DATABASE MANAGEMENT SYSTEMS

ER. V. K. JAIN
B.E., M.TECH
FIE, FIETE, MISHARE

Published by
dreamtech
19-A, Ansari Road, Daryaganj,
New Delhi-110002

Edition: 2006

Printed at : PrintWell Offset, Bhikaji Cama Place, New Delhi-66

Preface

The present day world belongs to information technology and the processing of information. The application of computers has spread to influence every domain of human activity. A plethora of data is available just for the click of the mouse; in no time, right from where one happens to be. However, unless this ocean of information is harnessed with some powerful technique of data management, none of its bounties can be availed for practical purposes. This accounts for Data Management being considered as vital as, and as imperative as data procurement.

This book is devoted to the subject of Database Management Systems and is intended for the general learner as well as for those who need to learn about Databases as part of the course they might be pursuing or for taking competitive exams which include Database in the syllabus.

The book begins with an informative and interesting account of the information explosion that has made the times we live in, a turning point in the history of mankind. The reader is then briefed about the general principles governing the technique of Database Management. Following this, the organization of traditional Database Models and their limitations are explained so as to set the stage for presenting the modern techniques. The Standard Query Language and its various versions are detailed thereafter. Then, the various modern Database Systems that are equipped to handle the current situation, which could be attributed to the Internet and other means of fast data transfer technologies, are explained exhaustively.

The book maintains a simple and easy-to-understand style of narration and every effort has been made to render the presentation interesting. The entire field of the subject dealt with has been covered so that the reader genuinely profits by reading this book.

Readers are requested to send their suggestions for improvement and to point out shortcomings or omissions, if any. These will be thankfully acknowledged and considered for incorporation in the future editions.

Er. V. K. Jain

CONTENTS

CHAPTER 1

OVERVIEW OF DATABASE MANAGEMENT SYSTEMS

1.1 Data, Information and Knowledge

Data is the name given to basic facts and entities such as names and numbers. Good examples of data are dates, weights, prices, costs, numbers of items sold, employee names, product names, addresses, tax codes, registration marks etc.

Data consists of a series of facts or statements that may have been collected, stored, processed and/or manipulated but have not been organised or placed into context. When data is organised, it becomes information. Information can be processed and used to draw generalized conclusions or knowledge.

For example, a file listing of all the orders placed through an online service is an example of data. If we sort the data by ZIP code and summarize the number of orders that come from each city, we have created information. We can create knowledge by taking this information and making statements such as "Most orders for Widget X come from the northeastern United States."

This term is used to describe basic facts about the activities of a system, that could be a business house, production centre or educational institution. Data is generally in form of names and numbers, times, dates, weights, prices, costs, employee's name, product's name, names of books, schools, students, teachers, roll numbers, etc.

In business, the data input is a collection of facts about environmental elements: consumers, suppliers, competitors, government, and the like. Data are something like raw materials used in production processes in factories or industries.

1.1.1 Information

Information is data which has been converted into a more useful or intelligible form. It is the set of data which has been converted or organised into a more useful or intelligible form for direct utilisation of mankind, as information helps human beings in their decision-making process. Examples are: time table, merit list, report card, headed tables, printed documents, pay slips, receipts, reports etc. The information is obtained by assembling items of data into a meaningful form. Whereas marks obtained by students and their roll numbers form data, the report card/sheet is the information. Other forms of information are pay-slips, schedules, reports, work-sheet, bar-charts, invoices, account-returns. It may be remembered that information may further be processed and/or manipulated to form knowledge.

Information output from one system may be data input for another system. An information system is a set of processes, executed on raw data, to produce information, which will be useful in decision making.

An information system must have a full range of functions to achieve its purpose, including observation, measurement, description, explanation, forecasting and decision making.

The heart of the integration of information needed in MIS (Management Information System) is a data base. A data base is an organised repository of the organisation's information resources (internal and possible containing some external data), including raw data and procedures. The idea is that the data base consists of most of the data available, in the organisation and can be accessed by different managers for their varied uses, One manager may access the data base for planning, another manager may need data for controlling, and in general all managers may need to access the data base for decision making.

Information has been recognised as one of the critaria: corporate respiratory, which facilitates better utilisation of other important resources such as men, machines, materials, money and methods.

1.1.2 Knowledge

Information containing wisdom is knowledge. Knowledge is of two types:

- **Facts based or information based:** Knowledge gained from fundamentals and through experiments. The knowledge like the information contained in fundamental science, which has been derived from experiments, rules, regulations that are commonly agreed by experts.
- **Heuristic Knowledge:** It is knowledge of good practice, experience and good judgment like hypothesis. It is the knowledge underlying "expertise", rules of thumb, rules of good guessing, that usually achieve desired results but do not guarantee them.

1.1.3 Difference between Information and Data

In this chapter it is necessary to make a distinction between two terms, which are usually used interchangeably, namely information and data. Data is the material, on which computer programs work upon. It can be numbers, letters of the alphabet, words, special symbols. But by themselves they have no meaning. For example, the following sequence of digits 240343 is

meaningless by itself. Since, it could refer to a date of birth, a part number for an automobile, the number of rupees spent on a project, population of a town, the number of people employed in a large organisation, etc. Once we know what the sequence refers to, then it becomes meaningful and can be called information. When we write above as 24-03-43, it may mean date of birth as 24th March, 1943.

A set of words would be data but text would be information. For example "ANNUAL-EXAMINATION, AMITABH, JYOTSNA, PHYSICS" is a set of data and "JYOTSNA SCORED THE HIGHEST MARKS IN PHYSICS IN ANNUAL- EXAMINATION" is information. And that is not the end of affair, Information may be processed or manipulated further of course e.g. a printed text may be reorganised. Also information received from one source or system may become data input for another system. Say in case of examination result individual's score card is an information about the individual but is actually a source of input data for making merit list of the entire board or university.

In the business the data input is a collection of facts about environmental elements: consumers, suppliers, competitors, government, and the like. As data, these facts are relatively unprocessed, although they may have been sorted, classified, or summarised as an adjunct to the collection process. Data are something like raw materials used in production processes practiced in factories or industries. In a paper manufacturing factory bamboos and old clothes are the inputs and paper is the output.

1.2 Increasing Use of Data as Resource

Modern civilisation has become so complicated and sophisticated that to survive one has to be competitive. This compels the people to keep himself informed of all types of happenings in the society. With the advent of educational reforms in society, mankind is surrounded with a vast amount of data available. In USA alone about 1 trillion documents are being created every year and this rate is increasing by 70% every year. Modern business management system has also rendered itself to bulk collection of data from various sources, which needs to be rearranged in a fashion so that it can be utilised with minimum possible time. All this needs a high amount of filing, either at data stage or at information stage. No office can be without files, if you go to any tax collection department or municipal office, you will find a high amount of files stacked here and there. Modern rules, regulation and law requires every transaction to happen in a written form, may be an agreement, application, voucher, bill, letter, memo, order etc.

Paper files require a high amount of storage space and paper storage creates several other problems like fire risk, spoilage and deterioration by way of aging micro-organism and humidity etc. Computerisation of documents and files has solved this problem to a great extent. Not only this, but it has led to a high amount of relaxation to human mind, as everything can be automated. Now the document can be created in a number of ways, styles and any number of documents can be created as and when required, without making any mistake.

Humanity is currently leading to office automation, where every transaction in business is conducted through computers. Electronic Data Processing (EDP) systems makes and stores

documents in magnetic media i.e. the soft copy, which is in very condensed form of storage of information. A floppy disk contains information worth 300 typed pages and CD ROM (laser disk) contains matter worth 45,000 typed papers.

The urge for converting facts into useful information is not a phenomenon of modern life. Throughout history, and even prehistory, people have found it necessary to sort data into forms that were easier to understand. For example, the ancient Egyptians recorded the ebb and flow of the Nile River and used this information to predict yearly crop yields. Today, computers convert data about land and water into recommendations to farmers on crop planting. The great Mughal emperor Akbar appointed Raja Todarmal to keep record of property possessed by all farmers and citizens and to keep track of personal and state's properties. That seems to be first ever database organised in written manner by Indians.

In modern days information is needed to run man's own livelihood, to run a system or process or to command a business. Information is needed to:

- To gain knowledge about the surroundings, and whatever is happening in the society and universe.
- To keep the system up-to-date.
- To know about the rules and regulations and bye-laws of society, local government, provincial and central government, associations, clients etc., as ignorance is no bliss.
- Based on above three, to arrive at a particular decision for planning current and prospective actions in process of forming, running and protecting a process or system.

Use of information systems has not been restricted to commerce and industry. In the public sector, governments have felt obliged to collect and maintain information on their operations, finances, and citizenry for as long as there have been governments. And there is no sign that government's appetite for information will be diminished in the future.

Prior to industrialisation, approximately 90% of the labour force was engaged in agriculture, i.e. society was agrarian. Methods of communication were limited and a very small proportion of the labour force was involved with the processing storage and retrieval of information, which in any case merely involved manual paper based methods or through word of mouth.

Industrialisation produced a major shift in the labour force, with the proportion involved in agriculture falling below 10% in the UK. With industrialisation came the beginning of Information Technology and the start of a series of IT developments taking us right up to the present day: Telegraph, Telephone, Radio/TV, Computers, Microelectronics etc.

These new forms of IT, and other developments, produced new forms of work. The larger scale of organisations has given rise to large administrative structures, in which there are large numbers of clerical works and people with technical and managerial skills collectively known as "white collar workers". Computerisation has mainly affected white collar work so far.

Organisations are increasingly recognising that data is an important resource, which, like other resources such as staff and materials, has both value and cost to the enterprise. Consequently data needs to be managed so that it effectively serves the information systems of the organisation.

For this management to be successful there needs to be:

- Knowledge of what data exists and how it is used.
- Control of modifications to existing data or processes using data.
- Control over plans for new uses of data and over the acquisition of new types of data.

This need for control is increased, where data is used for more than one application and the trend towards database solutions naturally means that this is often the case. In such circumstances, the central definition of data and its use is usually delegated to a Data Administration function. The Data Administrator will be concerned with the correct use and maintenance of data, the integration of new applications and amendments, and the implementation details of data storage, access and manipulation. The data dictionary is an essential support tool for the successful performance of these tasks.

In addition to changes in the type of work, there has been an increase in the number of organisations involved in activities other than manufacture. Some such organisations, for example those in the power industries, contribute to manufacturing and provide a general service. As a result of this change only 25% of the labour force remained in organisations directly involved in the manufacture of goods. For a number of reasons, not particularly related to IT, that 25% has fallen to 20% in the last few years and levels of unemployment have risen.

The fact that so few remain in manufacture, although manufacturing continues to generate most wealth, has lead to society today being called "Post Industrial Society".

Looking at the whole of the national and international community, and at the way organisations are run, highlights the fact that modern society is heavily dependent on the communication, processing and storage of information. It is claimed by some, that we are moving towards an "Information Society", in which the majority of the labour force will be engaged in Information Processing and the use of "Information Technology".

1.2.1 Information Explosion

The world produces between 1 and 2 exabytes of unique information per year, which is roughly 250 megabytes for every man, woman, and child on earth. An exabyte is a billion gigabytes. Printed documents of all kinds comprise only .003% of the total. Magnetic storage is by far the largest medium for storing information and is the most rapidly growing, with shipped hard drive capacity doubling every year. Magnetic storage is rapidly becoming the universal medium for information storage.

The cost of magnetic storage is dropping rapidly; as of Fall 2000, a gigabyte of storage costs less than Rs. 10 and it is predicted that this cost will drop to Rs. 1 by 2005. Soon it will be technologically possible for an average person to access virtually all recorded information. The natural question then becomes: how much information is there to store? If we wanted to store "everything," how much storage would it take?

1.2.2 Information Produced by Medium

Most information is stored in four physical media: paper, film, optical (CDs and DVDs), and magnetic. There are very good data for the worldwide production of each storage medium, and there are reasonably good estimates of how much original content is produced in each of these different formats.

Table 1.1 depicts the yearly worldwide production of original stored content as of 1999. In general, the upper estimate is based on the raw data, while the lower estimate reflects an attempt to adjust for duplication and compression. We discuss these adjustments below and in the medium-specific documents. Note that the growth rate estimates are very rough.

Table 1.1: Worldwide production of original content, stored digitally using standard compression methods, in terabytes circa 1999

Type of Content	Terabytes/Year Upper Estimate	Terabytes/Year, Lower Estimate	Growth Rate (%)
Paper Books	8	1	2
Newspapers	25	2	2
Periodicals	12	1	2
Office documents	195	19	2
Subtotal:	240	23	8
Film Photographs	410,000	41,000	5
Cinema	16	16	3
X-Rays	17, 200	17, 200	2
Subtotal:	427,216	58,216	4
Optical Music CDs	58	6	3
Data CDs	3	3	2
DVDs	22	22	100
Subtotal:	83	31	70
Magnetic Camcorder Tape	300,000	300,000	5
PC Disk Drives	766,000	7,660	100
Departmental Servers	460,000	161,000	100
Enterprise Servers	167,000	108,550	100
Subtotal:	1,693,000	577,210	55
TOTAL:	2,120,539	635,480	50

Three striking facts emerge from these estimates. The first is the "paucity of print." Printed material of all kinds makes up less than .003 percent of the total storage of information. This doesn't imply that print is insignificant. Quite the contrary: it simply means that the written word is an extremely efficient way to convey information.

The second striking fact is the "democratisation of data." A vast amount of unique information is created and stored by individuals. Original documents created by office workers are more than 80% of all original paper documents, while photographs and X-rays together are 99% of all original film documents. Camcorder tapes are also a significant fraction of total magnetic tape storage of unique content, with digital tapes being used primarily for backup copies of material on magnetic drives.

As for hard drives, roughly 55% of the total are installed in single-user desktop computers. Of course, much of the content on individual users' hard drives is not unique, which accounts for the large difference between the upper and lower bounds for magnetic storage. However, as more and more image data moves onto hard drives, we expect to see the amount of digital content produced by individuals stored on hard drives increase dramatically.

This democratisation of data is quite remarkable. A century ago the average person could only create and access a small amount of information. Now, ordinary people not only have access to huge amounts of data, but are also able to create gigabytes of data themselves and, potentially, publish it to the world via the Internet, if they choose to do so.

The third interesting finding is the "dominance of digital" content. Not only is digital information production the largest in total, it is also the most rapidly growing. While unique content on print and film are hardly growing at all, optical and digital magnetic storage shipments are doubling each year. Even today, most textual information is "born digital," and within a few years this will be true for images as well. Digital information is inexpensive to copy and distribute, is searchable, and is malleable. Thus the trend towards democratisation of data—especially in digital form—is likely to continue.

1.3 Data Processing Vs Data Management Systems

All data processing systems perform the same five basic operations. These are

- Inputting
- Storing
- Processing,
- Outputting,
- Controlling.

Storage may be both internal and external, and we can describe a data storage hierarchy. Within this hierarchy, in order of increasing data capacity are characters, data elements, records, files, and data bases.

The typical approach to file design has been to set up a separate file for each data processing application. This provides a satisfactory file for that application but leads to several files with similar data which must be separately maintained, and which are not necessarily compatible.

These systems perform the essential role of collecting and processing the daily transactions of the organisation, hence the alternative term, transaction processing. Typically, these include, all forms of ledger keeping, accounts receivable and payable, invoicing, credit control, rate demands, stock movements and so on.

These types of systems were the first to harness the power of the computer and originally were based on centralised mainframe computers. In many cases this still applies, especially for large volume repetitive jobs, but the availability of micro and mini computers has made distributed data processing feasible and popular. Distributed data processing has many variations but in essence means that data handling and processing are carried out at or near the point of use rather than in one centralised location.

Transaction processing is substantially more significant in terms of processing time, volume of input and output, than information production for tactical and strategic planning. Transaction processing is essential to keep the operations of the organisation running smoothly and provides the base for all other internal information support.

1.3.1 Characteristics of Data Processing Systems

These systems are 'pre-specified'; that is their functions, decision rules and output formats cannot usually be changed; by the end user. These systems are related directly to the structure of the organisation's data. Any change in the data they process or the functions they perform usually requires the intervention of information system specialists such as systems analysts and programmers.

Some data processing systems have to cope with huge volumes and a wide range of data types and output formats. As an example, consider the Electricity and Gas Board Billing and Payment Handling systems, the Clearing Bank's Current Accounting systems, the Motor Policy Handling systems of a large insurer and so on. The systems and programming work required for these systems represents a major investment. For example the development of a large scale billing system for a public; utility represents something like 100 man-years of effort.

Of course, data processing also takes place on a more modest scale and the ready availability of application packages i.e. software to deal with a particular administrative or commercial taskmeans that small scale users have professionally written and tested programs to deal with their routine data processing. The better packages provide for some flexibility and the suer can specify within limits, variations in output formats, data types and decision rules.

1.3.2 Evolution of Data Management System

In order to store all the new information, humanity invented the technology of writing. And though great scholars like Aristotle warned that the invention of the alphabet would lead to the subtle, but total demise of the creativity and sensibility of humanity, data began to be stored in voluminous data repositories, called books.

As we know, eventually books propagated with great speed and soon, whole communities of books migrated to the first real "databases", libraries.

Unlike previous versions of data warehouses (people and books), that might be considered the 'australopithecines' of the database lineage, libraries crossed over into the modern day species, though they were incredibly primitive of course.

Specifically, libraries introduced "standards," by which data could be stored and retrieved.

After all, without standards for accessing data, libraries would be like my closet, endless and engulfing swarms of chaos. Books, and the data within books, had to be quickly accessible by anyone if they were to be useful.

In fact, the usefulness of a library, or any base of data, is proportional to its data storage and retrieval efficiency. This one corollary would drive the evolution of databases over the next 2000 years to its current state.

Thus, early librarians defined standardised filing and retrieval protocols. Perhaps, if you have ever made it off the web, you will have seen an old library with its cute little indexing system (card catalog) and pointers (Dewey decimal system).

And for the next couple of thousand years libraries grew, and grew, and grew along with associated storage/retrieval technologies such as the filing cabinet, colored tabs, and three ring binders. All this until one day about half a century ago, some really bright folks including Alan Turing, working for the British government were asked to invent an advanced tool for breaking German cryptographic "Enigma" codes. That day the world changed again. That day the computer (Collossus) was born. The computer was an intensely revolutionary technology of course, but as with any technology, people took it and applied it to old problems instead of using it to its revolutionary potential.

1.4 File-Oriented Approach

Almost instantly, the computer was applied to the age-old problem of information storage and retrieval. After all, by World War Two, information was already accumulating at rates beyond the space available in publicly supported libraries. And besides, it seemed somehow cheap and tawdry to store the entire archives of "The Three Stooges" in the Library of Congress. Information was seeping out of every crack and pore of modern day society.

Thus, the first attempts at information storage and retrieval followed traditional lines and metaphors. The first systems were based on discrete files in a virtual library. In this file-oriented system, a bunch of files would be stored on a computer and could be accessed by a computer operator. Files of archived data were called "tables" because they looked like tables used in traditional file keeping. Rows in the table were called "records" and columns were called "fields".

1.4.1 File Processing Systems

In early processing systems, an organisation's information was stored as groups of records in separate files. These file processing systems (see Fig. 1.1) consisted of a few data files and many application programs. Each file, called a flat file, contained and processed information for one specific function, such as accounting or inventory. Programmers used programming languages such as COBOL to write applications that directly accessed flat files to perform data management services and provide information for users.

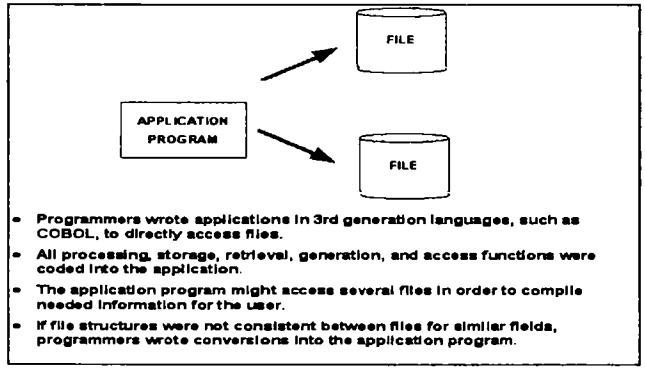


Fig.1.1: File Processing System

In creating the files and applications, developers focused on business processes, or how business was transacted, and their interactions. However, business processes are dynamic, requiring continuous changes in files and applications. In addition, early programmers focused on physical implementation and access procedures when designing a database. These physical procedures were written into database applications; therefore, physical changes resulted in intensive rework on the part of the programmer. As systems became more complex, file processing systems offered little flexibility, presented many limitations, and were difficult to maintain.

1.4.2 Limitations of File Processing Systems

1. **Separated and Isolated Data.**
   To make a decision, a user might need data from two separate files. First, the files were evaluated by analysts and programmers to determine the specific data required from each file and the relationships between the data. Then applications could be written in a third generation language to process and extract the needed data. Imagine the work involved if the data from several files was needed.
2. **Data Redundancy**
   Often, the same information was stored in more than one file. In addition to taking up more file space on the system, this replication of data caused loss of data integrity. For instance, if a customer's address was stored in four different files, an address change would have to be updated in each file separately. If a user was not consistent in updating all files, no one would know which information was correct.
3. **Program—Data interdependence involving file formats and access techniques**
   In file processing systems, files and records were described by specific physical formats that were coded into the application program by programmers. If the format of a certain record was changed, the code in each file containing that format must be updated. For example, a field in the sales file might be coded as "decimal," while the same field in the customer file could be coded as "binary." In order to combine these fields into one application, a programmer would have to write code to convert every value of the "decimal" field in the sales file to a "binary" field (or the reverse) in addition to coding the application. Furthermore, instructions for data storage and access were written into the application's code. Therefore, changes in storage structure or access methods could greatly affect the processing or results of an application.
4. **Difficulty in representing data from the user's view**
   To create useful applications for the user, often data from various files must be combined. In file processing, it was difficult to determine relationships between isolated data in order to meet user requirements.
5. **Data Inflexibility**
   Program-data interdependency and data isolation limited the flexibility of file processing systems in providing users with ad hoc information requests. Because designing applications was so programming-intensive, information requests usually were restricted by MIS department staff. Therefore, users often resorted to manual methods to obtain needed information.

1.4.3 File Management Systems (FMSs)

A traditional file system is made up of many programs accessing many files. These are a much less ambitious concept than database systems. Many products for PCs have the properties of FMSs and database systems. It is the system that an operating system or program uses to organize and keep track of files. For example, a hierarchical file system is one that uses directories to organize files into a tree structure.

Although the operating system provides its own file management system, you can buy separate file management systems. These systems interact smoothly with the operating system but provide more features, such as improved backup procedures and stricter file protection.

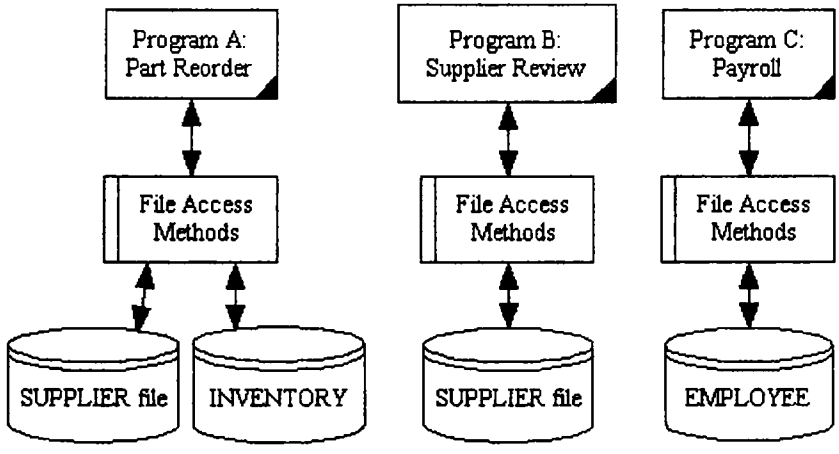


Fig 1.2: File Management Systems

Problems with the traditional approach to data processing system development systems evolve i.e.

