicleNo, MotDueDate, Size)

Conduct (Reference, Type, Notes, Date)

Week (WeekNo, Quota)

SickAbsence (PayNo, DateCommenced, Reason)

Walk (WalkNumber,Office, Type, DelMethod, Remarks, Status)

MailDrop (WalkNo, Office, DropPoint, Details)

Duty (DutyNo)

WalkDuty()

FullTimeDuty ()

PartTimeDuty (Hours)

NonWalkDuty(DutyDetails)

DutySecondPart (DutyNo, WeekNo, DayOff, SecondPart)

DutyCover (DutyNo, PayNo, Cause)

Overtime (DutyNo, EmployeeNo, Duration)

Complaint (ReferenceNo, Details, Date)

LicensedEmployee (LicenceNo, VehicleSize)

Address (PostCode, BuildingNumber, StreetName, Town, PostalArea)

B.2 Constraints

The total annual leave for a certain week must not exceed the quota for that week.

Only employees with a driver grade or postal higher grade can participate in the PermanatlyCovers relationship with NonWalkDuty.

Only full time employee can participate in PermanentlyCoveredBy with FullTimeDuty and only part time employees can participate in PermanentlyCoveredBy with PartTimeDuty.

An Employee participating in CoversDayOff must not be participating in PermanentlyCoveredBy

A duty can be covered by overtime or duty cover only if it does not participate into PermanatlyCovers and HasCover or the employee doing that duty is participating in the Has or MayBe relationships.

An employee can participate in PermanentlyCoveredBy with only FullTimeDuty or PartTimeDuty or NonWalkDuty.

Employee and duties participating in PermanentlyCoveredBy should belong to the same delivery office

An employee participating in Performs or ScheduledFor should not be absent form work.

An employee can only participate in the Performs relationship if the attribute OvertimeAvailability is ‘Yes’.

B.3 Assumptions

Only current sick absences are recorded, no history is kept.

A company vehicle might not be a spare vehicle and not used for a specific Delivery Office.

A duty might be vacant, i.e. with no employee covering it.

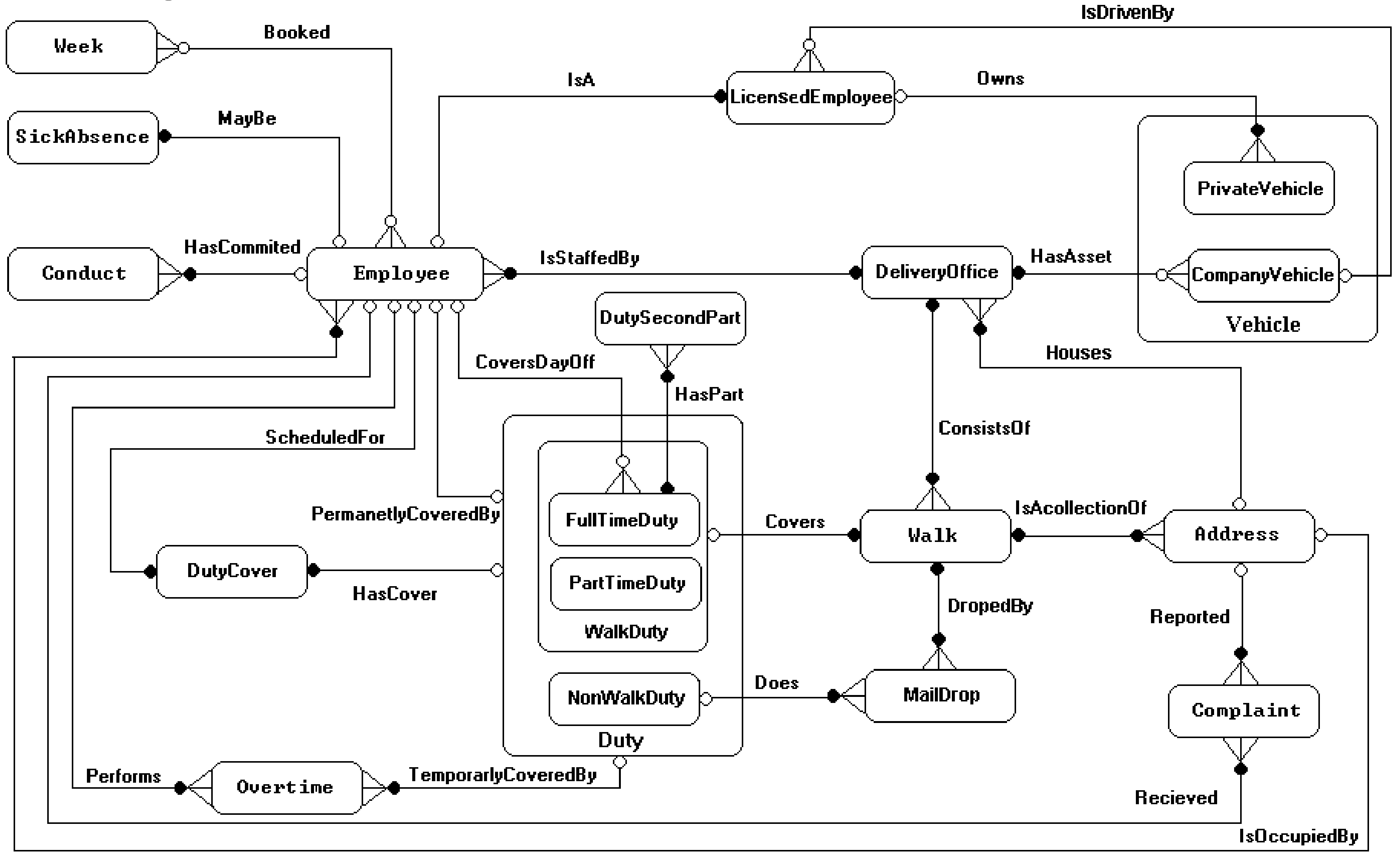
An employee can be a spare, i.e. not covering any duty.

The overtime is updated daily, i.e. no history is kept.

The duty cover are updated weekly, no history is kept.

All full time duties are 40 hours.

B.4 E-R Diagram



APPENDIX C The Delivery Office Relational Model

model *DeliveryOffice*

domains
AbsentDutyCauses = (AL, SL, VAC)
BadgeNumbers = string(4)
BuildingNumers = string
ComplaintReferenceNumbers = String
ConductReferenceNumbers = string
ConductType = (Councelling, Reprimand, SeriousReprimand)
Dates = Calenderdates
Days = Weekdays
DaysOff = Weekdays
DeliveryOfficeNames = (South, North, East, West)
DelMethod = (Van, Cycle, Onfoot)
DropPoints = string
DutyDetails = string
DutyHours = hours
DutyNumbers = string(4)
DutyWeeks = 1..6
EmployeeGrades = (OPG, OPGDriver, ex_PHG)
EmployeeNumbers = 10000 .. 99999
EmployeeSkills = string
EmployeeStatus = (FT, PT)
GeneralNotes = string
Holidays = 1..5
LicenceNumbers = string
OTAvailability = (yes, no)
PersonNames = string
PostalAreas = string
PostCodes = string(8)
SecondPartOptions = (SecondPost, ASAP)
StreetNames = string
TelephoneNumbers = string
TownNames = string
VehicleColour: string
VehicleModel: string
VehicleNumbers = string
VehicleRegsNumbers = string
VehicleSizes = (150, 50)