- Each new application which uses data has its own copy
- Data redundancy (due to duplication): leads to wasted file space, but more importantly integrity (in this case consistency) problems
- Data dependence: programs depend on data
- Ad hoc inquiry difficulties: traditionally require special program/system. Today 4GLs make this possible but clumsy without a DBMS

Consider the following example:

First Name	Last Name	Email	Phone
V.K	Jain	vkj1944@hotmail.com	011-3316697
Arihant	Negi	arihant_negi@eff.org	011-3318516
Hema	Rajput	hema48@rediff.com	011-7319999
Sunil	Vijay	sunilvijay58@yahoo.com	011-5447376

The "flat file" system was a start. However, it was seriously inefficient. Essentially, in order to find a record, someone would have to read through the entire file and hope it was not the last record. With a hundred thousands records, you can imagine the dilemma.

What was needed, computer scientists thought (using existing metaphors again) was a card catalog, a means to achieve random access processing, that is the ability to efficiently access a single record without searching the entire file to find it.

The result was the indexed file-oriented system in which a single index file stored "key" words and pointers to records that were stored elsewhere. This made retrieval much more efficient. It worked just like a card catalog in a library. To find data, one needed only search for keys rather than reading entire records.

1.4.4 Disadvantage of File Oriented System

However, even with the benefits of indexing, the file-oriented system still suffered from problems including:

- Data Redundancy - The same data might be stored in different places.
- Poor Data Control - Redundant data might be slightly different such as in the case when Ms. Jones changes her name to Mrs. Johnson and the change is only reflected in some of the files containing her data.
- Inability to Easily Manipulate Data - It was a tedious and error prone activity to modify files by hand.
- Cryptic Work Flows - Accessing the data could take excessive programming effort and was too difficult for real-users (as opposed to programmers).
- Consider how troublesome the following data file would be to maintain.

Name	Address	Course	Grade
Mr. V.K Jain	123 Kensigton	Chemistry 102	C+
Mr. V.K Jain	123 Kensigton	Chinese 3	A
Mr. V.K Jain	123 Kensigton	Data Structures	B
Mr. V.K Jain	123 Kensigton	English 101	A
Ms. Hema Rajput	88 West 1st St.	Psychology 101	A
Mrs. Naveena Bajaj	100 Capitol Ln.	Psychology 102	A
Ms. Hema Rajput	88 West 1st St.	Human Cultures	A
Ms. Hema Rajput	88 West 1st St.	European Governments	A

1.5 Database Oriented Approach to Data Management

Simply put, a database is a computerised record keeping system. More completely, it is a system involving data, the hardware that physically stores that data, the software that utiliSes the hardware's file system in order to 1) store the data and 2) provide a standardised method for retrieving or changing the data, and finally, the users who turn the data into information.

Databases, another feature of the 60s, were created to solve the problems with file-oriented systems in that they were compact, fast, easy to use, current, accurate, allowed the easy sharing of data between multiple users, and were secure.

A database might be as complex and demanding as an account tracking system used by a bank to manage the constantly changing accounts of thousands of bank customers, or it could be as simple as a collection of electronic business cards on your laptop.

The important thing is that a database allows you to store data and get it or modify it when you need to easily and efficiently regardless of the amount of data being manipulated. What the data is and how demanding you will be when retrieving and modifying that data is simply a matter of scale. Traditionally, databases ran on large, powerful mainframes for business applications. You will probably have heard of such packages as Oracle 8 or Sybase SQL Server for example.

However, with the advent of small, powerful personal computers, databases have become more readily usable by the average computer user. Microsoft's Access is a popular PC-based engine. More importantly for our focus, databases have quickly become integral to the design, development, and services offered by web sites. Consider a site like Amazon.com that must be able to allow users to quickly jump through a vast virtual warehouse of books and compact disks.

How could Amazon.com create web pages for every single item in their inventory and how could they keep all those pages up to date. Well the answer is that their web pages are created on-the fly by a program that "queries" a database of inventory items and produces an HTML page based on the results of that query.

1.5.1 Flat Databases

A single kind of record with a fixed number of fields.

	A	B	C	D	E	F	G	H
1	Ph #	Ph Name	M #	M Height	M Birth	Deliv	M Wt	B Wt
2	123	Smith	123456	150	4605	7803	60.3	3.4
3	123	Smith	123456	150	4605	8005	62.3	3.7
4	123	Smith	649308	174	5409	7902	58.7	2.9
5	123	Smith	649308	174	5409	8101	57.9	3.1
6	123	Smith	649308	174	5409	8205	55.2	1.4
7	123	Smith	238427	162	5204	7511	61.8	2.5
8	123	Smith	238427	162	5204	7801	64.1	2.7
9	220	Jones	732293	155	5810	7906	59.2	2.2
10	220	Jones	392382	177	4912	7512	57.4	3.6
11	220	Jones	392382	177	4912	7706	58.2	3.4
12							59.51	2.89

Fig 1.3: Flat Database

Notice the repetition of data, and thus an increased chance of errors.

1.5.2 Databases

A database can be defined in various ways, for example:

- A database is a collection of structured data. The structure of the data is independent of any particular application.
- A database is a file of data structured in such a way that it may serve a number of applications, without its structure being dictated by any one of those applications. The concept being that programs are written round the database, rather than files being structured to meet the needs of particular programs.

The centre of any information system is its database, which is a collection of the data resources of an organisation designed to meet the requirements of the company for processing and retrieving information by decision makers. One important use of database is to target more precisely marketing efforts. In USA, the latest trend in management information systems is the executive information system which is used by senior managers. Quality software produces full colour displays on large high quality monitors.

1.5.3 Database: Some Definitions

"A collection of interrelated data stored with controlled redundancy to serve one or more applications in an optimal fashion; the data are stored so that they are independent of the programs, which use the data; a common and controlled approach is used in adding new data and modifying existing data within the database." (Martin)

"A database is a collection of persistent data that is used by the application systems of some given enterprise."

"A database is a shared collection of interrelated data, designed to meet the varied information needs of an organisation. A database has two important properties: It is integrated and it is shared." (McFaddern).

Databases are collections of interrelated data organised according to a schema to serve one or more applications. Database is a sophisticated concept. You may not develop a good grasp of what database is for a few weeks. Many products have been described as database systems that are merely file management systems.

A database implies separation of physical storage from use of the data by an application program to achieve program/data independence. Using a database system, the user or programmer or application specialist need not know the details of how the data are stored and such details are "transparent to the user".

Changes (or updating) can be made to data without affecting other components of the system. These changes include, for example, change of data format or file structure or relocation from one device to another.

Briefly, a database is a collection of data supporting the operations of the organisation. It is however, rather more than that because it must also:

- Be substantially non-redundant.
- Be program independent (data independence).
- Be usable by all the programs.
- Include all the necessary structural interrelations of data and have a common approach to the retrieval, insertion and amendment of data.

Let us consider these requirements individually:

1. **Non-redundancy.**

   As already suggested, this is of benefit in eliminating contradictions and in saving storage space. Occasionally, amount of redundancy is acceptable, such as, when the need for data security or rapid access is paramount.

2. **Data Independence.**

   This means that the data and the programs are independent. That is to say, the data can be moved or restructured without the need to make alterations to the programs. Similarly, an enforced program change does not call for rearrangement of the data layout. This point is important because otherwise a program requiring more data items from a file necessitates rearranging it, and consequently other programs have to be modified to cope with the rearrangement of the file.

3. **Program Usage.**

   A database needs to be usable by not only all the existing applications but also by all foreseeable applications. These are ambitious aims, but nonetheless a database must be open-ended so as to accept new sets of data items and changes to existing data item sets.

4. **Data Inter-relationships.**

   These are necessary owing to the fact that the various applications use data in different ways. One application may demand a link between an employee's name and his pension contribution, another between his tax payment and his previous employer. Requirements such as these impose stringent demands upon the database's accuracy, security and flexibility.

5. **Common Approach.**

   This is in the interests of understanding and simplicity. Although application programmers are not concerned with the database's structure and techniques, a common approach simplifies the database control programs and facilitates the database administrator's work.

Table 1.3 Comparison of File Management Systems with Database Systems

File Management e.g. C++, VB or COBOL program	Database Management eg. Postgres, Oracle
Small systems often PC based relatively cheap	Large systems mini-mainframe relatively expensive
Few 'files'	Many 'files'
Files are files	Files not necessarily files
Simple structure	Complex structure
Little preliminary design	Vast preliminary design
Integrity left to application programmer	Rigorous inbuilt integrity checking
No security	Rigorous security
Simple, primitive backup/recovery	Complex & sophisticated backup/recovery
Often single user	Multiple users

These comparisons tend to be true but exceptions are possible.

Files tend to contain duplication. Therefore, they are susceptible to a loss of INTEGRITY if all files are not updated at the same time. Programs are bound to a file. If a file structure is modified then all programs that access it need to be modified.

Thus, alterations to file structures are difficult and expensive.

Data base systems originated in the late 1950s and early 1960s largely by research and development of IBM Corporation. Most developments were responses to needs of business, military, government and educational institutions, which are complex organisations with complex data and information needs. Trend has been to increase separation between the user and the physical representations of the data, i.e. increasing data "transparency".

1.6 Characteristics of a Database

- It should permit the establishment of a single area for common information that is usable by all authorised users.
- It should allow important information to be recorded on floppy disk, magnetic disk, or mass storage, while secondary records are maintained on magnetic tape or other low-cost storage medium.
- It should guarantee accuracy of updating by an automatic maintenance feature for all segments of the data file.
- It should allow variable-length records in order to conserve space.

- It should provided for expanding or reducing the file, both the number of records in the file and the data elements in each record.
- It should allow for security of files or segments of files.
- There must be a provision for some types of lockout feature so that certain files or areas of files, or even individual records, cannot be accessed during updating.

The important features of a database are:

- Data Independence, an item of data is stored for its own sake and not for one specific use. Thus the use of the data is generalised and is independent of the programs, which use it.
- Data Integrity means the avoidance of conflicting, duplicated data by only showing one copy. If, for example, a stock balance alters, only one update is needed and all programs, which access the stock balance will automatically be given the correct figure.
- Flexibility means that data can be accessed in many different way and for many different purposes. These can range form routine accesses for transaction processes to one-off queries by the Chief Executive.
- The database can grow and change and is built up, stage by stage within the organisation. It will actually comprise several databases, each providing the anticipated information for several logically related management information systems, where the data can be accessed, retrieved and modified with reasonable flexibility.
- The data structures and relationships require highly technical software, known as the Data Base Management System (DBMS) to deal with them. Fortunately, the user is shielded, to a large extent, from the complexity and is able to access the database with the minimum of technical knowledge.
- When the only form of data storage possible was unrelated, unique files for each application, this engendered a narrow, parochial view of information. The reality is that management need information which crosses functions, applications and levels and the flexibility of databases and the linkages possible make the concept a powerful one and essential for end user systems.
- An organisation is not confined to its own internal database. There are companies, which sell access to databases dealing with matters, which an individual organisation would find expensive to collect. Examples of these public databases, Stock Exchange, and financial data, environmental data. Legal matters including EU treaties and legislation, consumer research and marketing data and so on. In 1995, over 8000 external databases were available and the number is continually growing.

In the early days of computerisation, it was normal to maintain specific files for individual applications. Data were processed centrally in batches and there was little or no on-line interrogation of files.

This approach meant that there was duplication of data, inflexibility, concentration on the needs of the computer system rather than the user, and difficulties of accessing files by on-line users. To overcome these problems databases were developed.

1.6.1 Data Independence

We define data independence as the immunity of applications to changes in storage and access strategy, which implies that the applications concerned do not depend on any one particular storage or access strategy. In non database implementations a knowledge of how data is organised and accessed is built into application programs. Lots of examples of this exist in non database applications and such knowledge could be:

- Representation of numeric data
- Representation of character data
- Data coding
- Data materialisation
- Stored record structure
- Stored file structure

All these possible manifestations of data dependence are eliminated when a DBMS is used, because when using a DBMS it is possible for the DBA to change the storage structure or access methods without affecting application programs in any way. Users can add new data, new devices etc.

Data independence is important because:

- Different applications will need to look at data in different ways.
- One must be able to change storage mechanisms and formats without having to modify all application programmes. For example:
  - Method of representation of numeric and alphanumeric data (e.g., changing data format to avoid Y2K problem)
  - Units (e.g., metric vs. furlongs)
  - Coding (e.g., ICD-9 vs. descriptive text)
  - Record structure (e.g. file structure in sequential, sorted, indexed, etc.)

Data Independence is:

1. The ability to modify a scheme definition in one level without affecting a scheme definition in a higher level is called data independence.
2. There are two kinds:
   - Physical data independence.
     - The ability to modify the physical scheme without causing application programs to be rewritten.
   - Modifications at this level are usually to improve performance.
   - Logical data independence.

- The ability to modify the conceptual scheme without causing application programs to be rewritten.
- Usually done when logical structure of database is altered.

3. Logical data independence is harder to achieve, as the application programs are usually heavily dependent on the logical structure of the data. An analogy is made to abstract data types in programming languages.

1.7 Database Management Systems (DBMSs)

This is the interface between the users (application programmers) and the database (the data). A database is a collection of data that represents important objects in a user's business. A Database Management System (DBMS) is a program that allows users to define, manipulate, and process the data in a database, in order to produce meaningful information.

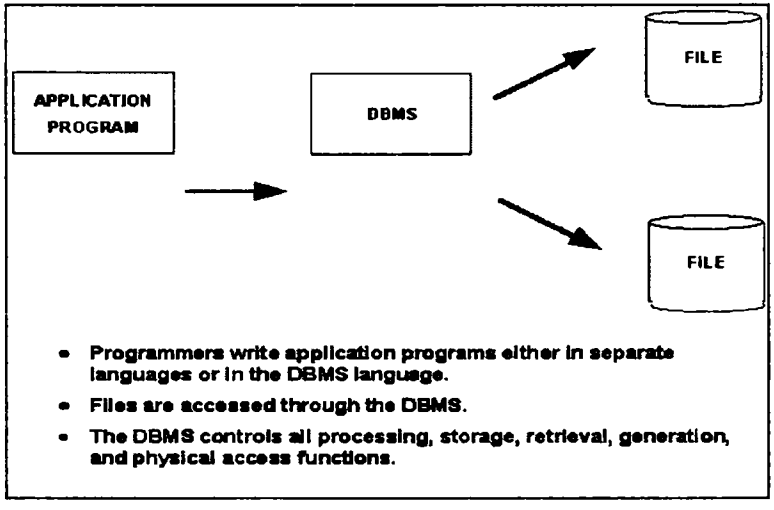


Fig 1.4: DBMS

DBMS is collection of programs that enables you to store, modify, and extract information from a database. There are many different types of DBMSs, ranging from small systems that run on personal computers to huge systems that run on mainframes. The following are examples of database applications:

- Computerised library systems
- Automated teller machines
- Flight reservation systems
- Computerised parts inventory systems

A DBMS is a software package for defining and managing a database.

A 'real' database includes:

- Definitions of:
  - Field names
  - Data formats (text? binary? integer? etc.)
  - Record structures (fixed-length? pointers? field order, etc.)
  - File structures (sequential? indexed? etc.)
- Rules for validating and manipulating data
- Data

A Database Management System, or DBMS, gives the user access to their data and helps them transform the data into information. Such Database Management Systems include dBase, Paradox, IMS, and Oracle. These systems allow users to create, update, and extract information from their databases. Compared to a manual filing system, the biggest advantages to a computerised database system are speed, accuracy, and accessibility.

DBMS is an intermediatory between programs and the data. Programs access the DBMS, which then accesses the data . Application programs are independent of the file structure i.e. changes to the file structure do not require changes to application programs. Changes to programs do not need a new file structure. DBMS uses many cross-references between files to avoid REDUNDANCY (duplication)

Databases are made possible through the use of special software called Database Management Systems (DBMSs.) A DBMS uses a Data Definition Language (DDL) to define a database and gives programmers and/or end-users a Data Manipulation Language (DML) to navigate, use, and maintain the database. Some DMLs must be used in conjunction with a host language like COBOL. The result is to provide programmers and end-users with tools to structure, maintain, and use their data with greater ease and flexibility.

The database umbrella covers many areas, including relational design, data semantics, DBMS software, and physical file design. In studying each area individually, one might not grasp the relationships between them. However, it is important to understand how the areas are related in order to provide information and applications that fulfill user needs. After all, the primary purpose of a DBMS is to provide the user with a workable, meaningful, and relevant system.

From a technical standpoint, DBMSs can differ widely. The terms relational, network, flat, and hierarchical all refer to the way a DBMS organises information internally. The internal organisation can affect how quickly and flexibly you can extract information.

Requests for information from a database are made in the form of a query, which is a stylised question. For example, the query

```
SELECT ALL WHERE NAME = "Vijay" AND AGE > 35
```

requests all records in which the NAME field is SMITH and the AGE field is greater than 35. The set of rules for constructing queries is known as a query language. Different DBMSs support different query languages, although there is a semi-standardised query language called SQL (Structured Query Language). Sophisticated languages for managing database systems are called Fourth-Generation Languages, or 4GLs for short.

The information from a database can be presented in a variety of formats. Most DBMSs include a report writer program that enables you to output data in the form of a report. Many DBMSs also include a graphics component that enables you to output information in the form of graphs and charts.

DBMSs come in many shapes and sizes. For a few hundred dollars you can purchase a DBMS for your desktop computer. For larger computer systems, much more expensive DBMSs are required. Many mainframe-based DBMSs are leased by organisations. DBMS of this scale are highly sophisticated and would be extremely expensive to develop from scratch. Therefore, it is cheaper for an organisation to lease such a DBMS program than to develop it. Since, there are a variety of DBMSs available, you should know some of the basic features, as well as strengths and weaknesses, of the major types.

Database Management Systems are applications that were developed to create, manage, and use data and to deal with the problems of file processing systems. The data is stored as records in various database files that can be combined to produce meaningful information for users. The DBMS controls all functions of capturing, processing, storing, and retrieving data from databases and generates various forms of data output. The application programs are written either in a separate language or in the DBMS language, and the DBMS can contain hundreds of applications and files. Modeling business data, as opposed to business processes, allows the definition of data objects that are important to the business. These data objects are more stable and less likely to change than business processes. Because the representation of the data is separate from the physical implementation and access functions, the relationships between the data files is more apparent. Therefore, DBMS have more flexibility than file processing systems and require less programming maintenance. This allows programmers to focus more on information representation than on physical aspects of data management.

1.7.1 Purpose of Database Management Systems

Functions of a DBMS:

- To store data
- To organise data
- To control access to data
- To protect data

Uses of a DBMS:

- To provide decision support;
- Managers and analysts retrieve information generated by the DBMS for inquiry, analysis, and decision-making.
- To provide transaction processing; and
- Users input, update, and process data with transactions that generate information needed by managers and other users or by other departments.

In detail, following are the functions:

1. To see why database management systems are necessary, let's look at a typical "file-processing system" supported by a conventional operating system.

The application is a savings bank:

- Savings account and customer records are kept in permanent system files.
- Application programs are written to manipulate files to perform the following tasks:
  - Debit or credit an account.
  - Add a new account;
  - Find an account balance; and
  - Generate monthly statements.

2. Development of the system proceeds as follows:
   - New application programs must be written as the need arises;
   - New permanent files are created as required;
   - But over a long period of time files may be in different formats; and
   - Application programs may be in different languages.
3. So we can see there are problems with the straight file processing approach:
   - Data redundancy and inconsistency;
   - Same information may be duplicated in several places;
   - All copies may not be updated properly.
   - Difficulty in accessing data:
     - May have to write a new application program to satisfy an unusual request.
     - E.g. find all customers with the same postal code.
     - Could generate this data manually, but a long job...
   - Data isolation.
   - Data in different files.
   - Data in different formats.
   - Difficult to write new application programs.
   - Multiple users:
     - Want concurrency for faster response time.
     - Need protection for concurrent updates.
     - E.g. two customers withdrawing funds from the same account at the same time. Account has $500, in it, and they withdraw $100 and $50. The result could be $350, $400 or $450 if no protection.
   - Security problems:
     - Every user of the system should be able to access only the data they are permitted to see.
     - E.g. payroll people only handle employee records, and cannot see customer accounts; tellers only access account data and cannot see payroll data.
     - Difficult to enforce this with application programs.

- Integrity problems:
  - Data may be required to satisfy constraints.
  - E.g. no account balance below $25.00.
  - Again, difficult to enforce or to change constraints with the file-processing approach.

These problems and others led to the development of Database Management Systems.

1.7.2 Requirements of a Well-Designed DBMS

- Data Integrity

1. Eliminate or/reduce Redundancy:
   - Separate parts of the database must correlate to reduce data redundancy and provide efficient data updates. Duplication of data should at best be eliminated. If not the DBMS should be aware of it, and control it.
2. Maintain Consistency:
   - If there is duplication of data, e.g. a person's telephone number may be stored in two tables (it shouldn't), then a change will automatically be reflected in both tables.
3. Data Sharing:
   - A single copy of the data can be used by many users.
4. Standards can be enforced:
   - Both in presentation and storage
   - Useful to aid data interchange.
5. Provide Security of the Data:
   - Restrict access of data to authorised users
   - Permissions
   - Passwords
   - Log transactions
   - Provide backup/recovery facilities
6. Maintain Integrity:
   - Ensure the data in the DB is correct. e.g. range checks
   - Deletion check
7. Data Independence:
   - Different applications need different views of the same data
   - Storage structure can change without having to alter existing applications (programs) e.g. new fields added
8. Reduced program maintenance.
9. Interface with a Data Dictionary facility - METADATA:
   - Provides information about the database
   - Schemes and users
10. Provide a Query Language to access the data

- SQL is the standard
- Ingres has SQL, QUEL and QBF
- Embedded in at least one language (eg. C, C++, C# ...)

11. Avoid possible deadlock situations in on-line DB's
12. Downloading facility to a PC?
13. Allow a program on PC to access the data?
14. Sensitive data must be protected.
15. Concurrency control.
16. Ability to lock records.
17. Transaction processing
18. An individual transaction either is completed or not completed.
19. Recovery functions.
20. Efficient data recovery after system or power failures.

1.7.3 Characteristics of DBMS

Typically, a data base management system (based upon controlled activities) has the following characteristics:

- It is a computerized record-keeping system.
- It contains facilities that allow the user to . . .
  - Add and delete files to the system.
  - Insert, retrieve, update, and delete data in existing files.
  - It is collection of databases.
- A DBMS may contain many databases that can be used for separate purposes or combined to provide useful information.
- Operating environment independence-capability to run on many computers with certain types of operating systems.
- User oriented provision for an English-like language which enables the user to consider logical entities in place of physical entities, such as hardware and system software.
- Data base independence from application programs-use of a data base language and "call" commands from the application program.
- Security featuresprovision for controlling data base access and control over the database elements.
- DBMS Overcomes Limitations.
- Eliminates separation and isolation of data.
- In a DBMS, all data is stored in the database. The DBMS allows for complex relationships between data files, providing efficient data integration. Programmers specify, how the data is to be combined; the DBMS performs the functions to provide the information.
- DBMS Reduces Data Redundancy.

Minimal duplication of data provides more storage space in the system and allows for efficient updates, when changes in records occur, maintaining data integrity. For example, if a customer's address is stored in only one file and a change is made to that file, all affected applications are updated automatically.

- Eliminates Dependence between Programs and Data.

In a DBMS record formats are defined in the database and accessed by the DBMS. Application programs only define, which data items are needed, not data formats. Therefore, format changes are made to the database by the DBMS, and, since most fields are stored in only one file, the changes are made only to the affected file. Also, because structure formats and access methods are independent of the database, there is little impact on application programs if structure/access changes are made.

Allows for representation of data from the user's view.

Relationships between data objects in a user's environment are stored in the DBMS. Data elements from any number of files can be combined to create useful forms, reports, and other applications.

- Increases Data Flexibility.

Because of the data-independent structure of DBMS, applications can be created without intensive programming. Thus, ad hoc requests for information and applications can be filled with less time and labor from programmers, than with file processing systems.

1.7.4 Components of a DBMS

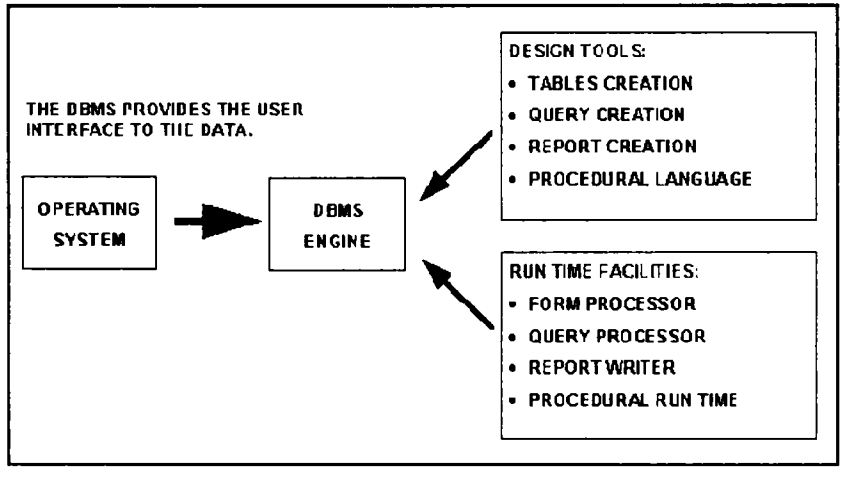


Fig 1.6: Components of a DBMS

The basic components of a DBMS can be divided into three subsystems:

1. Design Tools:

Provide features for creating the database and various applications, forms, and reports.

2. Run-time Facilities:

Process the applications created by the design tools.

3. DBMS Engine:

Translates between the design tools and run-time facilities and the data.

Although they are all excellent advantages to DBMS users the initial investment of system personnel time, as well as software and equipment cost can be high; however, increasing experience with Database Management Systems is reducing these costs. Another disadvantage of DBMS is that incorrect transaction data tend to precipitate additional problems and errors throughout the system,. Therefore, various tests should be incorporated to catch errors entering the database before they are stored. Overall, these disadvantages of DBMS are capable of being overcome to a great degree, if the management information system is properly designed and managed.

1.7.5 Database Architecture

Let's briefly examine how the database environment is implemented. The discussion is simplified but nonetheless adequate for an introductory level treatment.

In a database environment, the notion of conventional files disappears. In its place, we have a structure, where records of one type, such as ORDERs, can be related to records of different types, such as CUSTOMERs. If this is all precisely correct! Database physically implements logical entities and relationships.

Figure 1.7 depicts the database technical environment. A systems analyst, or database analyst, designs the structure of the data in terms of record types, fields contained in those record types, and relationships that exist between record types (once again, this should sound like data modeling). The analyst then uses the DBMS's Data Definition Language (DDL) to physically establish those record types, fields, and relationships. Additionally, the DDL defines views of the database. Views restrict the portion of a database that may be in a permanent data dictionary. Some data dictionaries include formal, elaborate software that helps database specialist keep track of everything stored and inquiry responses (very similar to our CASE concept).

Computer programs are then written to load, maintain, and use actual data. These programs may be written in a host programming language, such as COBOL, PL/1, or BASIC that is supported by the DBMS. Using the host language, the programs call subroutines in the DBMS's Data Manipulation Language (DML) to retrieve, create delete, and modify records, and navigate between record types, for example, from CUSTOMER to ORDERs for that customer. The programmers don't have to understand how the data is physically stored (file organisation) or accessed. The DBMS takes care of such details. The DML refers to the data dictionary (DDL) during execution.

Alternatively, many DBMSs don't require a host programming language. They provide their own self-contained programming language that includes a DDL and a DML. Generally speaking, these self-contained languages greatly simplify applications prototyping and development. These languages and features are typically designed to be simple to learn and use, so much so that experienced programmers can be replaced by analysts and end-users.

Many main-frame DBMSs, greatly simplify internal controls by automatically logging all updates and enforcing security as defined by the database analyst. Eventually, this should even be true of microcomputer DBMSs.

Multiple-user DBMSs frequently include a teleprocessing or TP monitor. This is specialised software that supervises and controls access to the database via terminals in online environments. Most such systems can also interface with TP monitors other than their own, such as IBM's CICS.

A DBMS allows us to physically implement a logical data model. The physical data model is usually called a schema. Any given DBMS support two schemas, a logical schema (which, unfortunately, is semiphysical as constrained by the DBMS itself) and a physical schema (which truly describes the physical storage of the data).

The physical schema defines records, files access methods, file organisations, indices, blocking, pointers, and other physical attributes. DBMS don't replace file structure, they just hide them from the programmers and end-users. This aspect of the database is not of concern to most system analysts.

The logical (or semiphysical) schema, defines the database in simple terms, as seen by end-users and programmers. It defines records and associations, just like the entity relationship data models. However, the logical schema is constrained by one of three popular DBMS structures: hierarchical, network or linked, and relational. As previously mentioned, different programs or users may be restricted to different views of the database. The views are sometimes called subschemas.

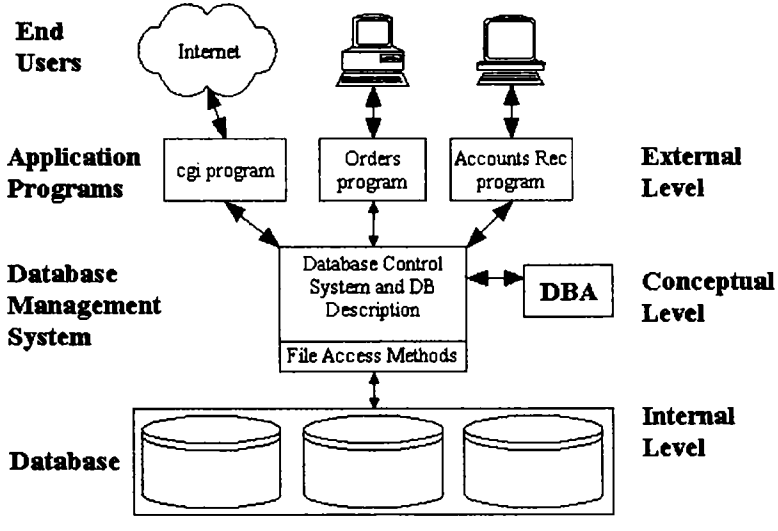


Fig. 1.7: Architecture of an Ideal DBMS

Not all Database Management Systems contain all of the components shown in this diagram; however, the diagram is representative of a typical DBMS. The key components are the Data Definition Language and the Data Manipulation Language, the common components of all DBMSs.

1.7.6 Elements of Databse Management Systems

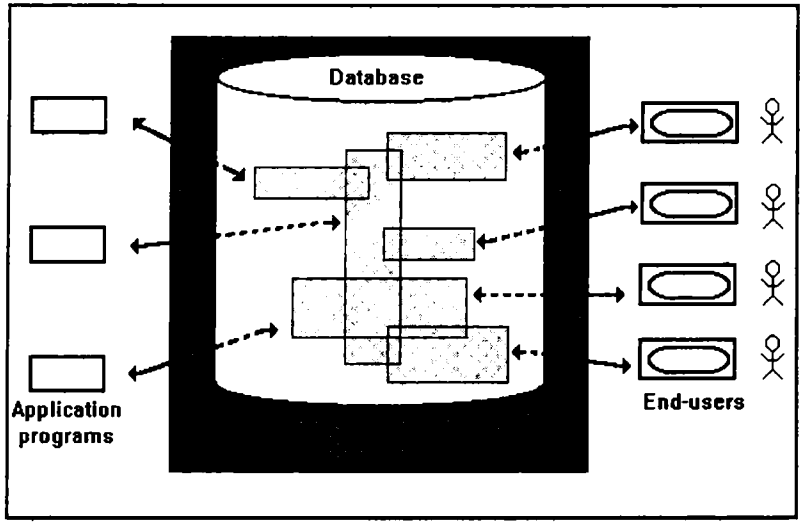


Fig 1.8: Simplified picture of database system

A DBMS is a computerised record-keeping system that stores, maintains and provides access to information. A database system involves four major components:

- Data
- Hardware
- Software
- Users

DBMS are used by any reasonably self-contained commercial, scientific, technical or other organisation from a single individual to a large company and a DBMS may be used for many reasons. Data itself consists of individual entities, in addition to which, there will be relationships between entity types linking them together. Given an enterprise with a nebulously defined collection of data, the mapping of this collection onto the real DBMS is done based on a data model. Various architectures exists for databases and various models have been proposed including the relational, network, and hierarchic models. Other important components are:

- **Data:** Data stored in a database include numerical data, which may be integers (whole numbers only) or floating point numbers (decimal), and non-numerical data such as characters (alphabetic and numeric characters), date or logical (true or false). More advanced systems may include more complicated data entities such as pictures and images as data types.

- **Standard Operations:** Standard operations are provided by most DBMS. These operations provide the user basic capabilities for data manipulation. Examples of these standard operations are sorting, deleting and selecting records.
- **Data Definition Language (DDL):** DDL is the language used to describe the contents of the database. It is used to describe, for example, attribute names (field names), data types, location in the data base, etc.
- **Data Manipulation and Query Language:** Normally a Fourth-Generation Language (4GL) is supported by a DBMS to form commands for input, edit, analysis, output, reformatting, etc. Some degree of standardisation has been achieved with SQL (Structured Query Language).
- **Programming Tools:** Besides commands and queries, the database should be accessible directly from application programs through function calls (subroutine calls) in conventional programming languages.
- File Structures: Every DBMS has its own internal structures used to organise the data although some common data models are used by most DBMS.

1.7.7 Properties of DBMS Data

DBMS are available on any machine, from small micros to large mainframes, and can be single or multi-user - obviously, there will be special problems in multi-user environments in order to make other users invisible, but these problems are internal to DBMS.

1. Data may be shared over many databases, giving a distributed DBMS, though quite often it is centralised and stored in just one database on one machine. In general, the data in the database, at least in a large system, will be both integrated and shared.
2. The goal of a DBMS is to provide an environment that is both convenient and efficient to use in:
   - Retrieving information from the database.
   - Storing information into the database.
3. Databases are usually designed to manage large bodies of information. This involves:
   - Definition of structures for information storage (data modeling).
   - Provision of mechanisms for the manipulation of information (file and systems structure, query processing).
   - Providing for the safety of information in the database (crash recovery and security).
   - Concurrency control if the system is shared by users.

1.7.8 Advantages and Disadvantages of DBMS

Because the new management information system design will include a Database Management System, the advantages of this arrangement should be carefully considered. First, DBMS is not only effective for generating and maintaining a wide variety of routine management and operating reports, but also adaptable to meeting the new and emerging requirements of management to answer a myriad of "What if?" questions. The latter capability means Database Management Systems will be important aids to manager seeking to explore and understand new

relationships among various data elements. Second, data elements can be structured in a manner more suitable to their application, allowing their retrieval with a minimum of effort. Third, DBMS keeps redundancy of data elements to a minimum. Since one data file serves many users, as a result of this "single record" concept, a transaction is entered once for all users. The DBMS also allows two or more files to be updated with the entry of a single transaction. Fourth, application programs are independent of the changes in the database, so that their maintenance is kept to a bare minimum. Fifty, DBMS provides data protection not only for accessing one database record at a time, but also for preventing database access by unauthorized personnel.

The main advantage of using a DBMS is that the formalism of the model of data underlying the DBMS is imposed upon the data set to yield a logical and structured organisation of the data. Given a fuzzy, real-world data set, when a model's formalism is imposed on that data set the result is easier to manage, define and manipulate.

Different models of data lead to different organisations. In general, the relational model is the most popular because that model is the most abstract and easiest to apply to data, while still being powerful.

- Advantages of Using DBMS

To summarise using a DBMS we have the following advantages :

- Clear picture of logical organisation of data set.
- Centralisation for multi-users.
- Data independence.
- Monitoring database performance.
- Centralised data reduces management problems.
- Data redundancy and consistency are controllable.
- Program-data interdependency is diminished.
- Flexibility of data is increased.
- Reduction in data redundancy.
- Maintenance of data integrity and quality.
- Data are self-documented or self-descriptive.
- Avoidance of inconsistency.
- Reduced cost of software development.
- Security restrictions.
- Application programs are independent of structure of DB.
- Application programs share the same data.
- New programs are easier and cost less to implement.

1.8 Data Base Administrator (DBA)

DBA are the person(s) responsible for overall control of the total system. The DBA is responsible for organising the system so as to get the performance that is "best for the system" and for making

the appropriate adjustments as user requirements change. A DBA can be someone who, from the start, has concentrated in the area of database design and administration. A DBA can be a programmer who, by default or by volunteering, took over the responsibility of maintaining a SQL Server during project development and enjoyed the job so much that he switched. A DBA can be a system administrator who was given the added responsibility of maintaining a SQL Server. DBAs can even come from unrelated fields, such as accounting or the help desk, and switch to Information Systems to become DBAs.

The role of the database administrator is very important in an organisation. Learn about some of the responsibilities of a this position. In a very general sense, a database administrator is the individual responsible for maintaining the RDBMS system. In this article, we will focus on Microsoft SQL Server; however, the concepts apply to virtually any database.

The DBA has many different responsibilities, but the overall goal of the DBA is to keep the server up at all times and to provide users with access to the required information, when they need it. The DBA makes sure that the database is protected and that any chance of data loss is minimized.

To help the DBA in this task, certain database utilities and tools may be used.

The responsibilities that an administrater has to discharge include:

- Defining the conceptual schema.
- Defining the internal schema.
- Liaising with users, mainly application programmers.
- Includes defining external views for external users for access to the DB.
- Defining security and integrity rules.
- Defining backup and recovery procedures.
- Monitoring and fine-tuning performance.
- Scheme definition: The creation of the original database scheme. This involves writing a set of definitions in a DDL (data storage and definition language), compiled by the DDL compiler into a set of tables stored in the data dictionary.
- Storage structure and access method definition: Writing a set of definitions translated by the DDL compiler.
- Scheme and physical organisation modification: Writing a set of definitions used by the DDL compiler to generate modifications to appropriate internal system tables (e.g. data dictionary). This is done rarely, but sometimes the database scheme or physical organisation must be modified.
- Granting of authorization for data access: Granting different types of authorisation for data access to various users.
- Integrity constraint specification: Generating integrity constraints. These are consulted by the database manager module, whenever updated.
- Enforcing standards.

The DBA must ensure that all applicable standards are observed in the representation of the data in the DBMS, where applicable standards may include any or all of corporate, installation, departmental, industry, national and international standards. Standardising is essential, when many people use the same data and is particularly desirable as an aid to data interchange between systems, but this is only possible, when data independence has been achieved.

Database Administrator is also a name of the program (software). A database administrator (DBA) controls and manages the database.

1.8.1 Other DBA Responsibilities

The following sections examine the responsibilities of the database administrator and how they translate to various Microsoft SQL Server tasks.

- Installing and Upgrading an SQL Server:

The DBA is responsible for installing SQL Server or upgrading an existing SQL Server. In the case of upgrading SQL Server, the DBA is responsible for ensuring that if the upgrade is not successful, the SQL Server can be rolled back to an earlier release until the upgrade issues can be resolved. The DBA is also responsible for applying SQL Server service packs. A service pack is not a true upgrade, but an installation of the current version of software with various bug fixes and patches that have been resolved since the product's release.

- Monitoring the database server's health and tuning accordingly.

Monitoring the health of the database server means making sure that the following is done:

- The server is running with optimal performance.
- The error log or event log is monitored for database errors.

Databases have routine maintenance performed on them, and the overall system has periodic maintenance performed by the system administrator.

- Using Storage Properly:

SQL Server 2000 enables you to automatically grow the size of your databases and transaction logs, or you can choose to select a fixed size for the database and transaction log. Either way, maintaining the proper use of storage means monitoring space requirements and adding new storage space (disk drives), when required.

- Performing Backup and Recovery Duties:

Backup and recovery are the DBA's most critical tasks; they include the following aspects:

- Establishing standards and schedules for database backups.
- Developing recovery procedures for each database.
- Making sure that the backup schedules meet the recovery requirements.
- Managing Database Users and Security:

With SQL Server 2000, the DBA works tightly with the Windows NT administrator to add user NT logins to the database. In non-NT domains, the DBA adds user logins. The DBA is also responsible for assigning users to databases and determining the proper security level for each user. Within each database, the DBA is responsible for assigning permissions to the various database objects such as tables, views, and stored procedures.

- Working with Developers:

It is important for the DBA to work closely with development teams to assist in overall database design, such as creating normalised databases, helping developers tune queries, assigning proper indexes, and aiding developers in the creation of triggers and stored procedures. In the SQL Server 2000 environment, a good DBA will show the developers how to use and take advantage of the SQL Server Index Tuning Wizard and the SQL Server profiler.

- Establishing and Enforcing Standards:

The DBA should establish naming conventions and standards for the SQL Server and databases and make sure that everyone sticks to them.

- Transferring Data:

The DBA is responsible for importing and exporting data to and from the SQL Server. In the current trend to downsize and combine client/server systems with mainframe systems and Web technologies to create Enterprise systems, importing data from the mainframe to SQL Server is a common occurrence that is about to become more common with the SQL Server 2000 Data Transformation Services. Good DTS DBAs will be in hot demand as companies struggle to move and translate legacy system to Enterprise systems.

- Replicating Data:

SQL Server version 2000 has many different replication capabilities such as Merge replication (2-way disconnected replication) and queued replication. Managing and setting up replication topologies is a big undertaking for a DBA because of the complexities involved with properly setting up and maintaining replication.

- Data Warehousing:

SQL Server 2000 has substantial data warehousing capabilities that require the DBA to learn an additional product (the Microsoft OLAP Server) and architecture. Data warehousing provides new and interesting challenges to the DBA and in some companies a new career as a warehouse specialist.

- Scheduling Events:

The database administrator is responsible for setting up and scheduling various events using Windows NT and SQL Server to aid in performing many tasks such as backups and replication.

- Providing 24-Hour Access:

The database server must stay up, and the databases must always be protected and online. Be prepared to perform some maintenance and upgrades after hours. Also be prepared to carry that dreaded beeper. If the database server should go down, be ready to get the server up and running. After all, that's your job.

- Learning Constantly:

To be a good DBA, you must continue to study and practice your mission-critical procedures, such as testing your backups by recovering to a test database. In this business, technology changes very fast, so you must continue learning about SQL Server, available client/servers, and database design tools. It is a never-ending process.

1.8.2 Required Skills

- Good understanding of the Oracle database, related utilities and tools.
- A good understanding of the underlying operating system.
- A good knowledge of the physical database design.
- Ability to perform both Oracle and operating system performance tuning and monitoring.
- Knowledge of ALL Oracle backup and recovery scenarios.
- A good knowledge of Oracle security management.
- A good knowledge of how Oracle acquires and manages resources.
- A good knowledge Oracle data integrity.
- Sound knowledge of the implemented application systems.
- Experience in code migration, database change management and data management through the various stages of the development life cycle.
- A sound knowledge of both database and system performance tuning.
- A DBA should have sound communication skills with management, development teams, vendors and systems administrators.
- Provide a strategic database direction for the organisation.
- A DBA should have the ability to handle multiple projects and deadlines.
- A DBA should possess a sound understanding of the business.

1.8.3 Qualifications

- Must be certified as an Oracle/MS SQL/SYBASE/ DBII DBA
- Preferably a BS in computer science or related engineering field
- Lots and lots of EXPERIENCE

1.8.4 Database Manager

1. The database manager is a program module, which provides the interface between the low-level data stored in the database and the application programs and queries submitted to the system.
2. Databases typically require lots of storage space (gigabytes). This must be stored on disks. Data is moved between disk and main memory (MM) as needed.
3. The goal of the database system is to simplify and facilitate access to data. Performance is important. Views provide simplification.
4. So the database manager module is responsible for:
   - Interaction with the file manager: Storing raw data on disk using the file system usually provided by a conventional operating system. The database manager must translate DML statements into low-level file system commands (for storing, retrieving and updating data in the database).
   - Integrity enforcement: Checking that updates in the database do not violate consistency constraints (e.g. no bank account balance below Rs. 25).

- Security enforcement: Ensuring that users only have access to information they are permitted to see.
- Backup and recovery: Detecting failures due to power failure, disk crash, software errors, etc., and restoring the database to its state before the failure Concurrency control: Preserving data consistency, when there are concurrent users.

5. Some small database systems may miss some of these features, resulting in simpler database managers. (For example, no concurrency is required on a PC running MS-DOS.) These features are necessary on larger systems.

1.8.5 Monitoring Database Performance

The DBA is responsible for organising the system so as to get the performance that is "best for the system" and for making the appropriate adjustments as user requirements change. To help the DBA in this task, certain database utilities and tools may be used.

1.8.6 Database Utilities and Tools

The DBA will need many utilities to implement proper control over the database. The following are some examples.

- Load routines - to create the initial databse.
- Dump/restore routines - for backup and recovery purposes.
- Reorganisation routines - to rearrange the data for various performance reasons.
- Statistical routines - to compute performance statistics etc.
- Analysis routines - to analyse the statistics.

These tools allow the DBA to monitor the performance of the DBMS.

1.9 Types of Data Base Systems

A modern DBMS uses one of the following data models for constructing its structure:

- Flat file (tabular) - data in a single table (no link between tables).
- Relational Databases.
- Networked Databases.
- Hierarchical Databases.
- Analytic Databases.
- Operational Databases.
- Client Server Databases.
- Object Oriented Databases.

1.9.1 Analytic Databases

Analytic databases (a.k.a. OLAP which stand for On Line Analytical Processing) are primarily static, read-only databases, which store archived, historical data used for analysis. For example, a company might store sales records over the last ten years in an analytic database and use that database to analyse marketing strategies in relationship to demographics.

On the web, you will often see analytic databases in the form of inventory catalogs, such as the one shown previously from Amazon.com. An inventory catalog analytical database usually holds descriptive information about all available products in the inventory.

Web pages are generated dynamically by querying the list of available products in the inventory against some search parameters. The dynamically-generated page will display the information about each item (such as title, author, ISBN), which is stored in the database.

1.9.2 Operational Databases

Operational databases (a.k.a. OLTP On Line Transaction Processing), on the other hand, are used to manage more dynamic bits of data. These types of databases allow you to do more than simply view archived data. Operational databases allow you to modify that data (add, change or delete data).

These types of databases are usually used to track real-time information. For example, a company might have an operational database used to track warehouse/stock quantities. As customers order products from an online web store, an operational database can be used to keep track of how many items have been sold and when the company will need to reord stock.