WalkNumbers = 1..100
WalkType = (Town, Rural, Bulk)
WeekNumbers = 1 .. 53
WeekQutas = 0 .. 100

relation *Employee*
PayNo: EmployeePayNumbers
Name: PersonNames
DatesOfBirth: Dates
DateOfEntry: Dates
Grade: EmployeeGrades
Phone: TelephoneNumbers
BadgeNo: BadgeNumbers
Skills: EmployeeSkills
OvertimeAvailability: OTAvailability
Office: DeliveryOfficeNames
PostCode: PostCodes
HouseNo: BuildingNumbers
Status: EmployeeStatus
primary key *PayNo*
{IsStaffedBy}
foreign key *Office* **references** *DeliveryOffice* **not allowed null**
alternate key *BadgeNo* **not allowed null**
{IsOccupiedBy}
foreign key *(PostCode, HouseNo)* **references** *Address* **not allowed null**
{no constraints regarding the relationship with relation Duty to reflect spare employees}

relation *DeliveryOffice*

Name: DeliveryOfficeNames

TelNo: TelephoneNumbers
ManagerName: PersonNames
BuildingNumber: BuildingNumbers
PostCode: PostCodes
primary key *Name*
{Houses}
{ represents mandatory participation with respect to HasAsset }
foreign key *(BuildingNumber, PostCode)* **references** *Address* **not allowed null**
constraint *(***project** *DeliveryOffice* **over** *Name)* **difference** *(***project** *CompanyVehicle* **over** *Office)* **is empty**
{ represents mandatory participation with respect to ConsistsOf }
constraint *(***project** *DeliveryOffice* **over** *Name)* **difference** *(***project** *Walk* **over** *Office)* **is empty**
{ represents mandatory participation with respect to IsStaffedBy }
constraint *(***project** *DeliveryOffice* **over** *Name)* **difference** *(***project** *Employee* **over** *Office)* **is empty**

relation *Walk*

WalkNumber: WalkNumbers

Office: DeliveryOfficeNames

Type: WalkType

DeliveryMethod: DelMethod
Remarks: GeneralNotes

Status : EmployeeStatus

DutyNo: DutyNumbers

primary key *(WalkNumber, Office)*
{ConsistsOf}
foreign key *Office* **references** *DeliveryOffice*
{ part of primary key i.e. not allowed null, represent mandatory participation with respect to ConsistsOf }
alternate key *DutyNo* **not allowed null**
{ represents mandatory participation with respect to IsAcollectionOf from Walk side}
constraint *(***project** *Walk* **over** *WalkNumber, Office)* **difference** *(***project** *Address* **over** *WalkNumbers, Office)* **is empty**
{ represent mandatory participation with respect to DropedBy from Walk side }
constraint *(***project** *Walk* **over** *WalkNumber, Office)* **difference** *(***project** *MailDrop* **over** *WalkNumbers, Office)* **is empty**
{represents the mandatory participation condition with regards to the 1 : 1 relationship Covers, and also here the referential integrity is maintained using a constraint}

{Covers}

constraint *(***project** *Walk* **over** *DutyNo)* **difference** *((***project** *FullTimeDuty* **over** *Duty)* **union** *(***project** *PartTimeDuty* **over** *WalkNo))* **is empty**

relation *MailDrop*

WalkNo: WalkNumbers

Office: DeliveryOfficeNames

DropPoint: DropPoints

Details: GeneralNotes
DutoNo: DutyNumbers
primary key *(WalkNo, Office, DropPoint)*
{DropedBy}
foreign key *(WalkNo, Office)* **references** *Walk* **not allowed null**
{Does}
foreign key *DutyNo* **references** *NonWalkDuty* **not allowed null**

relation PrivateVehicle

RegNumber: VehicleRegNumbers

Model: VehicleModel

Color: VehicleColour
LicenceNo: LicenceNumbers
primary key *RegNumber*
{Owns}
foreign key *LicenceNo* **references** *LicencedEmployee* **not allowed null**

relation *CompanyVehicle*

RegNumber: VehicleRegNumbers

VehicleNumber: VehicleNumbers
MotDueDate: Dates
Size: VehicleSizes

Model: VehicleModel

Office: DeliveryOfficeNames
primary key *RegNumber*
alternate key *VehicleNumber* **not allowed null**
{HasAsset}
foreign key *Office* **references** *DeliveryOffice* **allowed null**

relation *Conduct*

Reference: ConductReferenceNumbers

Type: ConductType
Notes: GeneralNotes
StaffNumber: EmployeeNumbers
Date: Dates
primary key *Reference*
{HasCommited}
foreign key *StaffNumber* **references** *Employee* **not allowed null**

{relationship Booked}

relation *Booked*

WeekNo: WeekNumbers

PayNo: EmployeeNumbers
primary key *(WeekNo, PayNo)*
{Forms}
foreign key *WeekNo* **references** *Week*
{Has}

foreign key *PayNo* **references** *Employee*
{being part of the primary key reflects the mandatory participation condition with regards to Has & Forms}

relation *Week*

WeekNo: WeekNumbers

Quota: WeekQuotas
primary key *WeekNo*

relation *SickAbsence*
PayNo: EmployeeNumbers
DateCommenced: Dates
Reason: GeneralNotes
primary key *(PayNo)*
{MayBe}
{ being part of the primary key reflects the mandatory participation condition with regards to MayBe }
foreign key *PayNo* **references** *Employee*
{DateCommenced not allowed to be null}
constraint select *SickAbsence* **where** *(DateCommenced* **is null***)* **is empty**

relation *FullTimeDuty*
DutyNo: DutyNumbers

DutyHolder: EmployeeNumbers

DayOffCover: EmployeePayNumbers
primary key *DutyNo*
alternate key *DutyHolder* **not null**
{PermanatlyCovers}
foreign key *DutyHolder* **references** *Employee*
{CoversDayOff}
foreign key *DayOffCover* **references** *Employee*

relation *PartTimeDuty*
DutyNo: DutyNumbers

DutyHolder: EmployeeNumbers

Hours: DutyHours
primary key *DutyNo*
alternate key *DutyHolder* **not null**
{PermanatlyCovers}
foreign key *DutyHolder* **references** *Employee*

relation *NonWalkDuty*
DutyNo: DutyNumbers

DutyHolder: EmployeeNumbers

Details: DutyDetails
primary key *DutyNo*
alternate key *DutyHolder* **not null**
{PermanatlyCovers}
foreign key *DutyHolder* **references** *Employee*

relation *DutySecondPart*
DutyNo: DutyNumbers
WeekNo: DutyWeeks
DayOff: Days
SecondPart: SecondPartOptions
primary key *(DutyNo, WeekNo)*
{HasPart}
foreign key *DutyNo* **references** *FullTimeDuty* **not allowed null**

relation *DutyCover*
DutyNo: DutyNumbers
Cover: EmployeeNumbers
Cause: AbsentDutyCauses
primary key *DutyNo*
{The following alternate key declarations represent the 1:1 relationship between Duty, Employee and DutyCover, not null represents the mandatory participation conditions in regards to ScheduledFor and HasCover}

alternate key *Cover* **not allowed null**
{ScheduledFor}
foreign key *Cover* **references** *Employee*
{because of the entity subtypes, referential integrity between DutyCover and Duty can only be maintained by a general constraint}
{HasCover}
constraint *(***project** *DutyCover* **over** *DutyNo)* **difference** *((***project** *FullTimeDuty* **over** *DutyNo)* **union** *(***project** *PartTimeDuty* **over** *DutyNo))* **is empty**

relation *Overtime*
DutyNo: DutyNumbers
EmployeeNo: EmployeeNumbers
Duration: DutyHours
primary key *(WalkNo, EmployeeNo)*
{Performs}
foreign key *EmployeeNo* **references** *Employee*
{because of the entity subtypes, referential integrity between Overtime and Duty can only be maintained by a general constraint}
{TemporarlyCoveredBy}

constraint *(***project** *Overtime* **over** *DutyNo)* **difference** *((***project** *FullTimeDuty* **over** *DutyNo)* **union** *(***project** *PartTimeDuty* **over** *DutyNo)* **union** *(***project** *NonWalkDutyDuty* **over** *DutyNo))* **is empty**

relation *Complaint*
Reference: ComplaintsReferenceNumbers
Details: GeneralNotes
PostCode: PostCodes
BuildingNumber: BuildingNumbers
PayNo: EmployeeNumbers
Date: Dates
primary key *ReferenceNo*
{Reported}
foreign key *(PostCode,BuildingNumber)***references** *Address* **not allowed null**
{Recieved}
foreign key *PayNo* **references** *Employee* **not allowed null**

relation *LicensedEmployee*
LicenseNo: LicenseNumbers
VehicleSize: VehicleSizes
EmployeeNo: EmployeeNumbers
RegNumber: VehicleRegNumbers
primary key *LicenseNo*
alternate key *EmployeeNo* **not allowed null**
{IsA}
foreign key *EmployeeNo* **references** *Employee* **not allowed null**
{IsDrivenBy}
foreign key *RegNumber* **references** *CompanyVehicle* **allowed null**

relation *Address*
PostCode: PostCodes
BuildingNumber: BuildingNumbers
StreetName: StreetNames
PostalArea: PostalAreas
Town: TownNames
WalkNo: WalkNumbers
Office: DeliveryOfficeNames
primary key *(PostCode, BuildingNumber)*
{IsAcollectionOf}
foreign key *(WalkNo, Office)* **references** *Walk* **not allowed null**

APPENDIX D The Delivery Office SQL Listings

D.1 SQL Data Definitions Statements

D.1.1 Domains definitions

```
CREATE DOMAIN delivery_office_names AS VARCHAR(10)
        CHECK (@col IN ('south', 'north', 'east', 'west'))!

CREATE DOMAIN post_codes AS VARCHAR(8)
  NOT NULL!

CREATE DOMAIN building_numbers AS VARCHAR(4)
  NOT NULL!

CREATE DOMAIN employee_numbers AS CHAR(5)!

CREATE DOMAIN work_status AS CHAR(2)
        NOT NULL
        CHECK(@col IN ('pt', 'ft'))!

CREATE DOMAIN phone_numbers AS CHAR(11)!

CREATE DOMAIN person_names AS VARCHAR(20) NOT NULL!

CREATE DOMAIN walk_numbers AS SMALLINT
  CHECK (@col BETWEEN 1 AND 100)!

CREATE DOMAIN walk_types AS VARCHAR(5)
        NOT NULL
        CHECK (@col IN ('Town','Rural','Bulk'))!

CREATE DOMAIN delivery_methods AS VARCHAR(5)
        NOT NULL
        CHECK (@col IN ('Van','Cycle','Walk'))!

CREATE DOMAIN vehicle_reg_numbers AS VARCHAR(8)!

CREATE DOMAIN duty_numbers AS CHAR(4)
        CHECK (LEFT (@col,1) IN ('s','n','w','e'))!

CREATE DOMAIN license_numbers AS CHAR(16)

CREATE DOMAIN vehicle_numbers AS VARCHAR(4)!

CREATE DOMAIN vehicle_sizes AS SMALLINT
  CHECK (@col IN (50, 150, 250))!
```

```
CREATE DOMAIN days AS CHAR(3)
  CHECK (@col IN ('MON','TUE','WED','THU', 'FRI','SAT'))!
```

D.1.2 Table definitions

```
CREATE TABLE postal_address
        (post_code post_codes,
         building_no building_numbers,
         street_name VARCHAR(20) NOT NULL,
         postal_area VARCHAR(20),
         town VARCHAR(14) NOT NULL,
         walk_no walk_numbers NOT NULL,
         office delivery_office_names NOT NULL,
         PRIMARY KEY (post_code, building_no))!

CREATE TABLE delivery_office
        (office_name delivery_office_names,
         telephone_no phone_numbers,
         manager_name person_names,
         building_no building_numbers,
         post_code post_codes,
         PRIMARY KEY (office_name),
         // Houses
         FOREIGN KEY (post_code, building_no) REFERENCES postal_address)!

CREATE TABLE employee
        (pay_no employee_numbers,
        employee_name person_names,
        date_of_birth DATE NOT NULL,
        date_of_entry DATE NOT NULL,
        grade VARCHAR(10) CHECK (grade IN ('opg','opgdriver','ex_phg')) NOT NULL DEFAULT
        'opg',
        phone phone_numbers,
        badge_no VARCHAR(4),
        skills LONG VARCHAR,
        overtime_availability VARCHAR(3) CHECK (overtime_availability IN ('yes', 'no')),
        office delivery_office_names NOT NULL,
        house_no building_numbers,
        post_code post_codes,
        status work_status,
        PRIMARY KEY (pay_no),
        UNIQUE (badge_no),
        // IsStaffedBy
        FOREIGN KEY (office) REFERENCES delivery_office,
        // defined as not null in the domain definition which reflects
        // madatory participation condition with regards to postal_address & delivery_office
        // IsOccupiedBy
        FOREIGN KEY (post_code, house_no) REFERENCES postal_address )!
```

```
CREATE TABLE walk
        (walk_number walk_numbers,
         office_name delivery_office_names,
         walk_type walk_types,
         delivery_method delivery_methods,
         remarks LONG VARCHAR,
         status work_status,
         duty_no duty_numbers,
         // ConsistsOf
         PRIMARY KEY (walk_number, office_name),
         /* duty number as alternate key guarantee 1: from walk side and the mandatory prticipation
         condition with regards to Covers */
         UNIQUE (duty_no),
         FOREIGN KEY (office_name) REFERENCES delivery_office,
         // represent mandatory participation with respect to IsAcollectionOf from Walk side
         CHECK
                (EXISTS (SELECT * FROM postal_address
                              WHERE (postal_address.walk_no = walk.walk_number)
                                    AND (postal_address.office = walk.office_name))))!

// IsAcollectionOf
ALTER TABLE postal_address
        ADD FOREIGN KEY (walk_no, office) REFERENCES walk!

CREATE TABLE company_vehicle
        (reg_no vehicle_reg_numbers,
         vehicle_no vehicle_numbers,
         mot_due_date DATE,
         vehicle_size vehicle_sizes DEFAULT NULL,
         model VARCHAR(15),
         office_name delivery_office_names DEFAULT NULL,
         PRIMARY KEY (reg_no),
         UNIQUE (vehicle_no),
         //HasAsset
         FOREIGN KEY (office_name) REFERENCES delivery_office)!

CREATE TABLE licensed_employee
        (license_no license_numbers,
         vehicle_size vehicle_sizes DEFAULT NULL,
         pay_no employee_numbers NOT NULL,
         vehicle_no vehicle_reg_numbers DEFAULT NULL,
         PRIMARY KEY (license_no),
         UNIQUE (pay_no),
         // IsA
         FOREIGN KEY (pay_no) REFERENCES employee,
         //IsDrivenBy
         FOREIGN KEY (vehicle_no) REFERENCES company_vehicle)!

CREATE TABLE private_vehicle
        (reg_no vehicle_reg_numbers,
         model VARCHAR(15),
         color VARCHAR(10),
         license_no license_numbers,
         PRIMARY KEY (reg_no),
         // Owns
         FOREIGN KEY (license_no) REFERENCES licensed_employee)!
```

```
CREATE TABLE complaint
        (reference_no CHAR(7),
         post_code post_codes,
         building_no building_numbers,
         pay_no employee_numbers NOT NULL,
         details LONG VARCHAR,
         complaint_date DATE,
         PRIMARY KEY (reference_no),
         // Reported
         FOREIGN KEY (post_code, building_no) REFERENCES postal_address,
         // Recieved
         FOREIGN KEY (pay_no) REFERENCES employee)!

CREATE TABLE conduct
        (reference_no CHAR(5),
         conduct_type VARCHAR(16)
                CHECK (conduct_type IN ('Councelling','Reprimand','SeriousReprimand')) NOT
                NULL,
         notes LONG VARCHAR,
         staff_no employee_numbers NOT NULL,
         conduct_date DATE,
         PRIMARY KEY (reference_no),
         // HasCommited
         FOREIGN KEY (staff_no) REFERENCES employee)!

CREATE TABLE week
        (week_no SMALLINT
                CHECK (week_no BETWEEN 1 AND 53),
         quota SMALLINT NOT NULL DEFAULT 0,
         PRIMARY KEY (week_no))!

CREATE TABLE booked
        (week_no SMALLINT
                CHECK (week_no BETWEEN 1 AND 53),
         pay_no employee_numbers,
         PRIMARY KEY (week_no, pay_no),
         // Forms
         FOREIGN KEY (week_no) REFERENCES week,
         // Has
         FOREIGN KEY (pay_no) REFERENCES employee)!

CREATE TABLE sick_absence
        (pay_no employee_numbers,
         date_commenced DATE NOT NULL,
         reason LONG VARCHAR,
         PRIMARY KEY (pay_no),
         // MayBe
         FOREIGN KEY pay_no REFERENCES employee)!

/*----------------------------------------*/
/* Duty relations implementation */
/*----------------------------------------*/
CREATE TABLE non_walk_duty
        (duty_no duty_numbers,
         pay_no employee_numbers DEFAULT NULL,
         duty_details VARCHAR(70),
         PRIMARY KEY (duty_no),
         // PermanatlyaCovers
         FOREIGN KEY (pay_no) REFERENCES employee)!
```

```
CREATE TABLE full_time_duty
        (duty_no duty_numbers,
         duty_holder employee_numbers DEFAULT NULL,
         day_off_cover employee_numbers DEFAULT NULL,
         PRIMARY KEY (duty_no),
         // PermanatlyCovers
         FOREIGN KEY (duty_holder) REFERENCES employee,
         // CoversDayOff
         FOREIGN KEY (day_off_cover) REFERENCES employee)!

CREATE TABLE part_time_duty
        (duty_no duty_numbers,
         duty_holder employee_numbers DEFAULT NULL,
         duty_hours SMALLINT,
         PRIMARY KEY (duty_no),
         // PermanatlyCovers
         FOREIGN KEY (duty_holder) REFERENCES employee)!

//// Duty Cover table to be accessed through a procedure only
CREATE TABLE duty_cover
        (duty_no duty_numbers,
         cover employee_numbers,
         reason VARCHAR(3) CHECK (reason IN ('SL','AL','VAC')),
         PRIMARY KEY (duty_no),
         UNIQUE (cover),
         // ScheduledFor
         FOREIGN KEY (cover) REFERENCES employee)!

CREATE TABLE over_time
        (duty_no duty_numbers,
         pay_no employee_numbers,
         duration SMALLINT,
         PRIMARY KEY (duty_no, pay_no),
         // Performs
         FOREIGN KEY (pay_no) REFERENCES employee)!

CREATE TABLE duty_second_part
        (duty_no duty_numbers,
         rotation_week SMALLINT CHECK (rotation_week BETWEEN 1 AND 6),
         day_off days,
         second_part VARCHAR(11) CHECK (second_part IN ('Second post','ASAP')),
         PRIMARY KEY (duty_no, rotation_week),
         // HasPart
         FOREIGN KEY (duty_no) REFERENCES full_time_duty)!

CREATE TABLE walk_drop
        (walk_no walk_numbers,
         office delivery_office_names,
         drop_point VARCHAR(70),
         license_no license_numbers,
         drop_details VARCHAR(70),
         PRIMARY KEY (walk_no, office, license_no),
         // Does
         FOREIGN KEY (license_no) REFERENCES licensed_employee,
         // For
         FOREIGN KEY (walk_no, office) REFERENCES walk)!
```

```
/*  Implementation tables  */
CREATE TABLE current_week
          (week_no SMALLINT
                    CHECK (week_no BETWEEN 1 AND 53),
           PRIMARY KEY (week_no))!


CREATE TABLE duty_cover_info
          (duty_no duty_numbers,
           reason VARCHAR(3) CHECK (reason IN ('SL','AL','VAC')),
           cover_type VARCHAR(10),
           PRIMARY KEY (duty_no))!
```