1.9.3 Object-oriented Databases

OODBMS are able to handle many new data types, including graphics, photographs, audio, and video, object-oriented databases represent a significant advance over its other database cousins. Hierarchical and network databases are all designed to handle structured data, that is, data that fits nicely into fields, rows, and columns. They are useful for handling small snippets of information such as names, addresses, zip codes, product numbers, and any kind of statistic or number you can think of. On the other hand, an object-oriented database can be used to store data from a variety of media sources, such as photographs and text, and produce work, as output, in a multimedia format.

1.10 Data Dictionary

Data Dictionary is a file that defines the basic organisation of a database. A data dictionary contains a list of all files in the database, the number of records in each file, and the names and types of each field. Most database management systems keep the data dictionary hidden from users to prevent them from accidentally destroying its contents.

Data dictionaries do not contain any actual data from the database, only bookkeeping information for managing it. Without a data dictionary, however, a database management system cannot access data from the database.

A data dictionary describes tables, columns, indexes, and other elements of the database structure. Other terms for data dictionary include data directory or metadata. Following is an example of a portion of the data dictionary for the entity COURSE.

Object Model: Text for Entities

Entity: COURSE Component: Entity

Description: A course of study

Examples: INSY312 Database Design
INSY410 Systems Analysis & Design
INSY115 Introduction to Computers

Business Policies:

* A course may be taught more than once in a semester.
* Each class of a specific course held in the same semester will be differentiated by an extension number.
* A course may be taught by more than one professor.
* Each class is held in only one room per semester.

Attributes: Course name
Course number
Course description

The data dictionary stores descriptions of data items and structures, as well as systems processes. It is intended to be used to understand the system by analysts, who retrieve the details and descriptions it stores, and during systems design, when information about such concerns as data length, alternative names (aliases), and data used in specific processes must be available. The data dictionary also stores validation information to guide the analysts in specifying controls for the system's acceptance of data.

Dictionary systems are important for five reasons:

1. to manage the detail in large systems;
2. to communicate a common meaning for all system elements;
3. to document the feature of the system;
4. to facilitate analysis of the details in order to evaluate characteristics and determine where systems changes should be made; and
5. to locate errors and omissions in the system. During analysis, particular attention is paid to understanding the nature of the transactions and inquiries made of the system and the requirements for output and report generation. these matters are significant in determining file and database requirements and system capacity needs.

The dictionary also contains definitions of data flows, data stores, and processes. The latter includes a summary of processing logic.

Data dictionaries can be developed manually or using automated systems. Automated systems offer the advantage of automatically producing data element, data structure, and process listings. They also perform cross-reference checking and error detection, important advantages when working on large systems that must be correct. Automated dictionary systems are becoming the norm in the development of computer information systems.

1. Data Elements

Data elements, the most fundamental data level recorded in a dictionary, are the building blocks for all other data in the system. A set of data items, termed a data structure, describes the relation between individual elements. Data are arranged according to one of four relationships:

sequence, selection (either/or), iteration (repetition), and optional relationships. Regardless of the specific relationship, each data item must be described fully, including the name, as well as specification of the length and the type.

A data dictionary (DD) is intended to provide a complete documentation of all the elements of a data Flow Diagram, namely data items, data stores, and data flows. Data described in a DD carries the following details.

Data characteristics	Characteristics of each data type
Data type	Data item/data store/data flow
Data name	Name of data item/data store/data flow
Data aliases	Alternate names used for convenience by multiple suers
Data description	A short description of data, explained in simple terms

A data item is characterised by its type (numeric/alphanumeric), width, etc.

A data store is characterised by its composition (set of data items), organisation (sequential, random), etc.

A data flow is characterised by its origin, destination, etc.

Below, we describe the contents of a Dd for a sample data item, data item, data store and data flow for a CBIS on payroll accounting.

(a) DATA ITEM

* Data type	Data item
* Data name	G-SALARY
* Data aliases	Wages
* Data description	Monthly gross salary of an employee

(b) Data characteristics

Type	Numerical
Width	7.2
	4 digits for the rupee component
	1 digit for the decimal point
	2 digits for the paise component
Associated data stores	Payroll file
	Personnel file
Associated data processes	Payroll file PF accounting
	Personnel information systems
	Comments Gross salary is based on employees designation
	and hence falls within a specified range

(c) Data Store

* Data type	Data store
* Data name	Payroll file
* Data aliases	Salary file
* Data description	Master file of employees for payroll accounting

(d) Data characteristics

Composition	EMP-NAME Designation B-Salary Department : : G-Salary : :
Organisation	Sequential file
Volume	1000 records (approximately)
Size	350 K bytes (approximately) per record
Associated data	Payroll accounting
Processes	PF accounting
Inbound data flow	—
Outbound data flow	—
Comments	This file gets updated every month at the time of pay roll processing. On an average, about 5 records are deleted per month (retiring/ leaving the organisation) and about 10 records are added per month (new appointments).

(e) Data Flow

* Data type	Data flow
* Data name	DATA ON EMPLOYEES
* Data aliases	—
* Data description	Data on employees required for payroll processing

Data characteristics

Origin	Accounts office
Destination	Process 1 in payroll accounting
Contents	EMP-NAME
	Designation
	B-Salary
	Department
Associated data processes	Payroll accounting
Associated data stores commands	

1.10.1 Advantages of Data Dictionary

Analysts use data dictionaries for five important reasons:

1. To manage the detail in large systems
2. To communicate a common meaning for all system elements
3. To document the features of the system
4. To facilitate analysis of the details in order to evaluate characteristics and determine where system changes should be made
5. To locate errors and omissions in the system

The data dictionary may be viewed as aiding most analysis and design activities. The BCS Working Party report suggested that it could be used during each of the following stages:

- Data analysis, to determine the fundamental structure of the data of the enterprise;
- Functional analysis, to determine the way in which event and functions use data;
- Database or conventional file design:
- Transaction or program design;
- Storage structure design, where this is a further refinement of the initial database or file design;
- Operational running of application systems;
- Collection and evaluation of performance statistics;
- Database tuning to improve performance; and
- Application maintenance and database restructuring.

The first type of data dictionary was a clerical card index system keeping track of the physical implementation. Information held on files and records used by programs made it easier to identify programs which needed recompiling in the event of a change in record or file structure (Redfearn, 1987). The advent of Data Base Management Systems (DBMS) with their emphasis on shared data reinforced this need for documentation. A data dictionary provided a means for ensuring standard terminology and effective documentation control through cross-referencing. The automation of this dictionary was a natural step.

Considerable advantages accrue from using an automated Data Dictionary System (DDs). This DDS software (usually referred to as data dictionary in this text) can effectively and efficiently maintain a large central repository of data about the data of an organisation-so-called metadata is a level of abstraction higher than the actual data used in operations; it is not the actual data used but data about the actual data used.

Fig. 1.9 shows the distinction between the two levels of data. Metadata is used to define, identify and describe the characteristics of the user data. Metadata usually falls into two categories.

- What the data is or what it means.
- Where the data can be found and how it can be accessed.

Metadata	Metadata	User data	Entities
Database name:	CUSTDB	DATABASE	The data values in the customer database
Database Size:	10Kbytes		
Record name:	CUSTREL	RECORD	The data values in the customer records
Record size:	140 bytes		
Data element name:	CUSTID	DATA	The data values in the CUSTID data
Data element size:	8 bytes	ELEMENT	Element such as 78534021

Fig. 1.9 Metadata Versus User Data

There has recently been an increased acceptance of the benefits of establishing and maintaining a data dictionary and a number of products are available in the commercial market-place. A comparative review of 13 data dictionary systems was made by Mayne. 1984) revealed that it increases the efficiency to enormous level. The central role of data dictionaries in commercial software and data processing organisation and standards demands that the concept be examined in greater depth.

1.10.2 Automated Data Dictionary

The first automated data dictionary systems were primarily concerned with documenting the physical aspects of data processing-systems, programs, files and data bases. However, their scope has now increased to storing and manipulating logical models-such as data flow and entity-relationship diagrams. In doing so the data dictionary has progressed from a passive documenter of systems to an active productivity tool.

The activity of a data dictionary system demands scrutiny. All data processing applications require metadata to operate (such as database schemes, Cobol File Definitions and Job Control Lan- guages) and an active data dictionary controls this processing environment. Indeed it is the scope of the metadata management that determines the activity of the data dictionary because the DDS is only active "with respect to a program or process if and only if that program or process is fully dependent upon the data dictionary for its metadata" (Plagman, 1978) In

a passive system metadata may be defined from other sources and the data dictionary acts as a documentation facility rather than an active tool in system development. In general, commercial DDS may be placed upon a spectrum of activity with the trend towards active or potentially active data dictionary systems.

This concept of activity is an example of how the term 'data dictionary' is used by vendors to describe software with a wide range of capabilities and facilities. Comparisons across products reveal marked differences in functionality. Consequently, the next section describes desirable features of a data dictionary system with the aim of providing an appreciation of the scope of DDS as well as a framework in which to assess competitive products. It is largely based upon the BCS Data Dictionary Systems Working Party report of 1977 which still remains an important benchmark in this field.

1.10.3 Desirable Features of a Data Dictionary

The BCS Working Party recommended that the Data Dictionary should operate at two distinct levels. The first is the logical level that gives the ability to record and analyse requirements irrespective of how they are going to be met. This logical view represents an implementation independent view of the enterprise and initial and successive implementations must take place within the scope of; this framework. At least four benefits may be obtained from building this logical model:

- A perspective for system planning.
- An appreciation of how systems interact;
- A method of communicating complexity.
- A database design tool.

The second level of the Data Dictionary is the implementation level. This gives the facility to record physical design decisions in terms of the implemented database or file structures and the programs that access them. If the logical level is how the data is seen from the enterprise, then the implementation level is how the data is viewed by the file handing system or the DBMS itself.

The logical view describes the nature of the enterprise and its data. It is a model of the organisation showing things of interest to it, functions it can perform and events which influence the way it performs. It is independent of any current or proposed implementation and so represents the logical requirements that successive implementations are designed to fulfill. The Data Dictionary should be able to support this model.

At this level the data dictionary should be able to record details of:

- Entities and relationships of concern to the enterprise;
- Processes of interest to the enterprise or carried out by it;
- Responsibility for processes, perhaps in terms of the structure of the organisation;
- Flows which result from processes or from external entities or events;
- The connections that exist between entities, processes and events.

The data dictionary should be able to record details of different versions recognised as valid at different times or contexts. It is also essential that the dictionary can define the relationships between the logical entities and the corresponding files and records of the implementation.

1.10.4 Implementation Level

The implementation view is the basic source of information about the physical data processing system. It provides data to help establish the design of the system, to prove its correctness and to identify the impact and cost of changes . It is likely to represent a partial implementation of the logical model. It must be logically consistent with this latter model and not exceed it in scope. At this level the Data Dictionary represents a coherent, centralised library of data about all aspects of the data processing system, enabling all users to have a clear ;and consistent view.

Two examples of such data are:

1. **Data Description elements.**

These will describe the different data types and structures used in the system, such as records and files. Elements should be described in terms of their:

Name:	Including aliases and past names.
Classification:	Description, ownership, status, etc.
Representation:	Type, length, order, etc.
Use:	Frequency and volumes.
Administration:	Memory and storage requirements.

2. **Process Description Elements.**

These will also demand the same type of metadata as the Data Description elements. Further information might include:

- Program size -in some appropriate metric.
- Processing type -Batch or On-line.
- Parameter -Number and types required.

Several versions of programs and data structures may exist at any one time. These may represent live, test or design states and this must be recognised and recorded as such.

The data dictionary should also enable descriptions of the implementation-level structures to be established and maintained. This may be achieved through a direct input language, from program data definitions in high level language, from a DBMS source definition or from program procedure definitions.

The implementation-view also demanded details about the physical storage of data and its use. Facilities required include recording of physical attributes such as:

- storage media: storage media type, eg disk
- storage size: describes space requirements, eg 640 Kbytes
- CPU: describes the CPU name and size required.

The data dictionary system should validate input for syntax, consistency and completeness. These checks should include:

- The characteristics of each physical file.
- The contented of each file.
- Each physical structure. checks that all the constituents of the implementation data structure are allocated to at least one physical file.

The implementation view contains all the information necessary to derive an 'optimum' operational schedule. This is supported by the collection of performance and utilisation statistics such as:

- Frequency: indicates the average frequency that the file is accessed (such as daily, weekly).
- Response: refers to the response time of a process.
- Log information: shows statistics on when a record or file is accessed, by whom, and the activity that is performed;
- Usage statistics: records summary of usage.

Holding data volumes in each operational definition and the physical description of the files themselves provides information for the realistic simulation of database performance. This gives the facility to tune the performance of the system to achieve a databases file structure that give optimum performance.

1.10.5 Data Dictionary Functions

The previous section provided a flavor of their contents of a Data Dictionary as well as introducing performance simulation as one of its possible functions. Similarly, the documentation and control features of a DDS have been described earlier in this chapter. However, the Data Dictionary has other important functions. These include:

1. Consistency Checking

This is an essential feature of a Data Dictionary. In the context of a data flow diagram this can answer such questions as:

- Are there any data flows specified without a source or destinations?
- Are there any data elements specified in any data stores that have no way of getting there, as they are not present in any of the incoming data flows?
- Does a process definition demand a data element that does not enter that process?
- Are there any data elements in any data flows entering processes that are not used in the process and/or do not appear in the output?

The verification of system consistency is a vital task that eliminates a considerable amount of desk checking. It permits the insertion of vital, but omitted, data elements and the deletion of irrelevant ones. Consequently, the data dictionary not only ensures the validity of the design but also identifies and justifies the role of each data element. Thus it is possible to demonstrate why certain data is collected and where and how it is used.

2. Testing

The development and entry of test data is extremely time consuming. But if descriptions and ranges of values are already stored then test data can be automatically generated.

3. Coding

The description of data structures may be detailed enough for the generation of data descriptions in the host language or Data Manipulation Language (DML) through a precompilation pass of the dictionary. Furthermore, if the implementation level model holds information on the sequence in which data is used by processes then automatic program code definition is feasible. A further obvious application lies within the definition of validation rule in the Data Dictionary permitting the generation of data validation routines and integrity checks.

4. Change

Program and system maintenance is a major system overhead. One of its most time consuming and difficult tasks is the tracing of the effects of changes through the complete system. Impact analysis is the term given to the analysis of the effect of proposed program and system changes. The recording of the relationships that exist between the various entities should allow the effects of addition, amendment or deletion of a particular entity to be predicted throughout the whole system. Thus the system and resource implications of a change can be completely understood. It may also be possible to generate some of the coding changes automatically in an active, or potentially active, dictionary.

5. Reporting and Security

The initial documentation role of many Data Dictionaries has led to most commercial DDS having flexible and comprehensive reporting facilities.

A variety of analyses will be required and the facility to search for textual descriptions probably essential if the full potential of the system is to be realised. The output facilities will play an important part in the selection of an appropriate DDS because these are the most immediate part of software. The principles of good output design, described in our companion text Introducing System Design, have to be observed.

Security codes can be assigned to individual systems and/or metadata entities, or to part or all of the Data Dictionary. These are used to restrict access to specific meta objects (entities, processes, files, records, etc.) and this type of security can be implemented through passwords, authority or ownership facilities. Security design is considered in the companion text and the general principles outline there apply to the design of Data Dictionary systems.

1.11 Data Modeling

Most people involved in application development follow some kind of methodology. A methodology is a prescribed set of processes through which the developer analyses the client's requirements and develops an application. Major database vendors and computer gurus all practice and promote their own methodology. Some database vendors even make their analysis, design, and development tools conform to a particular methodology. If you are using the tools of a particular vendor, it may be easier to follow their methodology as well. For example, when CNS develops

and supports Oracle database applications it uses the Oracle toolset. Accordingly, CNS follows Oracle's CASE* Method application development methodology (or a reasonable facsimile thereof).

One technique commonly used in analysing the client's requirements is data modeling. The purpose of data modeling is to develop an accurate model, or graphical representation, of the client's information needs and business processes. The data model acts as a framework for the development of the new or enhanced application. There are almost as many methods of data modeling as there are application development methodologies.

1.12 Types of Database Management System

Some commonly used relational DBMS are as following:

- MS-Excess
- INFO
- EMPRESS
- ORACLE
- Sybase
- IBM's DBII for Mainframe Computers
- dBASE VII.

1.12.1 Oracle Based Databases

A database is a structured collection of data. Data refers to the characteristics of people, things, and events. Oracle stores each data item in its own field. For example, a person's first name, date of birth, and their postal code are each stored in separate fields. The name of a field usually reflects its contents. A postal code field might be named POSTAL-CODE or PSTL_CD. Each DBMS has its own rules for naming the data fields.

A field has little meaning unless it is seen within the context of other fields. The postal code T6G 2H1, for example, expresses nothing by itself. To what person or business does it belong? The postal code field is informative only after it is associated with other data. In Oracle, the fields relating to a particular person, thing, or event are bundled together to form a single, complete unit of data, called a record (it can also be referred to as a row or an occurrence). Each record is made up of a number of fields. No two fields in a record can have the same field name.

During an Oracle database design project, the analysis of your business needs identifies all the fields or attributes of interest. If your business needs change over time, you define any additional fields or change the definition of existing fields.

- Extra processing overhead.
- As data is centralised there is a reliance on a central site.
- More complex to develop.
- If developed incorrectly, it can have disastrous consequences.

1.12.2 Microsoft's SQL Server 2000

Microsoft SQL Server 2000 (originally given the code name Shiloh) is the widest used SQL server after Oracle's SQL. Microsoft's development team included enhanced security mechanisms, simplified administrative tools and support for advanced web integration technologies. Many database professionals believe that this release marks SQL Server's entry into the big leagues, ready to take on Oracle in the high-reliability DBMS world.

It includes greatly enhanced analytical features. Full-fledged data mining support is provided through the use of classification, regression and clustering algorithms. Microsoft's implementation of online analytical processing (OLAP) technology allows data warehouses to store and manipulate large amounts of data with newfound ease. Additionally, database administrators concerned with importing data from diverse sources will be interested in SQL Server 2000's enhanced Data Transformation Services.

Security enhancements abound in the latest SQL Server release. Most significantly, Microsoft adopted the security industry standard Kerberos authentication algorithm to facilitate creation and enforcement of secure trust relationships. The National Security Agency recently announced their certification of SQL Server 2000 as a C2-compliant computing platform. Reliability improvements such as log shipping, online backups and failover clustering are discussed in the article Maximising Uptime.

An old adage reminds us that two statisticians can analyse the same set of numbers to reach diametrically opposed conclusions. Database server benchmarking studies prove no exception to this rule!

SQL Server 2000's debut marks the release of the first Back Office server product supporting Microsoft's .Net strategy. Developers will discover inherent support for Extensible Markup Language (XML) programming as one of a series of improvements designed to enhance the SQL Server development environment.

1.12.3 Microsoft SQL Server 7.0

The database is the heart and soul of any company's business infrastructure. Small to medium-size businesses planning to make this critical investment must consider their specific business needs, support requirements, and budgets. The solution that offers the best value with the least administrative hassle is Microsoft SQL Server 7.0.

In the small-business market, the differentiating factors are ease of database administration, Web connectivity, the speed and features of the database server engine, branch-office and mobile support, and the ability to warehouse data efficiently. SQL Server 7.0 shines in all of these areas except Web connectivity. Its administration tools include many wizards and self-tuning settings that make it the only database we reviewed that might not require a specially trained administrator.

For branch offices or distributed sales teams, SQL Server's per- seat client license includes the right to run its desktop version, and the database has excellent replication features that let you sync up with home base and vice versa. An OLAP server in the box lets you store and analyze all the data you have. All this comes at a fraction of the cost of competing databases.

Oracle 8i has some excellent features, including its multiversioning concurrency system, and its Internet-centric approach is the way of the future. But with the database server still more difficult to administer than the others we saw, and at its high prices (especially when many key features, standard elsewhere, are available only at an extra cost), Oracle is hard to recommend unless you are building a custom application that requires its Java or multimedia features.

IBM DB2 impresses with its stellar server engine, multimedia support, and Java programmability—all areas where it surpasses what Microsoft SQL Server provides. It costs more and will require more care and feeding than SQL Server, but this is one to evaluate seriously as you make your buying decision.

Packed with more features than competing products and sporting a very attractive price, Microsoft SQL Server 7.0 is a modern, full-featured SQL database that's right on target for the small or midsize organisation. Its complete set of tools, high-end engine features, and robust analysis capabilities provide most of what IBM and Oracle offer only in their Enterprise Edition databases (and a few key things that neither includes) for less than either one. In addition, we found SQL Server 7.0 amazingly easy to use, yet still powerful enough to crank through hundreds of complex transactions a second on our test hardware without choking. (SQL Server also comes in an Enterprise Edition version, whose main benefit is to allow automatic failover to a secondary, standby machine.)

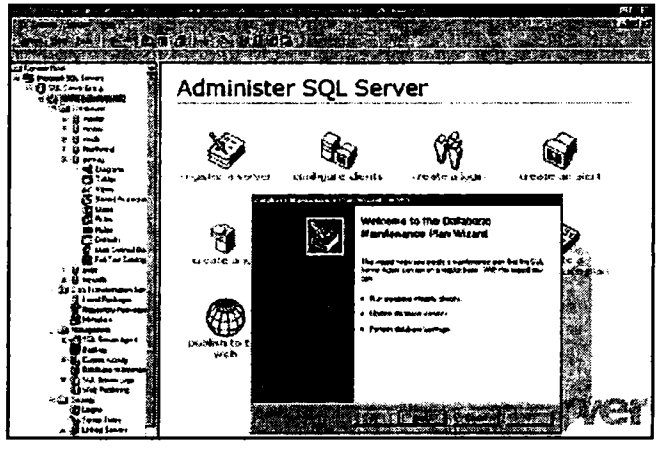


Fig 1.10: SQL Server Administration

Yet SQL Server still lags behind the competition in two key areas: programmability and multimedia data support. For those who write their own business applications and put part of the code in the database itself (as opposed to putting all the code into applications that run on your Windows client systems), and for those who are moving to HTML- or Java-based database client applications, SQL Server provides less built-in functionality and will be harder to use than DB2, Informix, Oracle, or Sybase.

SQL Server also runs only on Windows operating systems, so if you are running Unix-based servers, look elsewhere. But if you are running generic packaged Windows applications to access your database contents, or if you have custom applications that use only the ODBC standard to get information from the database, you will be right in SQL Server's sweet spot.

Version 7.0 is more than just an upgrade of SQL Server 6.5. It has a new heart and brain in its latest incarnation. Think of 7.0 as a totally new database that just happens to have somewhat similar administration tools and to use the same programming language (Transact-SQL).

Microsoft has made a whole host of changes to bring its enterprise database up to technical snuff. SQL Server's engine— the heart of the product— now has a different data file format than its predecessors had.

One of the key reasons for changing the database file format is a switch to using 8K disk pages (the basic unit of database storage) instead of 2K pages. This permits storing more information in any database row and also makes the database faster when reading in many rows at a time, since the server has to retrieve fewer data pages from disk.

Also, SQL Server can now isolate data and index information at the row level (a feature called row-level locking). Previously, SQL Server could protect one user's changes from the activity of other users only by blocking other users' access to a group of perhaps 40 or 50 rows (depending on the size of a row) around the one being changed. SQL Server 7.0 locks down only the actual rows being changed and, as a result, can support more users at once. Some packaged business applications, like those from People Soft and SAP AG, really need row-level locking to be at all usable.