D.1.3 Table modification statement

```
// represent mandatory participation with respect to IsStaffedBy
ALTER TABLE delivery_office
          ADD CHECK (office_name IN (SELECT office FROM employee))!

// represent mandatory participation with respect to ConsistsOf
ALTER TABLE delivery_office
          ADD CHECK (office_name IN (SELECT office_name FROM walk))!

// represent mandatory participation with respect to HasAsset
ALTER TABLE delivery_office
          ADD CHECK (office_name IN (SELECT office_name FROM company_vehicle))!

// maintains referential integrity with regards to HasCover
ALTER TABLE duty_cover
          ADD CHECK (
                              (duty_no IN (SELECT duty_no FROM part_time_duty))
                              OR
                              (duty_no IN (SELECT duty_no FROM full_time_duty))
                              OR
                              (duty_no IN (SELECT duty_no FROM non_walk_duty)))!


// maintains referential integrity with regards to TemporarilyCoveredBy
ALTER TABLE over_time
          ADD CHECK (
                              (duty_no IN (SELECT duty_no FROM part_time_duty))
                              OR
                              (duty_no IN (SELECT duty_no FROM full_time_duty))
                              OR
                              (duty_no IN (SELECT duty_no FROM non_walk_duty)))!


// ensures that a licenced employee can only drive the vehicle with vehicle_size from
// licenced_employee
ALTER TABLE licensed_employee
          ADD CHECK
                    (vehicle_size = (SELECT vehicle_size FROM company_vehicle WHERE reg_no =
                                                                  licensed_employee.vehicle_no))!
```

D.1.4 Views definitions

```
CREATE VIEW employee_car
        ( employee_name, license_no, reg_no, model) AS
         SELECT employee.employee_name, licensed_employee.license_no,
           private_vehicle.reg_no, private_vehicle.model
         FROM employee, licensed_employee, private_vehicle
         WHERE (employee.pay_no = licensed_employee.pay_no) AND
                        (licensed_employee.license_no = private_vehicle.license_no)!


CREATE VIEW sick_employees
        ("name", office, pay_number, grade, phone, "address") AS
         SELECT employee_name, office, pay_no, grade, phone, full_address(house_no, post_code)
         FROM employee
         WHERE pay_no IN (SELECT sick_absence.pay_no FROM sick_absence WHERE
                                    sick_absence.pay_no = employee.pay_no)!


CREATE VIEW all_duties
        (duty_no, duty_holder, status) AS
         SELECT duty_no, duty_holder, 'FT' FROM full_time_duty
         UNION
         SELECT duty_no, duty_holder , 'PT' FROM part_time_duty
         UNION
         SELECT duty_no, pay_no, 'ID' FROM non_walk_duty!


CREATE VIEW employee_duty_walk
        ("name", office, pay_number, grade, duty_number, walk_type, delivery_method, status) AS
         SELECT employee_name, office, pay_no, grade, all_duties.duty_no, walk_type,
                delivery_method, walk.status
         FROM employee, all_duties, walk
         WHERE (employee.pay_no = all_duties.duty_holder) AND (walk.duty_no =
                all_duties.duty_no)
         UNION
         SELECT employee_name, office, employee.pay_no, grade, all_duties.duty_no, '', '',
         all_duties.status
         FROM employee, all_duties
         WHERE (employee.pay_no = all_duties.duty_holder) AND (all_duties.status = 'ID')
         UNION
         SELECT '', duty_to_office (all_duties.duty_no), '', '', all_duties.duty_no, walk_type,
         delivery_method, all_duties.status
         FROM all_duties LEFT OUTER JOIN walk ON all_duties.duty_no = walk.duty_no
         WHERE (all_duties.duty_holder IS NULL)!


CREATE VIEW vacant_absent_duties
        (duty_number, reason , status) AS
         SELECT duty_no, 'VAC', 'FT' FROM full_time_duty WHERE duty_holder IS NULL
         UNION
         SELECT duty_no, 'VAC', 'PT' FROM part_time_duty WHERE duty_holder IS NULL
         UNION
         SELECT duty_no, 'VAC', 'ID' FROM non_walk_duty WHERE pay_no IS NULL
         UNION
         SELECT duty_number, 'SL', status FROM employee_duty_walk WHERE pay_number IN
                        (SELECT pay_number FROM sick_employees)
         UNION
         SELECT duty_number, 'AL', status FROM employee_duty_walk WHERE pay_number IN
         (SELECT pay_no FROM bookedholiday WHERE week_no = (SELECT week_no FROM
         current_week))!
```

```
CREATE VIEW absent_duties_after_cover
        (duty_number, reason , status) AS
         SELECT duty_number, reason, status FROM vacant_absent_duties
         WHERE duty_number NOT IN (SELECT duty_no FROM duty_cover)!


CREATE VIEW spare_employees
        ("name", pay_number, grade, office, status) AS
         SELECT employee_name, pay_no, grade, office, status FROM employee WHERE pay_no
         NOT IN (SELECT pay_number FROM employee_duty_walk)!


CREATE VIEW display_duty_walk_info
        ("Name", "Office", "Pay_number", "Grade", "Duty_number", "Walk_type", "Delivery_method",
         "Status", "Reason", "Cover_type") AS
         SELECT "name", office, pay_number, grade, duty_number, walk_type, delivery_method,
         status, reason, cover_type
         FROM employee_duty_walk LEFT OUTER JOIN duty_cover_info ON
         employee_duty_walk.duty_number = duty_cover_info.duty_no!


CREATE VIEW where_employees_live
        (name, office, pay_number, grade, phone, address) AS
         SELECT employee_name, office, pay_no, grade, phone, full_address(house_no, post_code)
         FROM employee!


CREATE VIEW absent_employees
        (pay_no) AS
        SELECT pay_no FROM sick_absence
        UNION
        SELECT pay_no FROM bookedholiday WHERE week_no = (SELECT week_no FROM
        current_week)!


CREATE VIEW available_spare_employees
        ("name", pay_number, grade, office, status) AS
        SELECT * FROM spare_employees WHERE pay_number NOT IN
        (SELECT pay_no FROM absent_employees)!
```