The query optimizer—SQL Server's brain—also got a total rewrite. Most important, it now supports intraquery parallelism, the ability to process a single query on multiple CPUs. This speeds up queries so much that you should consider the feature a must-have for your database software if you have a multi-CPU server.

Top-Notch Tools

Beyond all its engine changes, SQL Server includes an unmatched set of database tools. New with this release is a server product called SQL Server OLAP Services (OLAP stands for online analytical processing). It is a specialty-purpose database server that precalculates summary information for easy analysis and comparison.

For example, if same one may use OLAP Services to view sales figures by store, by town, by region, and by country, and then compared last quarter's figures with those of this quarter. Since the totals had all been figured out ahead of time, you could flip instantly between high-level totals. With other OLAP servers currently priced at tens of thousands of dollars, OLAP Services is an incredible bargain.

OLAP Services is powerful, flexible, and easy to use. However, it's not all roses. First, you have to buy separate, compatible software to use it, because OLAP Services uses a query language different from the one SQL Server uses. (Excel 2000 is one such client, but others are available from companies like Cognos Corp. and Seagate Software.)

Second, SQL Server doesn't make any use of OLAP Services (or of precalculation techniques in general) to speed up its own queries. The two are completely separate products; you have to load data from SQL Server into OLAP Services before we could do any OLAP analysis. By contrast, both IBM and Oracle have introduced some OLAP features directly into their latest databases, in the form of query summary tables.

Ease-of-use Leader

Microsoft SQL Server 6.5's administration tools were decent, but SQL Server 7.0 takes ease of use to a whole new level. Many engine settings in SQL Server 7.0 are self-tuning. For example, you didn't have to assign memory to our data cache and stored procedure cache separately; SQL Server dynamically balanced memory between the two. SQL Server also expands or contracts the amount of memory it is using as a whole: it automatically makes room in the memory for other applications when they were running, and expand again to fill extra memory when they are closed them down.

Automatic memory tuning isn't especially important on a dedicated database server machine. But on a server that has to run other applications like a mail server and Web server at the same time as a database, dynamic memory sizing makes a huge difference to system usability.

SQL Server can now run on Windows 9x systems, a very handy feature for organisations with mobile sales forces. Each salesperson could use locally installed database applications, such as an order-management system, and then synchronize customer and order data with headquarters as necessary.

Also boosting branch office support, SQL Server 7.0 includes competitive replication features that support a number of one-way and bidirectional links between SQL Server machines. As a result, databases in branch offices and headquarters automatically keep each other in sync.

SQL Server's unmatched auto-tuning features mean it is the best choice for organisations that don't have database administrators on staff. While a good DB2, Oracle, or Sybase administrator can manage several hundred database servers from a central location, you do need at least one experienced administrator to use these databases effectively. SQL Server can get by with a part-time or beginner administrator quite well.

Another interesting new tool in SQL Server 7.0 is the Index Tuning Wizard (part of SQL Server's Profiler tool). It suggests new server indexes based on actual database usage (indexes are special database objects that speed the process of looking up information in a database table, and they are essential for getting good database performance). Programmer are able to turn on index data capturing, run a set of queries against SQL Server, and see what index suggestions the wizard made. It then asked us whether it should go ahead and create the new indexes for us, providing one-step tuning. IBM and Oracle also offer similar tools, but Microsoft's Index Tuning Wizard has a small edge over those offerings, because its options for capturing database activity are more flexible.

Programming Pitfalls

As mentioned, SQL Server still offers only the same old cumbersome Transact-SQL programming language it always had. While everyone else in the SQL database market is moving (or has

already moved) to a modern programming language like Java, SQL Server customers are still stuck in the programming Dark Ages—no object-oriented development, no big class libraries to use, and no code interoperability with anything else. Although it is certainly possible to create large applications using Transact-SQL, you are in for a lot of work.

The same applies to companies that want to store multimedia data (such as the contents of a Web site) in their databases. While DB2, Informix, and Oracle go out of their way to make this easy, SQL Server has no special image, sound, video, or geographic data support. You will need to purchase a third-party application that has worked around SQL Server's limitations.

1.12.4 Sybase Adaptive Server Enterprise 12.0

Taking direct aim at Java programmers, the forthcoming Sybase Adaptive Server Enterprise (ASE) 12.0 hits them squarely between the eyes. The server will be more Java-centric than any other database on the market and will be a good choice for companies moving to Java as their preferred programming language.

In terms of its relational database features, ASE is competitive but not a leader. It supports recent database technology advances like the ability to use multiple CPUs to process a single query, and it supports row-level locking, which allows more users to access the database at once. ASE 12.0 has merge joins, a good way to join sorted information from large tables, as well as new support for SQL outer joins. This makes it easy to ask queries like "Show me a list of customers and their orders, and include customers who don't have any orders at all yet." These features were already provided by the competition.

Yet ASE still has to catch up in some important areas. It doesn't have summary tables, and it lacks the ability to do hash joins, a fast way to combine unsorted information from many large tables. DB2 and Microsoft SQL Server also provide multi-CPU support, plus all the relational engine features ASE lacks, as well as greater usability, making them better choices if Java support is not important to you.

ASE does have one very distinctive feature that only Oracle8i Enterprise Edition shares: It can prioritize queries based on who submitted them. You can limit groups of users (for example, those who run routine reports) to using only one CPU, while other users (say shop floor workers who need quick response times no matter what) could have access to all the CPUs on the machine. You could also choose to dedicate a CPU for shop-floor use only. On our tests, a reporting query mix took 5 minutes instead of 6 minutes 40 seconds when we ran it using two CPUs instead of restricting it to one.

ASE's Java support has its limitations, though. For example, since it requires a new query syntax, older applications will have to be rewritten to take advantage of ASE's Java features. By contrast, DB2 and Oracle both include functions that can make object-oriented data look like normal relational data. This allows you to update your database designs without having to change your client applications.

We also noticed that we couldn't retrieve Java objects in sorted order without adding some extra code to convert Java data types to SQL data types. We were able to use normal SQL commands to group Java objects into categories or retrieve only particular Java objects based on their contents (for example, selecting only employees that had salaries over Rs. 50,000).

Sybase has arranged for partner companies to develop extra-cost multimedia extensions for ASE, options that make complex text or geographic searches possible. The approach is clunky, though, because the multimedia products aren't integrated into the database's query optimizer or indexing engine. If you're going to be doing a lot of multimedia work, choose a database like DB2, Informix, or Oracle8i that's really designed for this task.

Sybase's latest offering holds some promise for companies standardizing their development around Java. But otherwise it still seems to be a step behind its competitors when measuring the state of the art for databases. (Sybase Inc., Emeryville, CA; 800-879-2273; www.sybase.com.)

1.12.5 IBM DB2 6.1 SQL Server

The result of almost 30 years of database research, IBM DB2 can run your small business as well as it can manage your bank's business—which is pretty well indeed. DB2 6.1, the latest in a series of big upgrades to DB2 over the past two years, wraps one of the best-designed sets of administration and tuning tools on the market with a database engine that scales from a glued- together plastic laptop running Windows 95 up to a cluster of S/390 mainframes running OS/390.

DB2 6.1 (replacing the current DB2 5.2) is due to hit shelves around the time this issue comes out. We tested beta 2 code of DB2 6.1, which was complete except for some bug fixing and performance tuning.

There are two versions of DB2: DB2 Workgroup, which costs $999 per server and $249 per concurrent user, and DB2 Enterprise Edition, which is $12,500 per CPU, with unlimited users.

The first step in DB2 program preparation is, of course, writing a program that contains embedded SQL statements. You could use COBOL, C (or C++), Assembler, FORTRAN, PL/I, or Java. (You may think that Java implies only dynamic SQL, but it doesn't if you take advantage of SQLJ. For a good introduction to SQLJ, see the Web Database column in the Spring 1999 issue.) You could write the program to run on the system on which the DB2 for OS/390 subsystem resides, or to access the DB2 database from a remote client via distributed relational database architecture (DRDA).

Once you've written the program, the DB2 precompiler processes it and generates two outputs:

1. A modified program source module. The precompiler comments out each of the program's embedded SQL statements, and inserts a call to DB2 for each statement.
2. A database request module (DBRM). A DBRM contains the SQL statements found in the program source.

The precompiler places a unique identifier, called a consistency token, into each of these outputs. I'll say more about consistency tokens when I get to program execution.

Following the precompile process, you compile and link-edit the modified source program into an executable load module and bind the associated DBRM. In the DB2 for OS/390 bind process, such tasks as access path selection (optimization), access authorization, and database object validation are performed. The output of the bind process is a control structure that DB2 will use to execute the SQL statements when the application program is run. The control structure will either be part of a plan (if the DBRM is bound directly into a plan) or contained within a package that will be executed via a plan.

How many of you work at a DB2 for OS/390 site that still binds DBRMs directly into plans? I wish I could see a show of hands. I would expect (and hope) that few hands would be raised. The DB2 package bind process has been around for quite some time (since DB2 version 2 release 3), and it offers some important advantages over plan-direct binds.

- **Improved availability:** If you change a SQL statement in a program, you only have to rebind one package. You can rebind one package quickly, and it's important to do so: A package cannot be executed while it's being rebound. If, on the other hand, programs are bound directly into plans, a change of one SQL statement requires that the plan be rebound. If you have bound a large number of DBRMs into the plan, the rebind could take a fair amount of time, during which the plan cannot be executed.
- **Improved flexibility:** As will be see, binding programs into packages allows you to do some nifty things with your database and application architecture.
- If you use the package bind process, you have to bind the package into what is called a collection. How do you create a collection? Pretty simple: You bind a package into it.

Of course, even if you are using packages, you still need to bind one or more plans if the program in question will run on the local or a remote DB2 for OS/390 subsystem. Programs that run on other remote clients and access DB2 via DRDA use a default plan called DISTSERV. You can execute a particular package using a plan if the collection into which you've bound the package appears in what is called the plan's package list (a list of one or more collections specified via the PKLIST option of the BIND PLAN command). The program, in turn, invokes the plan through a specification in the resource control table (or a DB2ENTRY if you are using resource definition online) for a CICS transaction, via the application program load module name for an IMS transaction, or with a control statement in the job control language (JCL) for a batch job.

When you execute the application program, each call to DB2 directs the database manager to execute the corresponding prebound SQL statement in the package associated with the program. (Recall that the precompiler comments out SQL statements in the source program and adds calls to DB2) DB2 searches for the package in one or more collections using as search criteria the package name (same as the program name) and the consistency token accompanying the call. (Recall that the consistency token, generated at precompile time, is carried in both the application program and the related package) When a match is found, the statement is executed and control passes back to the application program until the next DB2 call is issued.

After accepting this idea, then clearly you need to buy an application server. You have Oracle. You have a Web server program. But you have no software at all for one critical tier.

Before trashing the idea of the application server, let me trash the idea of the three-tiered architecture for Web services.

DB2 also has a full set of multimedia extensions for storing and manipulating full-text, sound, video, image, and geographic data. These extensions make designing Web-based applications and applications that include photos or long text reports much easier. They are extra-cost options, but they offer features and pricing that are very competitive with those of Informix and Oracle.

DB2 is competitive as an applications development platform as well. We could use Java to code database logic, and a new tool in DB2 6.1—Stored Procedure Builder—automatically turned a SQL statement into an equivalent Java class for us and then installed it right into the database. DB2 doesn't otherwise include an internal programming language (it is the only big database on the market that doesn't), so some Java expertise is essential when developing applications with DB2.

DB2 6.1 also provides much better interoperability than before because it includes built-in support for OLE DB, Microsoft's new database access standard (designed to replace ODBC). With this support, DB2 can connect to and access database contents (but not update data) on just about any other database. For example, using the OLE DB link we could transparently make a Microsoft SQL Server table look like a DB2 table—very handy for shops that have more than one type of database in house.

DB2's administration tools, rewritten in Java for this release and now Web-accessible, are top-notch. Only Microsoft SQL Server has better tools and takes less work to maintain. DB2 has a detailed graphical query plan display utility, comprehensive performance-monitoring tools, and a full task scheduler for running jobs at regular times.

Although DB2's tools are better, we think you will still need a trained database administrator to run DB2 effectively; it is not easy enough for a nonprofessional quite yet. For example, when we ran a test query that modified a lot of rows and filled up DB2's transaction log, DB2 aborted the transaction and gave us an error instead of just extending the log as Microsoft SQL Server does. It doesn't have the automatic memory tuning SQL Server does, either, and there are a huge number of server options to face when you try to reconfigure the server.

But IBM is making big usability strides with this release. For example, DB2 6.1's Index Smart Guide, similar to Microsoft SQL Server's Index Tuning Wizard, made tuning our DB2 databases much easier than in the past, letting us create the right indexes for our query workload. After we had created our test database in DB2, we submitted a number of queries and then started the Index SmartGuide. After analyzing our workload, it offered to create appropriate indexes for us.

DB2 is the only database in this roundup that provides summary tables, a major performance booster for shops that do data warehousing or a lot of end-of-month reports (summary tables are also available in Oracle8i Enterprise Edition). A summary table is a kind of scratch-pad area that a database can use to store answers to commonly asked queries.

For example, we created a table of monthly and yearly sales summaries from a table containing individual sales orders. Whenever we asked a query that needed any monthly or yearly sales totals (or any subset of our summary table, really), DB2 automatically pulled the information out of the related summary table. We saw queries on a 500,000-row table drop from taking 18 seconds to being answered as fast as we could blink.

With these new features, combined with its support for parallelism and almost complete selection of join and index types (except for bitmapped indexes), DB2 6.1 is shaping up as a top performer at a bottom-dollar price. Its administration tools are where they should be, and it offers great multimedia data handling and programmability, the two key weaknesses of Microsoft SQL Server. For organisations that can handle its somewhat higher complexity and cost, it will pay off big.

1.12.6 Informix's Centaur SQL Server

Sometimes being first just means you have to wait longer for everyone else to catch up. That's what Informix found out in 1997. The company was one of the first of the big relational database vendors to add multimedia extensions to its database. Now that everyone from IBM to Oracle to Sybase has leaped upon that notion, Informix seeks to differentiate itself again with a big relaunch of its core database technology. It will merge its core pure relational database, Informix Dynamic Server 7.3, with its object-relational database, Informix Universal Data Option 10.1. The idea is to create a product that provides the performance of Dynamic Server on relational data with the flexibility and multimedia support of Universal Data Option.

We couldn't get beta code to test, but we did get an early briefing from Informix officials to get the scoop on the new product, code-named "Centaur." It is targeted at customers involved in Internet development. The goal is to provide a database with a very flexible development environment, scalability to handle Internet-class loads, and the capacity to handle the new types of data the Web has made important.

Most notably, Informix has big plans to integrate Java into its database, but it is keeping most of the details secret until Centaur's official launch, around the time you are reading this issue.

Informix did note that it will distinguish its product by allowing customers to choose from a selection of standard, off- the-shelf Java virtual machines to use with its database. (IBM, Oracle, and Sybase all use specially customized Java virtual machines with their databases.) Java support will allow Informix developers to write Java-based stored procedures, user-defined routines, and Data Blades, which are what Informix calls custom extensions to the database.

This will be big step forward for Informix customers, because right now the only languages they can use for DataBlades are C and SPL, Informix's internal stored-procedure language. Centaur will also have built-in support for Active X objects, making it possible, for example, to write a database stored procedure in Visual Basic, as long as you are running Centaur on Windows NT.

Centaur will be a superset of Informix Dynamic Server and will use the same database format, so customers won't lose any features or have complicated upgrade procedures when making the switch. Centaur will also include all the design and programming flexibility that made

Informix Universal Server such a technical achievement. It will support object-oriented database designs, custom table and index routines, and an extensible SQL query parser, so customers can add arbitrary functions to their SQL queries and not just use standard SQL functions.

Informix's Centaur looks as if it will put the company back into the database race—and provide current Informix customers with a solid growth path for the future. (Informix Software Inc., Menlo Park, CA; 800-331-1763, 650-926-6300; http://www.informix.com/.)

1.12.7 Other SQL Databases

There are more SQL database systems available for Linux than anyone would expect. Lots of commercial products; a surprising number of free SQL databases.

This is presented at the MySQL web site, so it might well have certain biases.

- Informix,
- Oracle,
- Postgre SQL, MySQL)
- IBM Data Management: DB/2 UDB 5.2 for Linux

IBM Data Management: DB/2 UDB 5.2 for Linux

IBM now provides Linux support for their database system. This leaves only MSFT's SQL Server as the only member of the "top tier" of RDBMS products that don't have any sort of support for Linux.

An "embedded" version of DB/2 with around a 150K footprint. It is particularly designed for use on small form factor computers such as PDAs. It then offers a Sync Manager to synchronize mobile data with a full scale DB/2 server.

1.12.8 MySQL

This is a fairly mature database system that has recently been modified to use an API compatible with mSQL's. It's "freely available" for many uses, but also offers commercial support for those needing such. There is a version now that is free software, using the GPL as its license and it supports ODBC.

It is multi-threaded, optimized for speed and for use with large tables. As speed is one of its primary design goals, it does not implement some SQL features that tend to diminish performance, such as transactions, subselect, select into table, stored procedures, triggers, foreign keys, or views. The upside is that this allows it to be considerably faster than many other SQL databases. Of course, if you specifically wanted transactions, this may be troublesome, but I actually think that that's where one might want to start looking at a transaction processing monitor, which could separate that out.

1.12.9 Integra RDBMS

Adding to the set of "international" SQL databases is this one from India.

They are not as unknown as one might think; their Visual Query Builder was apparently included with some versions of Borland C++, and SCO has bundled their C-ISAM library (now named ObjecTrieve) as part of some editions of SCO Unix.

PrimeBase " The PrimeBase DBMS supports ODBC, SQL and DAL and the most popular protocols (TCP/IP, AppleTalk) and platforms (Mac, Windows 95/NT, Solaris, LINUX, IBM AIX, etc.)." Freely available for development projects...

1.12.10 PostgreSQL

This is an extensible "object-relational" database system, the most sophisticated and probably the best supported of the "free "databases.

At one time, this was the Postgres research RDBMS engine, by Michael Stonebraker (founder of Ingres, the first commonly- available relational database, now a "senior technical dude" at Informix). People liked Postgres enough that some people decided to make the query language SQL-compliant.

There was a commercialized version of Postgres that was called Illustra; the company was bought out by Informix, and its object- oriented features have been integrated into Informix Universal Server.

More "free third-party tools" have been created for PostgreSQL than any of the other free databases, and include:

- Perl, ODBC and JDBC support;
- PgAccess - a Tcl/Tk-based interface;
- KPGsql, a Qt-based query tool;
- GtkPGA, a Gtk-based query tool;
- EARP - Easily Adjustable Response Program; and
- Interfaces with Guile such as guile-pg and pg-guile.

There's a Linux Database HOWTO on PostgreSQL. The author waxes a bit overevangelistic/ overenthusiastic about PostgreSQL, somewhat overstating its advantages. He makes the same sorts of claims that allow Project Gutenberg proponents to claim that their work is worth "billions of dollars," when the fact that the public values them enough to donate a few tens of thousands of dollars suggests that they may be overvaluing things a mite. He has a HOWTO on "building your own CPU" that has similarly optimistic valuation of the merits of designing a "Free CPU" (in the free software sense).

1.12.11 dBASE 7

Developed by Borland, the company has finally presented the remoulded Visual dBASE in the 32-bit version. DOS-compatibility that plagued the earlier two Windows versions of dBase no longer dominates the product. Instead, Borland is all geared up for a new file format, a user-friendly interface and a programmable report generator, dBase will continue to support the classical DBF formats that can be read by most applications, while Microsoft Access and Paradox files will have native support.

Although Visual dBase offers a complete environment \for the developer, Borland also seems sto have considered the end-user while pricing the product (introductory price of Rs 15,400 for professional version of the software). Assistants and comprehensive multimedia support

(including JPG and animated GIF,s) make dBase a complete and attractive database. Borland apparently has also had a good look at Microsoft's Access. Like Access, they have made data entry in the table grid very comfortable and user-friendly.

Static and Dynamic data in HTML

Borland's Intra Builder based on the JavaScript is a reliable tool for Internet and intranet applications. So there is no scope for competition between these in-house products. The interface and operation are very similar in both programs. If you can work easily with one, you should not face problems working with the other.

Like the 16-bit version of Visual dBase 5.5, the new version offers several tools to publish data on the Internet. A form feed programmable in Visual dBase takes you through the process like an assistant. Even those who are not used to the Common Gateway Interface (CGI) script should face no difficulty while building a database query in the Web Browser. Other tools for creating HTML pages and page management utilities are provided by Deltapoint, Borland's Web Server that is supplied with the package. The import and export function in dBase is disappointing. Although the very fast database engine BDE 4.5 allows access to almost all data (even text, Excel, Access), the entire process could get uncomfortable. To convert a dBase table in the new format to one with an older, compatible format, you have to switch to the driver in the BDE.

1.12.12 MS Access 97 Market and Operation Leader

Microsoft's Access has been considered as the user-friendliest database, but is overrated as far as simple address and telephone management goes. Assistance support user operation and the program is also suitable for more complex projects. Each version might differ in compatibility, but this is something that the current users of MS Office should be accustomed to.

The MS Access 97-file format is based on the tried-and-trusted structure of the previous version (Access 95), the new field for hyperlinks and Internet addresses however causes complications while working with the older version. The entry of reference fields is very simple. An assistant allows you to define the data source while creating a table, from which reusable data like postal code can be taken. By skilfully using reference fields, work with several, linked databases is considerably simplified.

Access cuts a good picture when generating reports and labels. You can also view the standard reports produced with the assistants. Unlike dBase 7, Access does not provide you a WYSIWYS (What You See Is What You Get) data in the preview mode.

An HTML export of reports and forms directly possible through a Web assistant. It does produce reasonably good Web pages and enables text and table formatting, but bitmaps cannot be automatically linked on to the Web pages-a serious shortcoming for companies that want to publish their staff databases and photographs on the internet.

The Web assistant offers functions such as creating a homepage, from where you can launch pages along with the tables, evaluations and screens. However, there are no options to select the page background or create navigation buttons. However, you can link readymade HTML templates.

1.12.13 Corel Paradox 8

Background-compatibility is the strongest point of Corel's Paradox. New tables can be created and edited in the Paradox, dBase and even in MS Access 95 formats. Paradox allows you to use Access 97 through a native interface. Like dBase 7, Paradox access databases through the Borland database engine. Even the graphic SQL query generator corresponds to that of Visual dBase 7. Like Access 97, Paradox allows you to create new tables easily. Field selection is another advantage. Data set limitations, input screens and reference fields can be directly entered in the data set plan.

Layout and Report-generating utilities are flexible-albiet they need manual work too. For instance, a few mouse-clicks could create a report, which would be satisfactory except that the results would not match with the length of the data fields.

Reports and form feeds can be published in HTML former with an assistant, but text-formatting does not help. The Corel Web Server supplied with the package enables you to develop dynamic Web applications although templates and buttons for the data set search in Web browser are absent. Delphi programmers can extent the HTML interface through the developer kit provided with the product.

1.12.14 File Maker Pro 4.0

Simple but Internet friendly Claris advertises the possibility of importing Excel files to File Maker Pro, but its weak export filters do not really help this exercise. There is no way to do this even in the dBase format. Apart from the standard fields such as data and time, the File Maker data format also provides a formula and an evaluation field.

In case you have already used other Windows databases, you might not enjoy working on File Maker. Though the program offers usable templates, it refrains form providing help tips like Access and Paradox. File Maker is also compatible with the Macintosh.

1.12.15 Lotus Approach 97

Approach 97 comes across as a user-friendly application because of the backing of remarkable functions. Even if the typical Lotus interface should intimidate experienced users, Approach 97 offers more than assistant support for the home user. Data set format of the application makes it seem rather open. Use of the dBase and Paradox formats allow existing client databases to be integrated without conversion and can be linked with contacts from the Lotus Organizer.

Highlights of the program include access to Lotus Notes and DB2. Like Access, Approach provides several templates while designing a new table. 'Smart Masters'-Lotus templates, produce complete applications with a mouse-click, for example an order management system. Relational links between the client and the invoices are now nothing to worry about. Those familiar with linking tables will find the Link Editor very useful.

Approach performs well while generating reports. It displays data even as you do the initial layout. Unlike Paradox or Access, you need not switch between the layout and the preview to view the data. Grouping and integrating calculations are easy to do. Approach 97 is the only database that allows you to save files and applications directly on the Internet and the

Internet. Data can be stored on an FTP server from the Save Menu, from where it can be accessed by all users. Reports, from feeds and tables can also be directly saved as HTML documents.

1.13 Who Uses a DBMS

- Database Users
- Relative to the database a user could be:
- Data entry person using an application program
- Data entry person using a forms based system
- Professional non DP person using a query language application programmer
- Program that accesses the DB –written in, say, COBOL, C++, Perl (for web programming) or even Microsoft Access such programs include some sort of data sublanguage [where data sublanguages consist of: Data Definition/Description Language (DDL) and Data Manipulation Language (DML)] which is used to access the DB. eg. SQL.

Each user's program application has its own host language (eg. COBOL or Perl) with the DB's data sub-language (eg. SQL) embedded within the host language.

An end data entry user cannot differentiate between an applications host language and the DB's embedded language.

An external view only contains the data relevant to that user and an external view's definition of a record (logical record) may be different to what is actually stored. For example, an end user may see the entry of a new customer's order as a new, single record. However that single order may be mapped over many tables in the DB. This same end user should not even 'see' (i.e. have access) to the rest of the DB, which may include customer accounts, supplier's accounts, etc.

The database users fall into several categories:

- Application programmers are computer professionals interacting with the system through DML calls embedded in a program written in a host language (e.g. C, PL/1, Pascal).
- These programs are called application programs.
- The DML precompiler converts DML calls (prefaced by a special character like $, #, etc.) to normal procedure calls in a host language.
- The host language compiler then generates the object code.
- Some special types of programming languages combine Pascal-like control structures with control structures for the manipulation of a database.
- These are sometimes called fourth-generation languages.

Database often includes features to help generate forms and display data:

- Sophisticated users interact with the system without writing programs.
- They form requests by writing queries in a database query language.
- These are submitted to a query processor that breaks a DML statement down into instructions for the database manager module.

- Specialised users are sophisticated users writing special database application programs. These may be CADD systems, knowledge-based and expert systems, complex data systems (audio/video), etc.
- Naive users are unsophisticated users who interact with the system by using permanent application programs (e.g. automated teller machine).
- End users
- Database administrator
- Interactive users of a DBMS

1.14. Interactive Users of a DBMS

The interactive user is a class of database user who uses a DBMS from a terminal, probably using an interactive query language (like SQL) to the DBMS. This allows the user to create, update, insert and retrieve information. This is not the same as an online user using an application program which access a DBMS as in this case that user is running a program which access the database rather than the user accessing the database directly.

The database interface may be menu-driven (bullet proof but restricted as in INFORMIX) or command driven (more powerful, but requires expertise). ORACLE has a command-line interface via SQLPLUS.

Review Questions

Objective type question (1 to 5): Choose best possible atternative from following:

1. File-processing systems have important limitations:
   - Data is separated and isolated.
   - Data is often duplicated.
   - Application programs are dependent on file formats.
   - All of the above.
   - None of the above.
2. Database-processing programs:
   - Call the DBMS to access the stored data.
   - Cannot be used by more than one person.
   - Require at least one dedicated workstation.
   - Present problems with storage space.
3. In a database system, all the application data is stored in a single facility called the:
   - DBMS
   - CPU
   - Hard drive
   - Database
4. The self-describing characteristics of a database are important because:
   - They promote program/data independence.

- If the structure of the data in the database is changed, only that change is entered in the data dictionary.
- If the structure of the data in the database is changed, few (if any) programs will need to be changed.
- None of the above.
- All of the above.

5. The features and functions of a DBMS can be divided into three parts:
   - Fields, records, and files.
   - RAM, ROM, and floppy diskettes.
   - The design tools subsystem, the run-time subsystem, and the DBMS engine.
   - The file-processing subsystem, the transaction-processing subsystem, and the LAN.
6. Define the following terms:
   (a) Entity
   (b) Attribute
   (c) Data item
   (d) Record
   (e) Block
   (f) File
   (g) Pointer
   (h) Index
7. Explain, compare, and contrast the following pairs of terms:
   (a) File and database
   (b) Sequential and indexed file organisation
   (c) Indexed and hashed file organisation
   (d) Tape and disk storage
   (e) Master file and transaction file.
8. How is a record identified for retrieval? What are the criteria for selecting the identification?
9. Why should multiple record keys be needed? How can they be used in information retrieval?
10. Explain the organisation of a sequential file on tape or disk. Summarize advantages and disadvantages. Give an example where it would be appropriate. Explain how it is used in:
    (a) Updating based on a transaction
    (b) Retrieval request
    (c) Preparing a report
11. What is the relationship of improved data quality to the user in designing data files? Explain.
12. How important are the data file design criteria in developing an organisation's data base and off-line files.
13. Explain the concept of a data model. What data models are used in database management system? Discuss the distinguishing features of each.

14. What is the difference between a database and a database management system? For what reasons do organisations choose to invest in database management systems?
15. Explain the difference between a schema and a subschema, between a host language system and a self-contained system, and between an application program and a database management system.
16. What is the difference between a DBMS's physical structure and its logical structure? What are three popular logical views supported by DMBMSs? Prepare a simple schema for each type of logical view and explain how they differ.
17. Why is the current trend toward relational models in computer mainframe and microcomputer DBMSs?
18. Why is the use of relation databases growing? What advantages do they offer? What disadvantages do they present?
19. What operations are performed through relational database systems? explain the meaning and purpose of each operator.
20. Discuss the implications that a database would have on each phase of the systems development life cycle.
21. What is the relationship between the systems analyst and the database administrator?
22. A analysis of the proposed system's requirements indicated that a database system would give the best approach. Describe, with the aid of a relevant example in each case:
    (a) The advantages of the database approach compared with the use of separate files
    (b) How to keep the data accurate and consistent
    (c) How to obtain the factual information about breakdowns
    (d) How to prevent unauthorised access to data
23. Compare the advantages to the disadvantages of data base management system.
24. Describe data processing systems and their characteristics.
25. What are the sub-divisions of transaction processing?
26. What is a database?
27. What is a DBMS?
28. What is end-user computing?
29. What are DSS?
30. Where should DSS be applied?
31. What are the main types of DSS?
32. What types of packages can be of assistance in support of decision making?
33. What is a spread sheet package and how can it be used?

CHAPTER 2

TRADITIONAL DATA MODEL

2.1 Types of Data Models

Data models are usually categorized by levels of abstraction:

- Conceptual
- Logical
- Physical

These have no agreed formal definitions. Professional data modelers understand the approximate scope of each. These layers may appear in different ways. Some approaches deal only with the physical or logical model. Others offer elements of all three but not necessarily in three separate views.

2.2 Database Models

Besides differentiating databases according to function, databases can also be differentiated according to how they model the data.

A model defines the method of storing and retrieving data. A data model is defined as a set of guidelines for representing the logical organisation of data in the database; a pattern according to which data and relationships can be organised; an underlying mathematical formulation for building logical data organisations.

A data model consists of:

- A named logical unit (record type, data item)
- Relationships among logical units
- A data item is the smallest logical unit of data, an instance of which is known as a data item value.

- A record type is a collection of data items, and a record is hence defined as an instance of a record type.

Well, essentially a data model is a "description" of both a container for data and a methodology for storing and retrieving data from that container. Actually, there isn't really a data model "thing". Data models are abstractions, oftentimes mathematical algorithms and concepts. You cannot really touch a data model. But nevertheless, they are very useful. The analysis and design of data models has been the cornerstone of the evolution of databases. As models, have advanced so has database efficiency.

Before the 1980's, the two most commonly used Database Models were the hierarchical and network systems. Let us take a quick look at these two models and then move on to the more current models.

Note : A data model does not specify the data, data inplementation or physical organisation only the way it can be logically organised.

2.2.1 Model of a Model

A model is created to provide a small-scale representation of a subject in order to study the subject in depth.

When designing a database, the developer, in conjunction with the user, makes decisions about the data to be stored in the database and the structure and relationships of that data. From these decisions, the developer creates models for the user and for the system implementation staff. Ideally, the models represent the business from the user's view, not the real world view. Therefore, the created model is a model of what the user considers important in the business. This model depicts only objects and facts about the business, not processes, rules, or data storage and access methods.

- A representation of the user's world.
- It follows a set of rules.
- Includes business rules and database rules.

2.2.2 Business rules vs. Database rules

Business rules relate to how the business is conducted, in other words, rules regarding business transactions and processes. Sometimes these business rules can be enforced through the structure of the database. Often, however, these rules cannot be implemented in the DBMS or in the database; they must be written in application programs or enforced by user procedures.

Examples of business rules:

- The store is closed on Sunday.
- An employee can only work 40 hours a week.
- An invoice cannot be paid without an approved purchase order.
- Database rules concern the structure of the database. Generally, these rules can be implemented in the database and the DBMS.

Examples of database rules are as follows:

- Dates are formatted as MM/DD/YY (month/date/year).
- Invoice numbers range between 10000 and 99999.
- Sell price must be greater than 0.

2.3 MODELING: Three Schema Architecture

In general, modeling is based on the three-schema architecture. A Schema is a abstract definition of reality. The levels are:

- Level 0: Real World
- Level 1: Conceptual Model - E-R Model or Object-Oriented Model
- Level 2: Implementation Model - Relational Model
- Level 3: Physical Model - Physical Data Structures

In designing a database, we begin with the development of a conceptual model. A number of different conceptual modeling approaches are used including:

- Hierarchical (legacy)
- Network (legacy)
- Entity-Relationship (linked to relational model)
- Object-Oriented (generally converted to a relational model)

2.4 Conceptual level (or logical level)

- Conceptual view is the representation of the entire DB, of the data "as it really is", e.g. the logical model of the whole DB.
- Conceptual schema is a definition of the total DB.
- Security checks are defined so only allowed users obtain access e.g. passwords and grant permissions for each table.
- Integrity checks (validation) are also defined that ensure that users do not harm the correctness of the database, e.g. range checks on certain fields, referential integrity checks.
- Specifications should be data independent, with no reference to physical storage structure or access methods.
- Ideally, the conceptual schema describes the complete enterprise, including data flows from point to point, audit considerations etc.
- Once a good conceptual schema is developed, the rest is easy.
- Conceptual Data Definition Language (DDL) provides for creation and maintenance of the full schema.
- The Database Administrator is responsible for the maintenance of the conceptual schema.

2.4.1 Schema

Pronounce skee-ma, the structure of a database system, described in a formal language supported by the database management system (DBMS). In a relational database, the schema defines the tables, the fields in each table, and the relationships between fields and tables.

Schemas are generally stored in a data dictionary. Although a schema is defined in text database language, the term is often used to refer to a graphical depiction of the database structure.

The use of database management systems does not eliminate the need for computer programs. The database management system is a bridge between the application program, which determines what data are needed and how they are processed, and the operating system of the computer, which is responsible for placing data on the magnetic storage devices. A schema defines the database and a subschema defines the portion of the database that a specific program will use. (Typically, programs use only a section of the database.) To retrieve data from the database:

- The application program determines what data are needed and communicates the need to the database management system;
- The Database Management System determines that the data requested are in fact stored in the database (even though they may be stored under a different name-an alias). The data must be defined in the subschema (possible only if the data are in the database);
- The data base management system instruction the operating system to locate and retrieve the data from the specific location on the magnetic disk (or whatever device it is stored on); and
- A copy of the data is given to the application program for processing.

The Database Management system permits data independence, meaning that the application program can change without affecting the stored data. Under ordinary master file arrangements, if the program changes in such a way that the arrangement of the data retrieved or stored is modified, the master files must be recreated and restructured. With data independence, changes may occur in one data store or data use without affecting the other. A data dictionary is embedded in the data management system through the schema and subschema to ensure that data are properly defined and described and that duplicate names (aliases) do not result in redundant data storage or loss of data integrity.

The External schema describes the external view.

External schema is defined by the Data Definition Language (DDL)

The Data Manipulation Language (DML) is used to access data.

A formal definition of the logical structure of the database, often represented as an Entity-Relationship diagram.

The schema for the external and internal levels is kept by the database in its Data Dictionary, also know as the Catalogue or System Tables.

Internal level

Internal view is the representation of the actual storage of the DB. Internal schema defines the records, indexes, files and other physical attributes. For every conceptual file there may be several actual storage files which should be able to change, reorganise and optimize the physical characteristics without affecting conceptual view.

Fine tuning here affects all users (programs) that access the DB.

Internal Data Definition Language is used to write the internal schema. This language does not specify actual disk block sizes, disk pages etc. which is performed by the OS. So the database's internal level is one removed from the computer's physical level.

2.4.2 Mappings

DB needs two sets of mappings:

1. Between Conceptual/Internal Levels:
   - Defines how the conceptual schema is actually to be stored
   - Alterations to the internal level should be hidden from the conceptual view by updating the mappings
2. Between External/Conceptual Levels:
   - Defines how an external user sees the database
   - Eg. only certain fields from a table are visible,
   - Fields from different tables are combined as one single view
   - A new external view is created by specifying a new mapping
   - Alterations to one level may not necessarily have any impact on another level.
   - Eg: addition of a new field to the database (at the conceptual level) does not impact an external user as the external schema mapped for the user cannot see the new field anyway, but naturally, the new field needs to be mapped onto the internal database.

Similarly, addition of a new index for this new field at the internal level may significantly improve performance, but has no effect on the external schemas.

2.4.3 Schema Integration

Schema integration, as defined by the authors, occurs in two contexts:

1. View integration (in database design), which produces a global conceptual description of a proposed database; and
2. Database integration (in distributed database management), which produces the global schema of a collection of databases.

Schema Objects

A schema is a collection of logical structures of data, or schema objects. A schema is owned by a database user and has the same name as that user. Each user owns a single schema. Schema objects can be created and manipulated with SQL and include the following types of objects as shown below in table:

Table 2.1 Schema Objects

• Clusters • Database links • Stand-alone stored functions* • Indexes • Packages* • Stand-alone procedures* • Sequences • Snapshots*+ • Snapshot logs0 • Synonyms • Tables • Database triggers* • Views
* These objects are available only with ORACLE's procedural option. + These objects are available only with ORACLE's distributed option.

Other types of objects are also stored in the database and can be created and manipulated with SQL, but are not contained in a schema:

- Profiles
- Roles
- Rollback segments
- Tablespaces

Most of these objects occupy space in the database.

2.5 Overall System Structure

- Database systems are partitioned into modules for different functions. Some functions (e.g. file systems) may be provided by the operating system.
- Software Components include:
- File manager manages allocation of disk space and data structures used to represent information on disk.
- Database manager: The interface between low-level data and application programs and queries.
- Query processor translates statements in a query language into low-level instructions the database manager understands. (May also attempt to find an equivalent but more efficient form.)
- DML precompiler converts DML statements embedded in an application program to normal procedure calls in a host language. The precompiler interacts with the query processor.
- DDL compiler converts DDL statements to a set of tables containing metadata stored in a data dictionary.

In addition, several data structures are required for physical system implementation:
Data files: Store the database itself.

- **Data dictionary:** Stores information about the structure of the database. It is used heavily. Great emphasis should be placed on developing a good design and efficient implementation of the dictionary.
- **Indices:** Provide fast access to data items holding particular values.

Fig. 2.1 Shows these components.

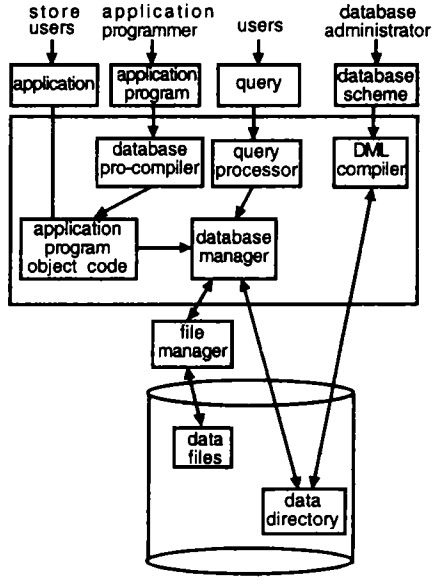


Fig. 2.1: Database System Structure.

2.6 The Hierarchical Model

The hierarchical model has some characteries ties:

- Similar to the network model.
- Organization of the records is as a collection of trees, rather than arbitrary graphs.

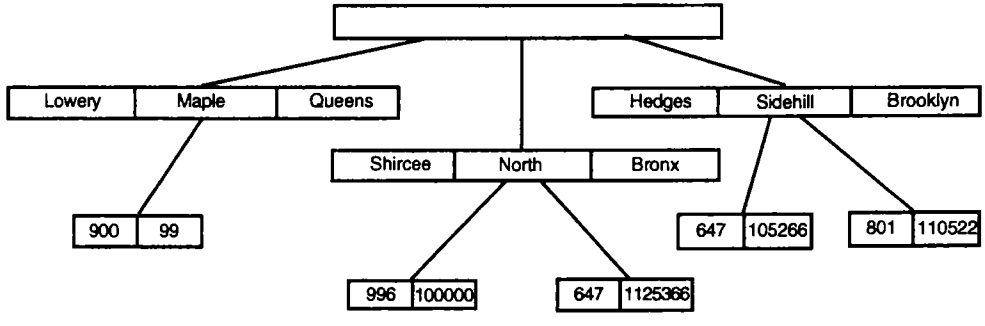


Fig. 2.2: A Sample Hierarchical Database

The hierarchical data model related entities by superior/subordinate or parent/child relationship. An organization chart, for example, shows the layers of executives, middle managers, and operating personnel. Graphically, the hierarchical data; model is shown as an upside-down tree, with the highest level of the tree known as the root. The nodes of the tree represent entities.

A hierarchical data model permits two of relationship:

- One-to-one — An entity at one level is related to one entity at the next level.
- One-to-many — An entity at one level is related to zero, one, or more entities at the next level.

The systems analyst is affected by the decision made when a hierarchical database is designed. during design, the database administrator, who is responsible for the design, determines the entities to be included in the database and the relationship that will exist between entities. The nodes represent instances of records containing the appropriate data items as determined by the data administrator.

The design of a hierarchical database will affect the accessibility of the data (the design tradeoffs need not be made when using relational databases).

The systems analyst must work within the design constraints that result. For example, the data model stipulates that items are accessible only through an occurrence of an order record. This relationship implies that the entire item list for all orders must be searched to prepare a report for the historical sale of items.

Anomalous (undesirable) side effects occur under certain database designs. Hierarchical databases involve anomalies with respect to the following:

- Insertion of records — A dependent record cannot be added to the database without a parent.

Example: Items cannot be added without their inclusion in an order.

- Deletion of records — Deleting a parent from the database also deletes all its descendants.

Example: deleting a customer also deletes outstanding orders.

If these situations are likely to occur in a particular application setting, it is necessary to establish multiple copies of the records, even multiple databases (adding redundancy and additional complexity) to avoid the problem.

2.7 The Network Model

- Data are represented by a collection of records.
- Relationships among data are represented by links.
- Organization is that of an arbitrary graph.

The relational model does not use pointers:

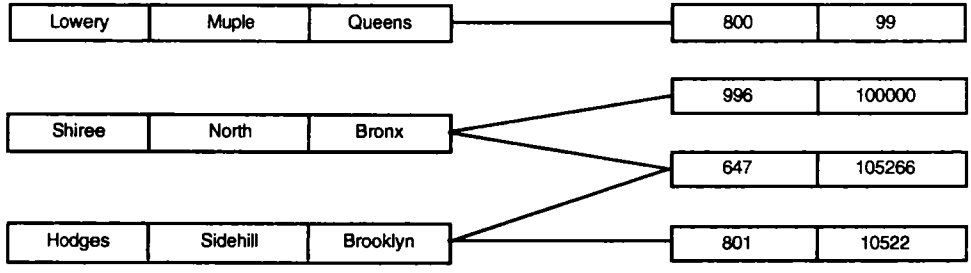


Fig. 2.3: A Sample Network Database

The network data model is similar to the hierarchical model, except that an entity can have more than one parent. Thus, as shown in Fig.2.3, members can belong to more than one relationship (i.e.,have more than one owner). In the laundry example, a relationship can be shown between customers and orders, as well as between orders and items. This capability introduces the use of an additional type of relationship in the data:

- Many-to-many — An entity can be related to zero, one, or more than one entity at another level.

In network databases, as in hierarchical databases, the relations between entities must be established at the time the data model is established and the database created (in contrast to the relational data model, which does not require predefined access paths or entity relations). The systems analyst must conform to these details when developing applications that enter or retrieve data during processing.

The hierarchical and network databases are conceptually simple and appear uncomplicated when first examined. In a large database environment, however, they can rapidly evolve into a complicated we of interrelationships that are difficult to manage as the database evolves with use.

Anomalies similar to those in the hierarchical data model occur. Thus, if an order is cancelled, we do not want to cancel the customer, although the model suggests this would happen.

There is a positive side to hierarchical and network data models that systems analysts should note. Suppose access paths and relationships between entities can be predefined (when the schema is developed and the database created). If the access and retrieval fined access paths, the processing of inquiries, updates, and additions to the database will be faster than when using relational databases.

Systems analysts also recognize that there are a great many databases based on the hierarchical and network models currently installed in the business community. If these databases are meeting operational requirements, it is unlikely that organizations will replace them. Thus, developing information systems applications that take into account the opportunities and constraints they offer is a necessity.

2.8 The Relational Model

The relational data model is currently the most popular one in database management systems because it is conceptually simple and under standable by information systems professionals and many end=users; it can evolve, since relationships need not be predefined; and it uses data values to imply relationships. The relational data model, developed in 1970 by E.F. Codd, is based on a relation, a two-dimensional table. Rows in the table represent the records; and columns show the attributes of the entity (Fig.2.4). Relational databases use a model to show how data in a record are logically related.

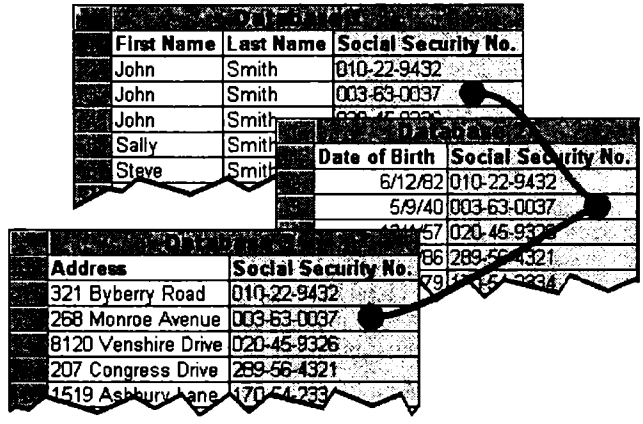


Fig. 2.4: Relational Database

The order of the data in the table is not significant and no order is implied when records are included in the relation. Similarly, the physical details of storage (whether random, indexed, or sequential organization) are not of concern to the analyst. Relational table shown logical, not physical relationships.