D.1.5 Triggers definitions

```
CREATE TRIGGER add_modify_fulltime_duty
        BEFORE INSERT, UPDATE ON full_time_duty
        REFERENCING NEW AS new_full_time_duty
        FOR EACH ROW
        BEGIN
                DECLARE invalid_duty_details EXCEPTION FOR SQLSTATE '99999';
                IF (
                     //// ensures that duty_holder is unique
                    (new_full_time_duty.duty_holder IN (SELECT duty_holder FROM full_time_duty))

                    //// ensures that duty_holder does not exist in part_time_duty or non_walk_duty
                    OR
                    (new_full_time_duty.duty_holder IN (SELECT duty_holder FROM part_time_duty))
                     OR
                     (new_full_time_duty.duty_holder IN (SELECT pay_no FROM non_walk_duty))

                     //// ensures that duty_no is a primary key across the subtypes of duty
                     OR
                     (new_full_time_duty.duty_no IN (SELECT duty_no FROM part_time_duty))
                     OR
                     (new_full_time_duty.duty_no IN (SELECT duty_no FROM non_walk_duty))

                    //// ensure only full time employees can participate in PermanentlyCovers with
                    full_time_duty
                    OR
                     ((SELECT status FROM employee WHERE employee.pay_no =
                            new_full_time_duty.duty_holder) <> 'FT')

                    //// ensure only employees and duties from the same office can participate in
                    PermanentlyCovers
                     OR
                     ((SELECT LEFT (employee.office,1) FROM employee WHERE employee.pay_no
                                    = new_full_time_duty.duty_holder)
                                     <> (SELECT LEFT (new_full_time_duty.duty_no,1) ) )

                    //// ensure that employee covering a day off does not participate in
                    PermanentlyCovers
                    OR
                    (new_full_time_duty.day_off_cover IN (SELECT duty_holder FROM
                                    full_time_duty))
                    OR
                    (new_full_time_duty.day_off_cover IN (SELECT duty_holder FROM
                                    part_time_duty))
                     OR
                     (new_full_time_duty.day_off_cover IN (SELECT pay_no FROM non_walk_duty)))
                THEN
                        SIGNAL invalid_duty_details;
                END IF;
        END!
```

```
CREATE TRIGGER add_modify_parttime_duty
        BEFORE INSERT, UPDATE ON part_time_duty
        REFERENCING NEW AS new_part_time_duty
        FOR EACH ROW
        BEGIN
                DECLARE invalid_duty_details EXCEPTION FOR SQLSTATE '99999';
                IF (
                   //// ensures that duty_holder is unique
                   (new_part_time_duty.duty_holder IN (SELECT duty_holder FROM
                          part_time_duty))

                    //// ensures that duty_holder does not exist in full_time_duty or non_walk_duty
                   OR
                   (new_part_time_duty.duty_holder IN (SELECT duty_holder FROM full_time_duty))
                   OR
                   (new_part_time_duty.duty_holder IN (SELECT pay_no FROM non_walk_duty))

                    //// ensures that duty_no is a primary key across the subtypes of duty
                   OR
                   (new_part_time_duty.duty_no IN (SELECT duty_no FROM full_time_duty))
                   OR
                   (new_part_time_duty.duty_no IN (SELECT duty_no FROM non_walk_duty))

                    //// ensure only part time employees can participate in PermanentlyCovers with
                    part_time_duty
                   OR
                   ((SELECT status FROM employee WHERE employee.pay_no =
                   new_part_time_duty.duty_holder) <> 'PT')

                    //// ensure only employees and duties from the same office can participate in
                    PermanentlyCovers
                   OR
                   ((SELECT LEFT (employee.office,1) FROM employee WHERE employee.pay_no
                          = new_part_time_duty.duty_holder)
                                         <> (SELECT LEFT (new_part_time_duty.duty_no,1) ) ) )
                THEN
                        SIGNAL invalid_duty_details;
                END IF;
        END!
```

```
CREATE TRIGGER add_modify_non_walk_duty
        BEFORE INSERT, UPDATE ON non_walk_duty
        REFERENCING NEW AS new_non_walk_duty
        FOR EACH ROW
        BEGIN
                DECLARE invalid_duty_details EXCEPTION FOR SQLSTATE '99999';
                IF (

                    //// ensures that duty_holder does not exist in full_time_duty or part_time_duty
                   (new_non_walk_duty.pay_no IN (SELECT duty_holder FROM part_time_duty))
                   OR
                   (new_non_walk_duty.pay_no IN (SELECT duty_holder FROM full_time_duty))

                    //// ensures that duty_holder is unique
                   OR
                   (new_non_walk_duty.pay_no IN (SELECT pay_no FROM non_walk_duty))


                    //// ensures that duty_no is a primary key across the subtypes of duty
                   OR
                   (new_non_walk_duty.duty_no IN (SELECT duty_no FROM full_time_duty))
                   OR
                   (new_non_walk_duty.duty_no IN (SELECT duty_no FROM part_time_duty))

                    //// ensure only employees and duties from the same office can participate in
                    PermanentlyCovers
                   OR
                   ((SELECT LEFT (employee.office,1) FROM employee WHERE employee.pay_no
                   = new_non_walk_duty.pay_no)
                                  <> (SELECT LEFT (new_non_walk_duty.duty_no,1) ) ) )


                    //// ensure only employees with grades 'ex_phg' and 'opgdriver' can participate in
                    PermanentlyCovers with non_walk_duty
                   OR
                   ((SELECT grade FROM employee WHERE employee.pay_no =
                   new_part_time_duty.duty_holder) NOT IN ('ex_phg', 'opgdriver'))


                THEN
                        SIGNAL invalid_duty_details;
                END IF;
        END!



// ensures that annual leave doesn't exceed the quota
CREATE TRIGGER add_annual_leave
        BEFORE INSERT, UPDATE ON bookedholiday
        REFERENCING NEW AS new_annual_leave
        FOR EACH ROW
        BEGIN
                DECLARE leave_exceed_quota EXCEPTION FOR SQLSTATE '99999';
                IF ((SELECT COUNT(pay_no) FROM bookedholiday WHERE
                                bookedholiday.week_no = new_annual_leave.week_no)
                                >= (SELECT quota FROM week WHERE week.week_no =
                                new_annual_leave.week_no)) THEN
                        SIGNAL leave_exceed_quota;
                END IF;
        END!
```

```
CREATE TRIGGER add_modify_duty_cover
        BEFORE INSERT, UPDATE ON duty_cover
        REFERENCING NEW AS new_duty_cover
        FOR EACH ROW
        BEGIN
                DECLARE invalid_dutycover_details EXCEPTION FOR SQLSTATE '99999';

                //// Make sure duty cover is for a vacant or absent duty
                IF (NOT (new_duty_cover.duty_no IN (SELECT duty_number FROM
                                vacant_absent_duties)))
                THEN
                        SIGNAL invalid_overtime_details;
                END IF;

                //// Make sure the employee covering the duty is not absent from work
                IF ((new_duty_cover.cover IN (SELECT pay_number FROM sick_employees))
                            OR
                            (new_duty_cover.cover IN (SELECT pay_no FROM bookedholiday
                             WHERE week_no = (SELECT week_no FROM current_week))))
                THEN
                        SIGNAL invalid_dutycover_details;
                END IF;
        END!
```

```
CREATE TRIGGER add_modify_over_time
        BEFORE INSERT, UPDATE ON over_time
        REFERENCING NEW AS new_over_time
        FOR EACH ROW
        BEGIN
                DECLARE invalid_overtime_details EXCEPTION FOR SQLSTATE '99999';

                //// Make sure overtime is used to cover a vacant or absent duty
                IF (NOT (new_over_time.duty_no IN (SELECT duty_number FROM
                          absent_duties_after_cover)))
                THEN
                        SIGNAL invalid_overtime_details;
                END IF;

                //// Make sure the employee performing the overtime is not absent from work
                IF ((new_over_time.pay_no IN (SELECT pay_number FROM sick_employees))
                            OR
                            (new_over_time.pay_no IN (SELECT pay_no FROM bookedholiday
                            WHERE
                                    week_no = (SELECT week_no FROM current_week))))
                THEN
                        SIGNAL invalid_overtime_details;
                END IF;

        END!
```

```
// ensure that employee entered as sick are not participating in ScheduledFor or Performs
CREATE TRIGGER add_modify_sick_absence
        AFTER INSERT, UPDATE ON sick_absence
        REFERENCING NEW AS new_sick_absence
        FOR EACH ROW
        BEGIN
                IF (new_sick_absence.pay_no IN (SELECT pay_no FROM over_time))
                THEN
                        DELETE FROM over_time WHERE new_sick_absence.pay_no =
                                        over_time.pay_no;
                END IF;

                IF (new_sick_absence.pay_no IN (SELECT cover FROM duty_cover))
                THEN
                        DELETE FROM duty_cover WHERE new_sick_absence.pay_no =
                                        duty_cover.cover;
                END IF;
        END!
```

```
// maintains the referential integrity with part_time_duty and full_time_duty
CREATE TRIGGER add_modify_walk
        BEFORE INSERT, UPDATE ON walk
        REFERENCING NEW AS new_walk
        FOR EACH ROW
        BEGIN
                DECLARE invalid_walk_details EXCEPTION FOR SQLSTATE '99999';
                IF (new_walk.status = 'PT')
                THEN
                        IF (NOT (new_walk.duty_no IN (SELECT duty_no FROM part_time_duty)))
                        THEN
                                SIGNAL invalid_walk_details;
                        END IF;
                END IF;

                IF (new_walk.status = 'FT')
                THEN
                        IF (NOT (new_walk.duty_no IN (SELECT duty_no FROM full_time_duty)))
                        THEN
                                SIGNAL invalid_walk_details;
                        END IF;

                END IF;
        END!
```