When a request for information is made, the system produces a table containing the information. In the laundry example, if a manager wishes to determine who uses napkins, the systems will produce a table containing the names of all users of napkins.

Data items are individual units of information, e.g. name, address, number of items in inventory, weekly sales total, grade in a course. A record normally consists of a set of related data items, e.g. an employee record would consist of various data items concerning a given employee. A file is a unified set of data which is usually in the form of a collection of records. (Not all files are subdivided into records, e.g. the government may keep a file consisting of all information on a particular airplane disaster. Such a file is, in effect, a single record.) A data base is a structured set of files.

2.8.1 Relation

Each record has a set of attributes (fields or items). The range of possible values (domain) is defined for each attribute.

Records of each type form a table or relation. In a table, each row is a record or tuple and each column is an attribute or field

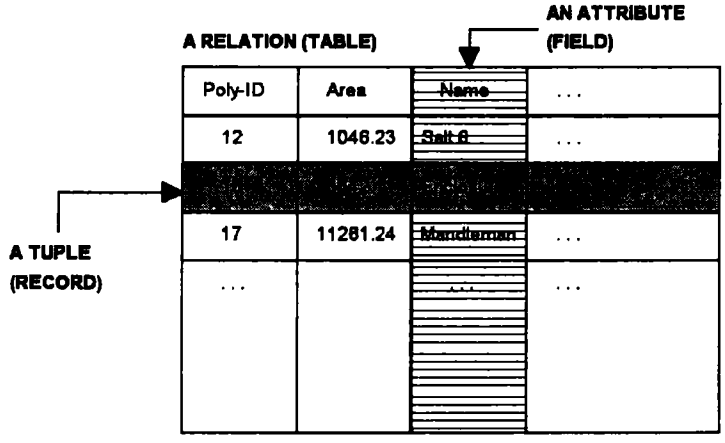


Fig. 2.5: Relation, Tuple and Field.

The degree of a relation is the number of attributes in the table. A one-attribute table is a unary relation. A two-attribute table is a binary relation. A n-attribute table is an n-ary relation.

Paddocks-ID
12
13
17
20

unary

Paddocks-ID	Name
12	Salt 6
13	Salt 5
17	Mandleman
20	Saloon

binary

Paddocks-ID	Name	Area
12	Salt 6	1046.23
13	Salt 5	1376.90
17	Mandleman	11261.24
20	Saloon	4761.74

ternary

Fig 2.6: The degree of a relation.

Keys

A key of a relation is a subset of attributes with the following properties:

- Unique identification: the value of a key is unique for each tuple.
- Non-redundancy: no attribute in the key can be discarded without destroying the key'suniqueness.

- A prime attribute of a relation is an attribute which participates in at least one key and all other attributes are non-prime.

The relationship of character, item, record, file and database is given in Fig 2.7. Some additional data structures used in more complex files will be described in a later chapter.

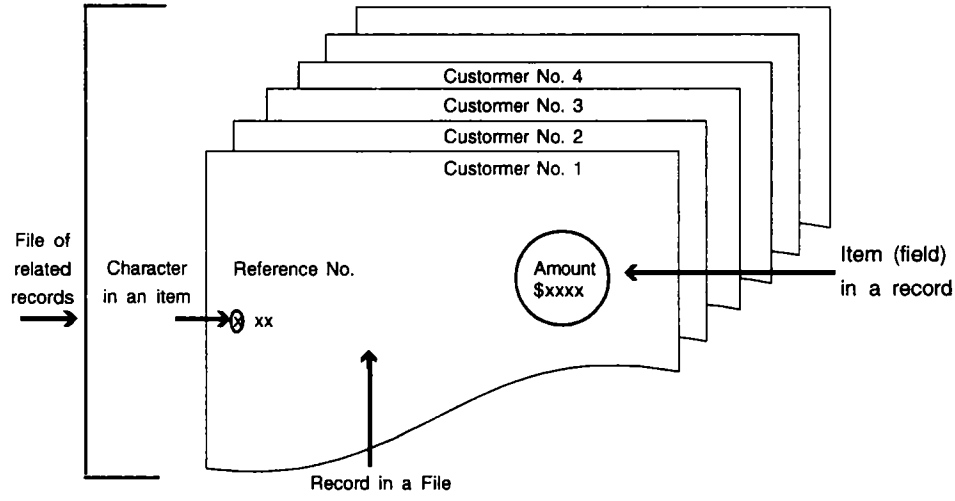


Fig. 2.7: Relatiaonship of character, item, record and file

Data are simply values or sets of values. A data item refers to a single unit of values. Data items that are divided into subitems are called group items; those that are not are called elementary items. For example, an employee's name may be divided into three subitems-first name, middle initial and last name-but the social security number would normally be treated as a single item.

An entity is something that has certain attributes or properties which may be assigned values. The values themselves may be either numeric or nonnumeric. For example, the following are possible attributes and their corresponding values for an entity, an employee of a given organization:

```
Attributes: Name Age Sex GPF NO
Values: JAYANT KUMAR 34 M PAOIND2/15
```

Entities with similar attributes (e.g., all the employees in an organization) from an entity set. Each attribute of an entity set has a range of values, theset of all possible values that could be assigned to the particular attribute.

The term "information" is sometimes used for data with given attributes, or, in other words, meaningful or processed data.

The way that data are organized into the hierarchy of fields, records and files reflects the relationship between attributes, entities and entity sets. That is, a field is a single elementary unit of information representing an attribute of an entity, a record is the collection of field values of a given entity and a file is the collection of records of the entities in a given entity set.

Each record in a file contain many field items, but the value in a certain field may uniquely determine the record in the file. Such a field is called a primary key.

Data and relationships are represented by a collection of tables. Each table has a number of columns with unique names, e.g. customer, account.

name	street	city	member
Lowery	Maple	Qneena	900
Shiver	North	Bronx	556
Shiver	North	Bronx	647
Hodger	Sidehiil	Brooklyn	801
Hodger	Sidehill	Brooklyn	617

name	balance
900	55
556	100000
647	105366
801	10533

Fig. 2.8: A Sample Relational Database.

2.8.2 Relational Database Management System (RDBMS)

A collection of integrated services which support database management and together support and control the creation, use and maintenance of relational databases. Servers as defined in this MIB provide the functions of the RDBMS.

2.9 Data Definition Language (DDL)

DDL is used to specify a database scheme as a set of definitions.

DDL statements are compiled, resulting in a set of tables stored in a special file called a data dictionary or data directory.

The data directory contains metadata (data about data). The storage structure and access methods used by the database system are specified by a set of definitions in a special type of DDL called a data storage and definition language.

2.9.1 Data Manipulation Language (DML)

1. Data Manipulation is:
   - Retrieval of information from the database;
   - Insertion of new information into the database;
   - Deletion of information in the database; and
   - Modification of information in the database.
2. A DML is a language which enables the users to access and manipulate data.

The goal here is to provide efficient human interaction with the system.

3. There are two types of DML:
   - Procedural: In this type of DML, the user specifies what data is needed and how to get it.
   - Nonprocedural: The user only specifies what data is needed. It is easier for the user. However, it may not generate code as efficient as that produced by procedural languages.
4. A query language is a portion of a DML involving information retrieval only. The terms DML and query language are often used synonymously.

CHAPTER 3

RELATIONAL DATABASE

3.1 RDBMS

Short for Relational Database Management system and pronounced as separate letters, a type of Database Management System (DBMS) that stores data in the form of related tables. Relational databases are powerful because they require few assumptions about how data is related or how it will be extracted from the database. As a result, the same database can be viewed in many different ways.

An important feature of relational systems is that a single database can be spread across several tables. This differs from flat-file databases, in which each database is self-contained in a single table.

Almost all full-scale database systems are RDBMS's. Small database systems, however, use other designs that provide less flexibility in posing queries.

Relational databases use a model intended to greatly simplify the end-users' and programmers' view of a database.

As shown in Fig. 3.1, files are seen as simple table, also known as relations. The rows are record occurrences–also called tuples–and the columns are fields, also called domains. Relationships are handled quite differently from the way they are in hierarchical and network models. Instead of physically denoting relationships by using pointers, location, or indexes, true relational databases do not store relationships. Instead, relationships are inferred when needed. Here it is imperative to question how this happens. Relationships are determined from intentionally redundant fields, usually keys, common to the different tables. This concepts requires further explanation.

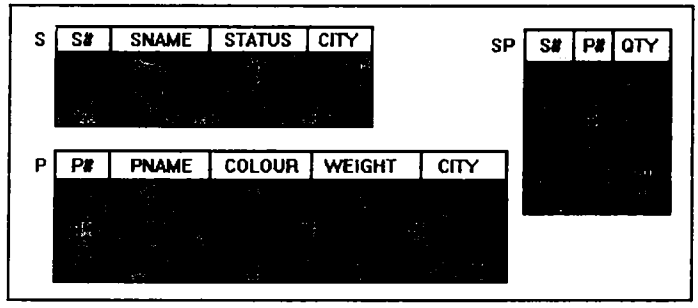


Fig. 3.1: Relational Data Structure

The DML for a relational DBMS doesn't navigate paths and pointers, which is what happens with hierarchical and network databases. Instead, to write reports and answer inquiries, the DML lets the programmer or end-user perform simple table operations to create temprorary tables. These operations include

- Selecting specific records from a table and creating a new, but temporary, table that contains only those occurrences. Criteria can be set to determine which records to select from the initial table.
- Projrcting out specific fields from a table, creating a temporary table that has fewer fields.
- Joining two or more table across a common field (this is the same as navigating relationship paths in a hierarchical or network database). Again, a temporary table is created.

In all cases, the above relational operations–which, unfortunately, have different names in the varioius relational DBMSs on the market–create temporary, working tables that will go away when you exit the program. If changes are made to the data in those tables, the changes will be updated into the permanently stored tables before the working tables are discarded (assuming that the end-user has update authority).

Also, note that the relational commands can be combined. For instance, let us say we need a table of orders for certain customers, perhaps to produce a report. First, we can use SELECT on the CUSTOMER table to create a working table of those customers needed. Second, we can PROJECT out only those fields needed in both the CUSTOMER and ORDER working table to reduce their size. Finally, we JOIN the two working tables to give us the final working table needed for the report.

Relational databases are definitely the trend. They present a simpler view-point to both programmers and end-users. Query language and report writers that are easy to learn and use have been built around relational databases. Most relational databases are currently converging to supersets of a de facto standard language called SQL that is utilized to create, update, and use tables. SQL includes the SELECT, PROJECT, and JOIN operations discussed earlier.

Examples of relational DBMSs include IBM's DB2 and SQL/DL, ADRs DATACOM, Relational Technololgy's Ingress, Oracle Corporation's Oracle, and Information Builder's FOCUS (which

is also hierarchical). Additionally, most microcomputer DBMSs are relational. Examples include Ashton-Tate's dBASE-IV (which recently embraced true SQL relational standards) and Microrim's R:BASE for DOS. Many relational dBMSs are offered in both mainframe and micro versions. FOCUS is o ne such DBMS.

That's about all we want to say about the three common data models database text bookis and courses will offer entire chapters and units on each of the three models. Let us consider a simplified translation of a logical data model into each of the three physical data structures that we have introduced. But first we need to describe the roles of the database design participants.

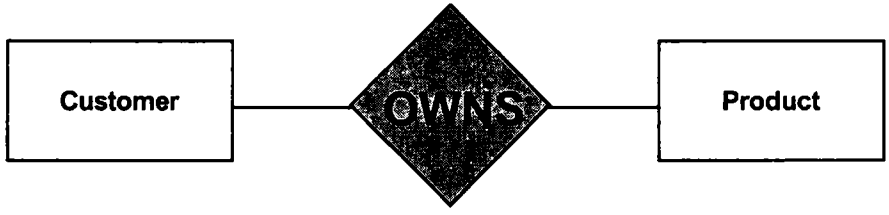


Fig. 3.2: Relational Data Structure

A relational DBMs depicts record types as simple tables, much like those seen in spreadsheets that exhibit relational-like qualities. Tables are related to one another via intentionally redundant fields, usually keys. Relational DBMSs provide operators that build temporary working tables from the tables illustrated that are physically stored.

3.2 The Relational Model

The relational model was the first example of a formal data model. With it, the user data is represented and manipulated intuitively. The model also uses techniques that help developers spot and correct possible design problems as data is prepared for implementation to the DBMS. Relational models involve:

- Determining data structure; and
- Data is stored in a structure of relations (tables) defined by a data definition language (DDL). The elements of data structure used in relational models are relations, attributes, tuples, and domains.

3.3 The Relational Database Model

The relational database model has become the de-facto standard for the design of databases both large and small. While the concepts involved are not terribly complex, it can be difficult at first to get a handle on the concept. Let us dwell on a little introduction to the following topics:

- Primary and Foreign Keys;
- Queries;
- Structured Query Language (SQL);

- Referential Integrity; and
- Normalisation.

The simplest model for a database is a flat file. You have only a single table that includes fields for each element you need to store. Nearly everyone has worked with flat file databases, at least in the form of spreadsheets. The problem with flat files is that they waste storage space and are problematic to maintain. Let us consider the classic example of a customer order entry system. Assume that you are managing the data for a company with a number of customers, each of which will be placing multiple orders. In addition, each order can have one or more items.

Before moving on, let us describe the data that we wish to record for each component of the application:

Customers

- Customer Number
- Company Name
- Address
- City, State, ZIP Code
- Phone Number

Orders

- Order Number
- Order Date
- PO Number

Order Line Items

- Item Number
- Description
- Quantity
- Price

It doesn't take a database design expert to see what the problem is in using a flat file to represent this data. Each time an order is placed, you will need to repeat the customer information, including the Customer Number, Company Name, etc. What is worse is that for each item, you not only need to repeat the order information such as the Order Number and Order Date, but you also need to continue repeating the customer information as well. Let us say there is one customer who has placed two orders, each with four line items. To maintain this tiny amount of information, you need to enter the Customer Number and Company Name eight times. If the company should send you a change of address, the number of records you need to update is equal to the sum of product of orders and order line items. Obviously, this will quickly become unacceptable in terms of both the effort required to maintain the data and the likelihood that at some point there will be data entry errors and the customer address will be inconsistent between records.

The solution to this problem is to use a relational model for the data. Do not let the terminology get you down – the concept isn't that hard to understand. It simply means that in this example each order entered is related to a customer record, and each line item is related to an order record. A Relational Database Management system (RDBMS) is then a piece of software that manages groups of records which are related to one another. Let us take our flat file and break it up into three tables: Customers, Orders, and OrderDetails. The fields are just as they are shown above, with a few additions. To the Orders table, we will add a Customer Number field, and to the OrderDetails table we will add an Order Number field. Here's the list again with the required additional fields and modified field names.

Customers

- CustID
- CustName
- CustAddress
- CustCity
- CustState
- CustZIP
- CustPhone

Orders

- OrdID
- OrdCustID
- OrdDate
- OrdPONumber

OrderDetails

- ODID
- ODOrdID
- ODDescription
- ODQty
- ODPrice

I will digress here a little to describe how we came up with the field names. This is totally a personal preference. Every time we created a table, we have decided on a field name prefix to be used. As you can see, we have used "Cust" for the Customers table, "Ord" for the Orders table, and "OD" for the OrderDetails table. We have done this for a couple of reasons. First, by using a field name prefix, we can identify the table associated with a field by simply reading the field name. Second, We have avoided name collisions in queries because every field in a database, no matter how large and complex the structure, will always be unique. This is we have done particularly helpful in distinguishing between primary and foreign keys – a subject I will take up a bit later. One other thing you may have noticed is a table naming convention we have used, which is to name tables using the plural form of the data they contain.

OK, back to the tables. What we have done besides the name change is to add fields to the Orders and OrderDetails tables. Each have key fields used to provide a link to the associated Customers and Orders records, respectively. These additional fields are called foreign keys.

3.4 Relational Modeling Techniques

After an object model is developed, a logical data model can be created to further define the user's business data. To build the logical data model, the elements of the object model (such as the E-R diagram described in Lesson 3) are converted into logical data tables. Remember, the term logical is used because the tables may or may not be used as physical database tables in the final design. As logical tables are defined, business policies and rules are applied. This lesson will explore the concepts of the relational model as a logical data modeling tool. Normalisation, a technique applied to the relational model, will also be discussed. Some examples in this lesson will be clarified with screen illustrations from Microsoft's Access DBMS.

The term relational comes from the mathematical field of relational algebra. Relational models are composed of relations, or two-dimensional tables, which follow the operations described in relational algebra. With this model, tables of data can be manipulated to return other tables of data providing users with information. All relational database structures are composed of a series of relations. The three aspects of data addressed by relational models are data structure, data integrity, and data manipulation.

The relational model was devised in 1969 by Dr. E.F. Codd, a mathematician associated with IBM's San Jose Research Laboratory. His model served as an introduction to a relational database structure that was designed to represent user data in a form easily understood by both users and professionals without regard for physical implementation. The relational theory was explored and prototypes developed during the 1970s. Commercial relational database products began to emerge in the 1980s, originally for mainframe systems and later for microcomputers.

3.5 Components of the Relational Model

When building a relational model, developers and users must create tables from elements of the object model, determine each table's attributes, and establish relationships between the tables. As with other data modeling concepts, relational modeling has many interchangeable terms for its components as shown below:

Relation, table, file

A two-dimensional table consisting of columns and rows; created from the entities of the object model.

Attribute, column, field

The columns of the table, usually defined from the attributes of the object model.

Tuple, row, record

The rows of the table; derived from the entity occurrences of the object model.

Consider the relation STUDENT. The column headings signify the attributes of the relation.

RELATION = STUDENT

StuID	StuName	Age	Major
1001	Jones	21	Accounting
1005	Phillips	18	Science
1006	Stevens	20	Art
1010	Barber	18	Business

- **Each cell is atomic.**
- **Each attribute contains the same type of information.**
- **Each tuple is unique.**

Fig. 3.3: Relations

Relations

A relation is a two-dimensional table containing a set of related data. The true requirements of a relation are that:

- Each cell must be atomic (contain only one value);
- Each attribute contains the same type of physical and semantic information in all its cells.;
- Each tuple is unique; there can be no duplicate rows of data; and
- The order of columns and rows is not significant.

3.6 Definitions of Relational Terms

In addition to the above components, the following concepts are used in building a relational model. Notice the definitions for various types of keys. A key is an identifying property of a model and/or a database. However, the term is ambiguous when used without a modifier (i.e., primary, candidate, foreign). To eliminate confusion in modeling, the type of key should always be stated according to purpose.

Primary key

Primary keys are essential in relational modeling; one should be specified for each relation. A primary key uniquely identifies a record (or row) in a table; in other words, a particular primary key value returns a record that is identical to no other. A primary key is composed of one column (simple primary key) or a combination of columns (composite primary keys) that provide this unique identification. The best possibilities for primary keys are attributes that seldom change and are familiar to users. The primary key should contain the fewest columns needed to uniquely identify a record.

Simple primary key:

Consider the relation ADVISOR where each value of Advisor ID returns a unique record. Simple primary key Advisor-ID uniquely identifies records.

RELATION = ADVISOR

Advisor ID	Adv Name	Adv Phone
101	Brown	333-2111
102	Williams	405-8888
103	Benson	501-8241
104	Smith	222-2357

Fig. 3.4

Composite primary key:

Consider the relation STUDENT where each student can take more than one class. Student ID, Class, or Grade alone does not return a unique record; however, a composite primary key of Student ID- Class does.

RELATION = STUDENT

Student ID	Class	Grade
1001	INSY312	95
1001	ENGL101	90
1005	INSY312	85
1006	INSY430	87
1006	BUSN202	80
1010	HIST102	92

Fig. 3.5

Candidate key

A candidate key is a unique identifier that might be considered when choosing the primary key of a relation. Candidate keys can also be simple or composite. Usually one candidate key is designated as a primary key, while the others may be called alternate keys.

Foreign key

A foreign key is an attribute in a relation that is also a primary key in another relation. This foreign key-primary key match allows references between relations in order to locate information. For example, the attribute StudentID could be designated as a primary key in the relation ENROLLED STUDENT and as a foreign key in the relation INDIVIDUAL'S RECORD. This provides a reference to INDIVIDUAL'S RECORD information without placing duplicate information in both relations. Primary keys and their corresponding foreign keys must share the same physical and logical domains.

3687

- **Relation ADVISOR where AdvID is the primary key.**
- **Relation STUDENT where StuID is the primary key and AdvID is a foreign key.**

RELATION = ADVISOR		
AdvID	**AdvName**	**AdvPhone**
101	Brown	333-2111
102	Williams	405-8888
103	Benson	501-8241
104	Smith	222-2357

RELATION = STUDENT			
StuID	**StuName**	**StuPhone**	**AdvID**
1001	Jones	452-3311	104
1005	Phillips	555-1450	102
1006	Nelson	352-0913	101
1010	Terrell	452-5974	102

Fig. 3.6

Functional dependency

A functionally dependency is an association between the columns of a relation. This means that a given value of one column can determine a unique value from another column. Functional dependency is used in the normalisation technique in order to simplify the structure of relations.

Index

Indexes are used to sort and access the records in a relation. Not to be confused with a primary key, an index is overhead data that is specified when constructing the actual physical database. They provide the DBMS with a quick method of retrieving and accessing data elements. Indexes, unlike primary keys, do not have to be unique.

Domain

The concept of domains is used to add meaning to the attributes of a relation. A domain is the physical and logical description of an attribute. A physical domain identifies the format, such as field type and length, of a column, while a logical domain identifies its valid pool of values. Domain names can be used to represent the meaning of a column and can be used more than once in a relation or in other relations. Each column of a relation should have both physical and logical domains.

3.7 Features of Relational Databases

Data are organized as logically independent tables. Features:

- 'Natural'
- Not so strongly biased towards specific questions
- Expresses relationships by means of redundant data rather than explicit pointers
- Theoretical basis: relational algebra, calculus; closure
- Operations on tables (Join, Project, Select) to form new tables.

3.8 CODD's 12 Rules for a Fully Relational DBMS

According to Elmasri and Navathe (1994), Dr. E. F. Codd, the originator of the relational data model, published a two-part article in ComputerWorld (Codd, 1985) that lists 12 rules for how to determine whether a DBMS is relational and to what extent it is relational. These rules provide a very useful yardstick for evaluating a relational system. Codd also mentions that, according to these rules, no fully relational system is available yet. In particular, rules 6, 9, 10, 11, and 12 are difficult to satisfy.

Rule 1: The Information Rule

All information in a relational database is represented explicitly at the logical level in exactly one way-by values in tables.

Rule 2: Guaranteed Access Rule

Each and every datum (atomic value) in a relational database is guaranteed to be logically accessible by resorting to a table name, primary key value, and column name.

Rule 3: Systematic Treatment of Null Values

Null values (distinct from empty character string or a string of blank characters and distinct from zero or any other number) are supported in the fully relational DBMS for representing missing information in a systematic way, independent of data type.

Rule 4: Dynamic On-line Catalog Based on the Relational Model

The database description is represented at the logical level in the same way as ordinary data, so authorized users can apply the same relational language to its interrogation as they apply to regular data.

Rule 5: Comprehensive Data Sublanguage Rule

A relational system may support several languages and various modes of terminal use (for example, the fill-in-blanks mode). However, there must be at least one language whose statements are expressible, per some well-defined syntax, as character strings and whose ability to support all of the following is comprehensible: data definition, view definition, data manipulation (interactive and by program), integrity constraints, and transaction boundaries (begin, commit, and rollback).