D.1.6 Functions definitions

```
CREATE FUNCTION duty_to_office (dutyno VARCHAR(4))
        RETURNS VARCHAR(10)
        BEGIN
                DECLARE adelivery_office VARCHAR(10);
                IF (LEFT(dutyno,1) = 'E')
                        THEN
                                SET adelivery_office = 'East';
                        END IF;
                IF (LEFT(dutyno,1) = 'N')
                        THEN
                                SET adelivery_office = 'North';
                        END IF;
                IF (LEFT(dutyno,1) = 'S')
                        THEN
                                SET adelivery_office = 'South';
                        END IF;
                IF (LEFT(dutyno,1) = 'W')
                        THEN
                                SET adelivery_office = 'West';
                        END IF;

                RETURN (adelivery_office);
        END!


CREATE FUNCTION full_address (buildingno VARCHAR(4), postcode VARCHAR(8))
        RETURNS VARCHAR(75)
        BEGIN
                DECLARE fulladdress VARCHAR(80);
                DECLARE street VARCHAR(20);
                DECLARE postalarea VARCHAR(20);
                DECLARE post_town VARCHAR(14);
                SELECT street_name, postal_area, town
                INTO street, postalarea, post_town
                FROM postal_address
                WHERE (post_code = postcode) AND (building_no = buildingno);
                SET fulladdress = buildingno || ', ' || street || ', ' || postalarea || ', ' || post_town || ', ' ||
                                        postcode;
                RETURN (fulladdress);
        END!
```

D.1.7 Procedures definitions

```
// A procedure to set the current week in the finantial year
CREATE PROCEDURE set_week(IN weekno SMALLINT)
        BEGIN
                UPDATE current_week
                SET week_no = weekno;
        END!



CREATE PROCEDURE process_duty_cover()
        BEGIN
            DECLARE v_duty_no CHAR(4);
            DECLARE v_reason VARCHAR(3);
            DECLARE v_cover_type VARCHAR(10);
            DECLARE c_vacant_absent_duty CURSOR FOR
                        SELECT duty_number, reason FROM vacant_absent_duties;
            DECLARE err_notfound EXCEPTION FOR SQLSTATE '02000';

            DELETE FROM duty_cover_info;
            OPEN c_vacant_absent_duty;
             WHILE SQLSTATE <> err_notfound LOOP
                FETCH c_vacant_absent_duty INTO v_duty_no, v_reason;
                IF (SQLSTATE <> err_notfound) THEN

                        SET v_cover_type = '';

                        IF (v_duty_no IN (SELECT duty_no FROM duty_cover))
                                THEN SET v_cover_type = 'duty cover';
                        END IF;

                        IF (v_duty_no IN (SELECT duty_no FROM over_time))
                                THEN SET v_cover_type = 'Overtime';
                        END IF;

                        INSERT INTO duty_cover_info
                        VALUES (v_duty_no, v_reason, v_cover_type);
                END IF;
            END LOOP;
        END!



CREATE PROCEDURE add_over_time(IN dutyno CHAR(4), payno CHAR(5), ot_duration
SMALLINT)
BEGIN
        DECLARE err_input EXCEPTION FOR SQLSTATE '99999';
        IF ((SELECT overtime_availability FROM employee WHERE pay_no = payno) = 'yes' )
        THEN
                INSERT INTO over_time VALUES (dutyno, payno, ot_duration);
        ELSE
                SIGNAL err_input;
        END IF
END!
```

D.2 SQL Data Manipulation Statements

D.2.1 SQL statements to populate the tables

```
INSERT INTO postal_address VALUES
        ('CV1 1AA','40','BISHOPS STREET','','COVENTRY',1, 'east');
INSERT INTO delivery_office VALUES
        ('West','02476557263','G Sandeep','40','CV1 1AA');
INSERT INTO delivery_office VALUES
        ('South','02476557264','D Snowdon','40','CV1 1AA');
```

APPENDIX E The Delivery Office Sample Tables

Company Vehicle Table

reg_no	vehicle_no	mot_due_date	vehicle_size	model	office_name
X467 GFD	CV1	12/08/03	150	FORD	
BN52ARS	CV2	05/02/05	50	FORD	East
T765 VBC	CV3	07/11/02	250	FORD	North
SF02YNG	CV4	03/05/04	50	VAUXHALL	West
Y543 RAD	CV5	27/11/02	150	RENAULT	South
Y786 MUY	CV6	10/09/03	50	VAUXHALL	North
KN02HJC	CV7	10/04/05	50	FORD	East
OU51DAL	CV8	07/07/04	150	FORD	West
TN02OIO	CV9	10/05/04	50	FORD	South

Private Vehicle Table

reg_no	model	color	license_no
M505 RAU	Ford Escort	Blue	MAGEE563232ER3OM
T485 NBV	Vauxhall Vectra	Green	THOMA122334MT8BN
CK02MNA	Peugeot 406	Silver	STEVE908765OD4TY
DN51JKL	Honda Civic	Red	JONES501389YA8NM
X981 SHR	Nissan Micra	Yellow	FOXXY444908SP0TN
D123 TEW	Ford Fiesta	Grey	MCCAN012799JK5XY
L459 ZCV	Rover Tomcat	Silver	MOHAM982365SM8TY
P45 ERT	Renault Clio	Blue	PHILI657890MK0OI
J543 NAZ	BMW 320i SE	Silver	AHMED480546AM1TU

Delivery Office Table

office_name	telephone_no	manager_name	building_no	post_code
east	02476557261	Jon Campbell	40	CV1 1AA
North	02476557262	Steve Moore	40	CV1 1AA
West	02476557263	G Sandeep	40	CV1 1AA
South	02476557264	D Snowdon	40	CV1 1AA

Licensed Employee Table

license_no	vehicle_size	pay_no	vehicle_no
JOHNA862088MA3ER	50	86961	SF02YNG
MAGEE563232ER3OM	50	99856	BN52ARS
MOHAM982365SM8TY		88974	
FRANC981234EN1ZX	50	10088	Y786 MUY
THOMA122334MT8BN	50	23771	TN02OIO
PHILI657890MK0OI		78541	
RUSHT563123MH7WE		87961	
JONES501389YA8NM		12458	
SINGH096354JS4CV		19700	
WILCO455668JM1QW		54213	
STEVE908765OD4TY		99471	
FOXXY444908SP0TN		31692	
MCCAN012799JK5XY		10023	
GEORG111090NJ9LP		77890	
AHMED480546AM1TU		77564	
MICHA009871SR6BG		58470	

Walk Table

walk_number	office_name	walk_type	delivery_method	remarks	status
1	East	Town	Walk	Beware of the traffic in the dual carriage way when crossing to High Street	FT
2	East	Town	Walk		PT
3	East	Town	Walk		FT
4	East	Bulk	Van		PT
1	North	Town	Walk		PT
2	North	Town	Walk		PT
3	North	Bulk	Van		FT
4	North	Rural	Cycle		FT
1	West	Town	Walk		PT
2	West	Town	Walk		FT
3	West	Town	Walk		FT
4	West	Bulk	Van		FT
1	South	Town	Walk		FT
2	South	Town	Walk		PT
3	South	Rural	Cycle		FT
4	South	Bulk	Van		FT

Part Time Duty Table

duty_no	duty_holder	duty_hours
E004	87961	27
E002	78541	30
N001	24661	30
N002	88974	30
W001	54213	27
S002	55355	27
E007		30

Full Time Duty Table

duty_no	duty_holder	day_off_cover
E001	21652	
E003	12458	
N003	10023	
N004	75632	
W002	19700	
W003	11543	
W004	31692	
S001	77890	
S003	77564	
S004		
S007		

Non Walk Duty Table

duty_no	pay_no	duty_details
E005	20034	Sort the special delivery items into walks and handles customer querie
W005	99471	Sort the special delivery items into walks and handles customer querie
N005	63263	Sort the special delivery items into walks and handles customer querie
S005	96742	Sort the special delivery items into walks and handles customer querie
E006	99856	drop bags and packets for E001,E002,E003 & E004
N006	10088	drop bags and packets for N001,N002,N003 & N004
S006	23771	drop bags and packets for S001,S002,S003 & S004
W006	86961	drop bags and packets for W001,W002,W003 & W004
N007		Deal with returned mail and surcharges

Postal Address Table

post_code	building_no	street_name	postal_area	town	walk_no	office
CV3 3KB	45	Abbey Road	Stivichall	COVENTRY	2	south
CV3 3KB	89	Abbey Road	Stivichall	COVENTRY	2	south
CV3 3ND	36	Abbey Way	Stivichall	COVENTRY	2	south
CV3 3ND	85	Abbey Way	Stivichall	COVENTRY	2	south
CV2 4ZX	12	Adare Drive	Alderman's Green	COVENTRY	4	east
CV2 4ZX	40	Adare Drive	Alderman's Green	COVENTRY	4	east
CV4 5TX	34	Aragon House	Hawkes End	COVENTRY	2	west
CV4 5TX	65	Aragon House	Hawkes End	COVENTRY	2	west
CV1 1AB	5	Armorial Road	City Centre	COVENTRY	1	east
CV1 1AB	7	Armorial Road	City Centre	COVENTRY	1	east
CV2 3NN	12a	Asthill Croft	Walsgrave on Sowe	COVENTRY	3	east
CV2 3NN	13b	Asthill Croft	Walsgrave on Sowe	COVENTRY	3	east
CV3 3VB	30	Bankside Close	Stivichall	COVENTRY	2	south
CV3 3VB	63	Bankside Close	Stivichall	COVENTRY	2	south
CV3 2ZX	322	Bartholomew Court	Binley	COVENTRY	1	south
CV3 2ZX	76	Bartholomew Court	Binley	COVENTRY	1	south
CV4 4KB	6	Benedictine Road	Earlsdon	COVENTRY	2	west
CV4 4KB	89	Benedictine Road	Earlsdon	COVENTRY	2	west
CV1 1AA	30	Bishops Street	City Centre	COVENTRY	1	east
CV1 1AA	40	Bishops Street	City Centre	COVENTRY	1	east
CV3 2JQ	24	Bowater Court	Binley	COVENTRY	1	south
CV3 2JQ	58	Bowater Court	Binley	COVENTRY	1	south
CV3 2HG	57	Burnham Road	Binley	COVENTRY	1	south

CV3 2HG	80	Burnham Road	Binley	COVENTRY	1	south
CV5 2RW	15	Calder Close	Allesley	COVENTRY	3	west
CV5 2RW	18	Calder Close	Allesley	COVENTRY	3	west
CV4 3DD	10	Carthusian Road	Upper Eastern Green	COVENTRY	1	west
CV4 3DD	4	Carthusian Road	Upper Eastern Green	COVENTRY	1	west
CV6 4TR	30	Cleeves Mews	Radford	COVENTRY	2	north
CV6 4TR	88	Cleeves Mews	Radford	COVENTRY	2	north
CV4 2QW	45	Cornelius Street	Tile Hill	COVENTRY	1	west
CV4 2QW	7b	Cornelius Street	Tile Hill	COVENTRY	1	west
CV5 2OU	10	Courtleet Road	Allesley	COVENTRY	3	west
CV5 2OU	76	Courtleet Road	Allesley	COVENTRY	3	west
CV6 3PO	4	Daintree Croft	Longford	COVENTRY	2	north
CV6 3PO	7	Daintree Croft	Longford	COVENTRY	2	north
CV6 2GH	4	Daventry Road	Little Heath	COVENTRY	1	north
CV6 2GH	6	Daventry Road	Little Heath	COVENTRY	1	north
CV4 1DZ	7a	Franciscan Road	Kirby Corner	COVENTRY	1	west
CV4 1DZ	90	Franciscan Road	Kirby Corner	COVENTRY	1	west
CV5 1PU	1	Frankpledge Road	Westwood Heath	COVENTRY	3	west
CV5 1PU	9	Frankpledge Road	Westwood Heath	COVENTRY	3	west
CV4 6MN	13	Galeys Road	Pickford Green	COVENTRY	2	west
CV4 6MN	24	Galeys Road	Pickford Green	COVENTRY	2	west
CV6 7DH	24	Glover Street	Great Heath	COVENTRY	4	north
CV6 7DH	56	Glover Street	Great Heath	COVENTRY	4	north
CV3 4GA	1	Hilllfray Drive	Willenhall	COVENTRY	3	south
CV3 4GA	76	Hilllfray Drive	Willenhall	COVENTRY	3	south
CV2 3ML	33	Hiron Croft	Walsgrave on Sowe	COVENTRY	3	east
CV2 3ML	50	Hiron Croft	Walsgrave on Sowe	COVENTRY	3	east
CV1 1DS	1	Horsford Road	City Centre	COVENTRY	1	east
CV1 1DS	3	Horsford Road	City Centre	COVENTRY	1	east
CV6 2NM	15	Howard Mews	Keresley	COVENTRY	1	north
CV6 2NM	19	Howard Mews	Keresley	COVENTRY	1	north
CV2 3AD	44	Humphrey Burton Road	Walsgrave on Sowe	COVENTRY	3	east
CV2 3AD	60	Humphrey Burton Road	Walsgrave on Sowe	COVENTRY	3	east
CV6 1BB	2a	Kenilworth Court	Court House Green	COVENTRY	1	north
CV6 1BB	2b	Kenilworth Court	Court House Green	COVENTRY	1	north
CV6 6VB	45	Lichfield Road	Foleshill	COVENTRY	4	north
CV6 6VB	78	Lichfield Road	Foleshill	COVENTRY	4	north
CV2 2TY	13	Michaelmas Road	Upper Stoke	COVENTRY	2	east
CV2 2TY	18	Michaelmas Road	Upper Stoke	COVENTRY	2	east
CV2 1RJ	109	Orchard Crescent	Upper Stoke	COVENTRY	2	east
CV2 1RJ	2	Orchard Crescent	Upper Stoke	COVENTRY	2	east
CV6 6GE	13	Quinton Road	Foleshill	COVENTRY	4	north
CV6 6GE	90	Quinton Road	Foleshill	COVENTRY	4	north
CV5 4DS	23	Riverside Close	Chapel Fields	COVENTRY	4	west
CV5 4DS	45	Riverside Close	Chapel Fields	COVENTRY	4	west
CV3 4KL	34	Rutherglen Avenue	Willenhall	COVENTRY	3	south
CV3 4KL	84	Rutherglen Avenue	Willenhall	COVENTRY	3	south
CV3 5ER	8	Seedfield Croft	Binley	COVENTRY	4	south
CV3 5ER	9	Seedfield Croft	Binley	COVENTRY	4	south
CV6 5FD	16	Shortley Road	Little Heath	COVENTRY	3	north
CV6 5FD	34	Shortley Road	Little Heath	COVENTRY	3	north
CV2 1KY	1	Stoney Road	Upper Stoke	COVENTRY	2	east
CV2 1KY	5	Stoney Road	Upper Stoke	COVENTRY	2	east
CV6 5RT	1	Swifts Corner	Little Heath	COVENTRY	3	north
CV6 5RT	2	Swifts Corner	Little Heath	COVENTRY	3	north
CV5 3SC	65	The Avenue	Brownshill Green	COVENTRY	4	west
CV5 3SC	69	The Avenue	Brownshill Green	COVENTRY	4	west
CV2 4BA	3	The Monks Croft	Alderman's Green	COVENTRY	4	east

CV2 4BA	6	The Monks Croft	Alderman's Green	COVENTRY	4	east
CV6 5AH	105	The Mount	Keresley	COVENTRY	3	north
CV6 5AH	89	The Mount	Keresley	COVENTRY	3	north
CV5 3VN	45	Tonbridge Road	Brownshill Green	COVENTRY	4	west
CV5 3VN	62	Tonbridge Road	Brownshill Green	COVENTRY	4	west
CV2 4JO	4	Townsend Road	Alderman's Green	COVENTRY	4	east
CV2 4JO	50	Townsend Road	Alderman's Green	COVENTRY	4	east
CV3 5FK	20	Troyes Close	Binley	COVENTRY	4	south
CV3 5FK	93	Troyes Close	Binley	COVENTRY	4	south
CV3 5YA	78	Wanley Road	Binley	COVENTRY	4	south
CV3 5YA	85	Wanley Road	Binley	COVENTRY	4	south
CV6 4FF	47	Whitley Court	Radford	COVENTRY	2	north
CV6 4FF	53	Whitley Court	Radford	COVENTRY	2	north
CV3 4SO	4	Woodstock Road	Willenhall	COVENTRY	3	south
CV3 4SO	7	Woodstock Road	Willenhall	COVENTRY	3	south

Employee Table

date_of_birth	date_of_entry	grade	phone	badge_no	skills	overtime_availability	office	house_no	post_code	status
03/03/76	10/01/02	Opg	02476239087	1030	working safely course, first aid	yes	East	57	CV3 2HG	FT
16/03/79	10/01/02	OPGDriver	02476559884	3695		yes	East	15	CV6 2NM	FT
10/05/75	25/04/95	OPG	07786622431	6484		no	East	1	CV5 1PU	PT
24/02/70	08/08/92	ex_PHG	07596121121	2133		no	East	23	CV5 4DS	FT
06/09/74	06/07/98	OPG	07889256987	4452		no	East	45	CV5 4DS	PT
29/04/70	08/09/91	OPG		6547		yes	East	34	CV3 4KL	FT
18/08/69	15/07/80	OPG		1432		no	West	84	CV3 4KL	FT
07/01/80	04/04/01	OPG	02476551552	3435		yes	West	8	CV3 5ER	FT
06/08/75	04/09/02	OPGDriver	07986556211	2285		yes	West	9	CV3 5ER	FT
30/08/76	25/04/99	OPG	02476312521	7451		yes	West	16	CV6 5FD	PT
25/10/69	15/06/02	ex_PHG	07956454580	3312	First aid	yes	West	34	CV6 5FD	FT
10/01/71	05/05/95	OPG	07798123456	0701	First aid	yes	West	1	CV2 1KY	FT
12/07/70	01/03/88	OPG	02476778830	1230	First aid	yes	North	5	CV2 1KY	FT
23/05/60	13/10/85	OPG	02476453788	0270	Manual handling and lifting	yes	North	1	CV6 5RT	PT
04/09/78	01/01/01	OPG	07988614214	4978	First aid	yes	North	2	CV6 5RT	PT
01/01/65	14/07/87	ex_PHG	02476854124	5421		yes	North	65	CV5 3SC	FT
15/08/72	07/09/00	OPGDriver	02476884736	8795		yes	North	69	CV5 3SC	FT
30/09/75	01/05/92	OPG	02476737684	2254		no	North	3	CV2 4BA	FT
22/10/65	24/11/98	OPGDriver		1601	sign language for deaf employees	yes	South	6	CV2 4BA	FT
02/02/80	08/05/00	OPG	02476885965	5532		no	South	105	CV6 5AH	FT
25/06/62	09/01/80	OPG	07885641365	8542		no	South	89	CV6 5AH	FT
22/07/71	20/06/99	OPG	02476885414	2001		no	South	50	CV2 4JO	FT
23/05/76	08/10/98	OPG	02476100123	5103		yes	South	20	CV3 5FK	PT
25/07/77	03/09/00	ex_PHG	02476947586	3356		yes	South	93	CV3 5FK	FT

APPENDIX F Structure Of A PAF Address [Royal Mail, 2000, ch. 3]

F.1 PAF Details

PAF holds information about delivery points, a delivery point being a property, an Organisation or a letterbox. The names of private individuals are not normally held on PAF. They are present only when there is no other method of identifying a delivery point.

The information held for a Small User delivery point is split into two parts, these being Organisation and address. The vast majority of delivery points are residential addresses, which do not contain any Organisation details. The delivery points that contain Organisation details are called either Small or Large User Organisations. Large User Organisations are given their own Postcode, whereas a number of Small User Organisations and/or residential addresses can share a single Postcode.

All address text on PAF is held in upper case, except labels, which are available in mixed case format.

[..]

F.2 Organisation Details

The name of the Organisation is held. The Department can also be held.

Field name	max. field length
Organisation Name	60
Department Name	60

F.3 Address Details

An address is composed of the following address elements. Not all are present for every address, as addresses on PAF may be composed of different subsets of the elements. Postcode and Post Town are the only elements that are mandatory, i.e. they will be present for each address. The County is no longer required as part of a correct postal address. For further details refer to Flexible Addressing at the end of this section.

		max. field length
Premise Elements	- Sub Building Name	30
	Building Name	50
	Building Number	4
Thoroughfare elements	- Dependent Thoroughfare Name	60
	Dependent Thoroughfare Descriptor	20
	Thoroughfare Name	60
	Thoroughfare Descriptor	20
Locality elements	- Double Dependent Locality	35
	Dependent Locality	35
	Post Town	30
	County	30
	Postcode	7

F.4 PO Box Details

	max. field length
PO Box details may be present for Large Users only	6