Rule 6: View Updating Rule

All views that are theoretically updateable are also updateable by the system.

Rule 7: High-level Insert, Update, and Delete

The capability of handling a base relation or a derived relation as a single operand applies nor only to the retrieval of data but also to the insertion, update, and deletion of data.

Rule 8: Physical Data Independence

Application programs and terminal activities remain logically unimpaired whenever any changes are made in either storage representation or access methods.

Rule 9: Logical Data Independence

Application programs and terminal activities remain logically unimpaired when information preserving changes of any kind that theoretically permit unimpairment are made to the base tables.

Rule 10: Integrity Independence

Integrity constraints specific to a particular relational database must be definable in the relational data sublanguage and storable in the catalog, and not in the application programs.

A minimum of the following two integrity constraints must be supported:

1. Entity integrity: No components of a primary key is allowed to have a null value.
2. Referential integrity: For each distinct nonnull foreign key value in a relational database, there must exist a matching primary key value from the same domain.

Rule 11: Distribution Independence

A relational DBMS has distribution independence. Distribution independence implies that users should not have to be aware of whether a database is distributed.

Rule 12: Nonsubversion Rule

If a relational system has a low-level (single-record-at-a-time) language, that low-level language cannot be used to subvert or bypass the integrity rules or constraints expressed in the higher-level (multiple-records-at-a-time) relational language.

Note: There is a rider to these 12 rules known as Rule Zero: "For any system that is claimed to be a relational database management system, that system must be able to manage data entirely through its relational capabilities."

On the basis of the above rules, there is no fully relational DBMS available today.

3.9 The Relational Model and Relational DBMS

Although it is not the only modeling technique, since its introduction relational modeling has become a standard in the data modeling field. One reason is because the data is stored and presented in an understandable way. As a result, a system based on this type of model requires less professional systems support. With today's sophisticated applications and a well-structured system, the user can specify the criteria for needed information with a few simple commands, while the system performs the complicated operations required to provide the information. Also, since relational modeling separates data from physical structures, a stable data foundation is provided for relational systems. Data retrieval, updates, processing, and structural changes are more efficient in this type of system.

However, relational modeling does have its critics. Relational systems require more resources from the computer and are slower to process than previous systems. This presented a problem in the days when systems were more expensive and less powerful. The huge storage capacities and fast processors of today's computers provide the power needed to efficiently process relational database systems. Also, database developers have been slow in adopting the techniques of relational modeling. Remember, originally business systems modeled business processes and were

programming-intensive. Relational techniques focus on data, which requires a new way of thinking for developers. Even though most admit that relational systems are more flexible and easier to use, some professionals have balked at the changes.

Information systems formerly used, such as inverted list, hierarchical, and network systems are not based on relational concepts. Relational database management systems (RDBMS) are based on Codd's relational model. While a majority of the DBMS products currently in use claim to be relational, none of these systems exactly fits the true description of a RDBMS. This description can be found in Codd's "Twelve Rules of a Relational Database."

3.10 Fully Relational DBMS

An implementation of a relational database that supports all aspects of the relational model, including in particular domains and the two general integrity rules, is said to be fully relational. However, no system is fully relational at present, though several systems are beginning to become close.

3.10.1 Relational Implementations

An implementation of the relational model should support all the facilities prescribed by the model, but there are other desirable features of such systems which the relational model does not refer to - these include arithmetic operations, built-in functions, explicit updates, modifies, deletions etc. The relational model provides a core of functions that should be incorporated.

No system supports the relational model in its entirity though several come close; however a distinction must be made between models which are considered relational and relational-like models. The relational-like models can be categorised as follows.

- Tabular;
- Minimally relational;
- Relationally complete; and
- Fully relational.

3.11 Primary and Foreign Keys

Like most other programming disciplines, relational databases are ripe with jargon. A key is simply a field which can be used to identify a record. In some cases, key fields are a part of the data you are storing or derived from that data, but they are just as often an arbitrary value. For the Customers table, you could use the company name as a key, but if you ever had two companies with the same name, your system would be broken. You could also use some derivation of the company name in an effort to preserve enough of the name to make it easy for users to derive the name based on the key, but that often breaks down when the tables become large. I find it easiest to simply use an arbitrary whole number. You can completely hide the use of the numbers from the end users, or expose the data. Its your choice to make based on the needs and abilities of the users.

There are two types of key fields we are dealing with:

- Primary keys
- Foreign keys

A primary key is a field that uniquely identifies a record in a table. No two records can have the same value for a primary key. Each value in a primary key will identify one and only one record. A foreign key represents the value of primary key for a related table. Foreign keys are the cornerstone of relational databases. In the Orders table, the OrdCustID field would hold the value of the CustID field for the customer who placed the order. By doing this, we can attach the information for the customer record to the order by storing only the one value.

3.11.1 Primary/Candidate Keys

The primary key of a relation or table is actually a special case of a more general construct, namely the candidate key. A candidate key is an attribute that is a unique identifier within a given table. One of the candidate keys is choosen to be the primary key and the others are called alternate keys. Usually, there is only one candidate key anyway.

Primary keys provide the sole tuple-level addressing mechanism within the relational model - the only guaranteed way of pinpointing an individual tuple and thus they are fundamental to the operation of the overall relational model.

Tuples

A tuple of a relation or table corresponds to a row of such a table. Tuples are unordered (top to bottom) because a relation is a mathematical set and not a list. There are no duplicate tuples in a relation/table which again stems from the fact that a relation is a mathematical set and sets in mathematics by definition do not include duplicate elements.

An important corollary of this point is that the primary key always exists and since tuples are unique it follows that at least the combination of all attributes of the relation has the uniqueness property, so that at least the combination of all attributes can serve as the primary key - but it is usually not necessary to involve all of the attributes i.e. some lesser combination is usually sufficient.

3.11.2 Atomic Data Values

All data values in the relational model are atomic. This implies that at every row/column position in every table there is always exactly one data value and never a set of values (at the external/ conceptual levels).

3.12 Relationships in the Relational Model

As tables are created, the relationships between them must also be defined. Relationship cardinality determines how tables should be joined in the database. Relationship modality can also be illustrated with the same signs shown on the E-R diagram.

3.12.1 One-to-one relationships

In a relational model, the one-to-one (1:1) relationship is easily illustrated. In describing two related tables, the primary key of one table is placed as a foreign key in the other. For example, in representing a 1:1 relationship between Table A and Table B, the primary key of Table A is specified as a foreign key in Table B. However, there are no hard and fast rules governing this designation; the primary key from either table can be placed as a foreign key in the other. One consideration might be whether one relation is accessed more often than the other, which could affect database performance.

3.12.2 One-to-many relationships

A one-to-many (1:N) relationship is usually represented using a "parent-child" concept. Here, the "one" side of the relationship is called the parent, while the "many" side is designated as the child. In describing this to the relational model, the primary key of the parent entity is always placed as a foreign key in the child entity.

Example of a 6:1 Relationship link

Consider the following E-R diagram of a 6:1 relationship in which each class has a class roster and each class roster is for a class:

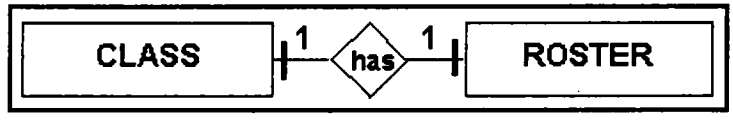


Fig. 3.8

The following data structure diagram can be used to illustrate the above relationship in a relational model. In the 1:1 relationship, the primary key of one relation is designated as a foreign key in the other. This primary key-foreign key link can be used to look up values between tables.

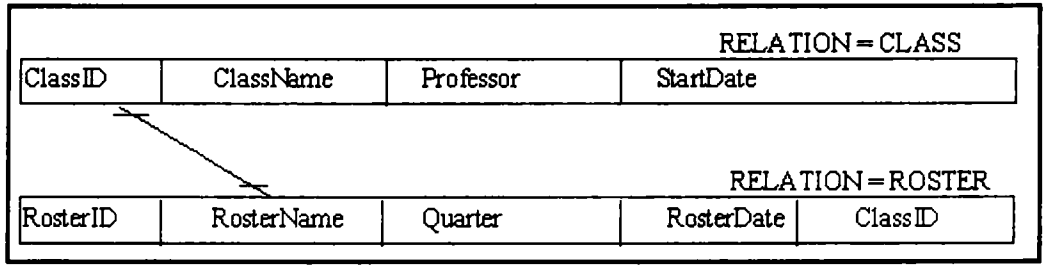


Fig. 3.9

However, the relationship could also be represented as follows:

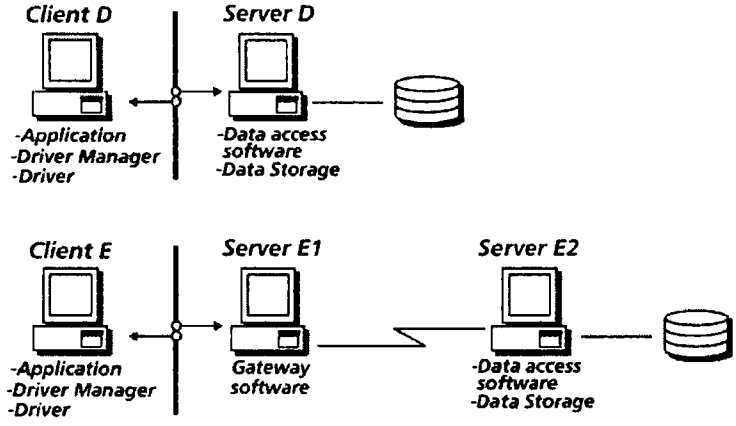


Fig. 3.10

The primary key can be designated either way, although the designer might consider whether one table is accessed more than the other. Still, ClassID can be used to look up values in the ROSTER relation, or Roster ID can be used to look up values in the CLASS relation.

Also notice that the modality is illustrated with bars drawn across the connecting line, as in the E-R diagram.

Example of a 6:2: N Relationship link

Consider this E-R diagram of a 6:1:N relationship in which a class can be offered more than once a quarter:

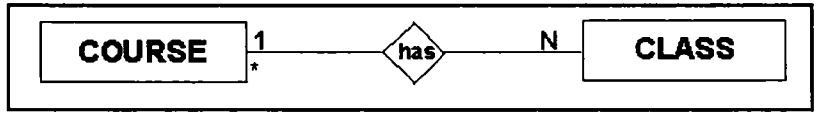


Fig. 3.11

Because it represents the "1" side of the relationship, COURSE is considered the parent entity while CLASS (the "M" side) is the child entity. The fork at the end of the connection line denotes "many." Remember, when building a relational model for this type of relationship, the primary key of the parent is always placed as a foreign key in the child. In this diagram, the primary key of COURSE (CourseID) is placed in the relation CLASS as a foreign key. Then the attribute CourseID can be used to look up values in the relation CLASS. For example, if the course INSY410 is offered on Tuesday and Thursday nights, CourseID INSY410 would return two records of class information from CLASS.

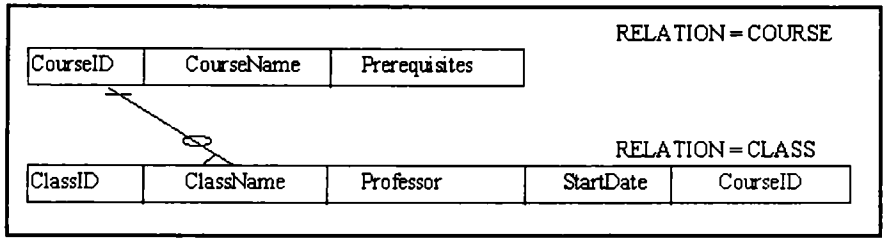


Fig. 3.12

To prove why this cannot be reversed, suppose the primary key of CLASS was placed as a foreign key in COURSE. Since a course can have more than one class, the relation COURSE would contain duplicate values in the primary key. This is unacceptable because a primary key field must have unduplicated values in order to a unique record.

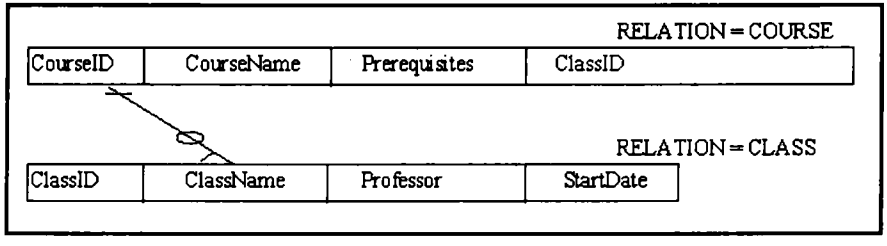


Fig. 3.13

3.12.3 Many-to-Many Relationships

Because many occurrences can exist on either side, the representation of a many-to-many (M:N) relationship is totally different from the 1:1 and the 1:N. To illustrate this type of relationship, a new relation, called an intersection table, is formed. It consists only of the primary keys of the first two relations.

Example of a M:N relationship link

Consider this E-R diagram of a M:N relationship in which a student can take many classes each quarter, and a class contains many students:

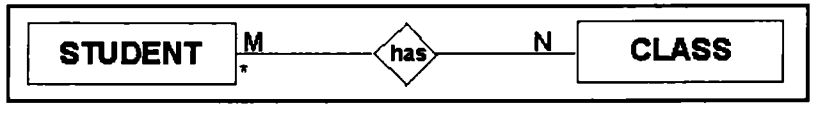


Fig. 3.14

If the primary key of STUDENT (StudentID) were placed as a foreign key in CLASS, the primary key of CLASS would be duplicated. This would also be true if ClassID were placed as a foreign key in STUDENT.

RELATION=CLASS

ClassID	ClassName	Professor	StartDate	StudentID
ENGL101	Composition I	Hall	9/25/96	101
HIST211	U.S. History	Bright	9/26/96	101
SCIE113	Geology	Hill	9/23/96	105
ENG101	Composition I	Hall	9/25/96	110
SCIE113	Geology	Hill	9/23/96	101

Fig. 3.15.

In the relational model, M:N relationships are represented by creating intersection tables containing only the primary keys of both relations as attributes. In this way, both original relations (STUDENT and CLASS) maintain unique primary key values.

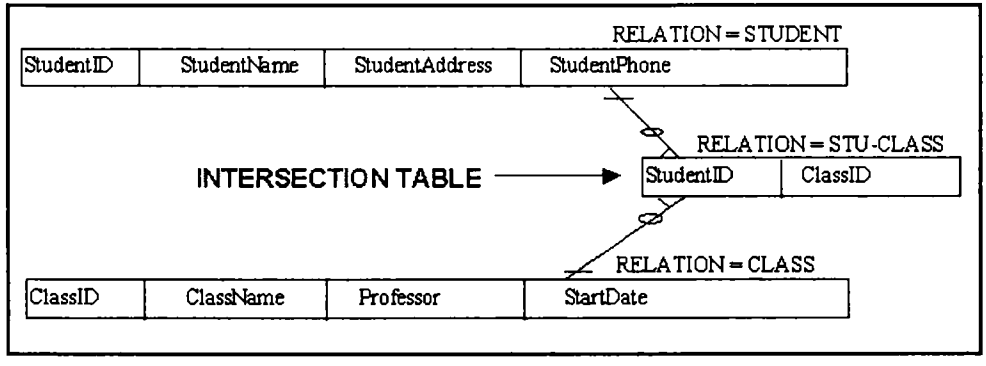


Fig. 3.16

3.12.4 Defining the Relational Model to the Database

Once the tables and their relationships have been defined, the database structure must be described to the database application. How this is done depends on whether the database application has a graphical or a textual interface. An application such as Access provides "fill-in-the-blank" tables to input structural definitions or "wizards" that prompt the user (or developer) to describe the database structure. Applications with a textual interface require the use of a data definition language (DDL) to write a text file that describes the database structure.

3.12.5 Example of Data Definition Language

The definition of the database structure to the database application depends on which type of application is used - graphical or textual. Following is an example of a text-based data definition source as described to an application designed for an AS/400 minicomputer.

STUDENT PERSONNEL FILE		
A	UNIQUE	
A	R STUDENT	TEXT('STUDENT RECORD')
A*		
A	SS# 9S 0	TEXT('SS NUMBER')
A	ADDR1 25	TEXT('STREET ADDRESS 1')
A	ADDR2 25	TEXT('STREET ADDRESS 2')
A	CITY 25	TEXT('CITY')
A	STATE 15	TEXT('STATE')
A	ZIP1 5	TEXT('ZIP 1')
A	ZIP2 4	TEXT('ZIP 2')
A	STATUS 1	TEXT('STUDENT STATUS')
A		VALUES('A' 'I')
		DFT('A')
A	K	SS#

Explanation:

- UNIQUE - NO DUPLICATE RECORDS BASED ON KEY SS#
- VALUES - RESTRICTS FIELD TO THESE VALUES
- DFT - DEFAULT VALUE WHEN RECORD IS CREATED

Descriptions of the database include:

- Naming tables and columns.
- Describing physical and logical domains.
- Defining indexes.
- Specifying constraints, such as business policies, security restrictions, and interrelational restraints, and integrity rules.

When the tables and columns have been created and defined, the database can be filled with the user's data. If the data is already computer-formatted (for example, in an existing database), it can often be imported automatically into the newly created database files. Otherwise, the data is manually keyed into the proper database fields.

3.13 Queries

So far we have broken down our order entry system into three tables and added foreign keys to the Orders and OrderDetails tables. Now, rather than repeating the Customers table data for each Orders table record, we simply record a customer number in the OrdCustID field. By doing this, we can change the information in the Customers table record and have that change be reflected in every order placed by the customer. This is accomplished by using queries to reassemble the data.

One of the inherent problems of any type of data management system is that ultimately the human users of the system will only be able to view data in two dimensions, which in the end become rows and columns in a table either on the screen or on paper. While people can conceptualize objects in three dimensions, its very difficult to represent detail data in anything other than a flat table. After all the effort we went through to break down the original flat file into three tables, we are now going to undo that effort and make a flat file again.

We are going to accomplish this amazing feat of backwards progress by using queries. A query is simply a view of data which represents the data from one or more tables. Lets say we want to see the orders placed by our customers. We can link the Customers and Orders tables using the CustID field from Customers and the OrdCustID field from Orders - remember, the value of the OrdCustID field represents a related record in the Customers table and is equal to the CustID value from that record. By joining together the two tables based on this relationship, we can add fields from both tables and see all orders along with any pertinent customer data.

Example Queries

1. For example, to find the branch-name, loan number, customer name and amount for loans over $1200:

$$\{t \mid t \in borrow \wedge t[amount] > 1200\}$$

This gives us all attributes, but suppose we only want the customer names. (We would use project in the algebra.)

We need to write an expression for a relation on scheme (cname).

$$\{t \mid \exists s \in borrow\,(t[cname] = s[cname] \wedge s[amount] > 1200)\}$$

In English, we may read this equation as "the set of all tuples s such that there exists a tuple t in the relation borrow for which the values of s and t for the cname attribute are equal, and the value of s for the amount attribute is greater than 1200."

The notation $\exists t \in r(Q(t))$ means "there exists a tuple in relation such that predicate $Q(t)$ is true".

How did we get the above expression? We needed tuples on scheme cname such that there were tuples in borrow pertaining to that customer name with amount attribute > 1200.

The tuples t get the scheme cname implicitly as that is the only attribute t is mentioned with.

Let us look at a more complex example.

Find all customers having a loan from the SFU branch, and the the cities in which they live:

$$\{t \mid \exists s \in borrow(t[cname] = s[cname] \wedge s[bname] = \text{“SFU”} \wedge \exists u \in customer(u[cname] = s[cname] \wedge t[ccity] = u[ccity]))\}$$

In English, we might read this as "the set of all (cname, ccity) tuples for which cname is a borrower at the SFU branch, and ccity is the city of cname".

Tuple variable s ensures that the customer is a borrower at the SFU branch.

Tuple variable u is restricted to pertain to the same customer as s, and also ensures that ccity is the city of the customer.

The logical connectives $\wedge$ (AND) and $\vee$ (OR) are allowed, as well as $\neg$(negation).

We also use the existential quantifier $\exists$ and $\forall$ the universal quantifier .

Some more examples:

1. Find all customers having a loan, an account, or both at the SFU branch:

$$\{t \mid \exists s \in borrow(t[cname] = s[cname] \wedge s[bname] = \text{“SFU”}) \vee \exists u \in deposit(t[cname] = u[cname] \wedge u[bname] = \text{“SFU”})\}$$

Note the use of the $\vee$ connective.

As usual, set operations remove all duplicates.

2. Find all customers who have both a loan and an account at the SFU branch.

Solution: simply change the $\vee$ connective in 1 to a $\wedge$.

3. Find customers who have an account, but not a loan at the SFU branch.

$$\{t \mid \exists u \in deposit(t[cname] = u[cname] \wedge u[bname] = \text{“SFU”}) \wedge \neg\exists s \in borrow(t[cname] = s[cname] \wedge s[bname] = \text{“SFU”})\}$$

4. Find all customers who have an account at all branches located in Brooklyn (We have used division in relational algebra).

For this example we will use implication, denoted by a pointing finger in the text, but by $\forall$ here.

The formula $P \Rightarrow Q$ means P implies Q, or, if P is true, then Q must be true.

$$\{t \mid \exists s \in borrow(t[cname] = s[cname] \wedge s[bname] = \text{“SFU”} \wedge \exists u \in customer(u[cname] = s[cname] \wedge t[ccity] = u[ccity]))\}$$

In English: the set of all cname tuples t such that for all tuples u in the branch relation, if the value of u on attribute bcity is Brooklyn, then the customer has an account at the branch whose name appears in the bname attribute of u.

Division is difficult to understand. Think it through carefully.

3.14 Structured Query Language

Queries are built in a relational database using Structured Query Language, or SQL. (Just in case you are wondering, some spell it out, and others say the word sequel, here we have used the latter SQL is the standard language for relational databases and includes the capability of manipulating both the structure of a database and its data. In its most common form, SQL

is used to create a simple SELECT query. Without getting into all the details now, suffice it to say that if you will be doing any serious work with databases, you're going to need to learn SQL.

Let us take the earlier example and build a query to look at customer orders. Here is the SQL for it:

That wasn't too tough. Lets look in a little more detail. This query starts with the SELECT keyword. Most of the queries you will be building will be SELECT queries. SELECT simply means that we wish to "select" records, or retrieve records from the tables. Following the SELECT keyword is the list of fields. Next comes the FROM keyword. This is used to indicate where the data is coming from. In this case, its coming from the Customers table and the Orders table. The key to this query is the INNER JOIN. There are two basic types of joins which can be done between tables: inner joins and outer joins. An inner join will return records for which only the matching fields in both tables are equal. An outer join will return all the records from one table, and only the matching records from the other table. Outer joins are further divided into left joins and right joins. The left or right specifies which side of the join returns all records. The balance of the example query specifies which fields are used to join the table. In this case we are matching the CustID field from Customers to the OrdCustID field (the foreign key) in Orders.

One thing that should be noted is that this is Jet SQL. Each RDBMS has its own particular dialect of SQL, just as Visual Basic is derived from some original BASIC language somewhere, Jet SQL is a variation of SQL particular to Microsoft's Jet database engine. For a complete description of the features of Jet SQL, search the VB help files for the topics "Jet SQL" and "reserved word, Jet".

3.15 Maintaining Integrity

Maintaining database integrity involves ensuring data in the database is accurate. This applies to cases where redundancy appears and also where it does not. Centralised control of the database can help in avoiding this problem, insofar as it can be avoided - by permitting the DBA to do value range checks whenever any update operation is attempted.

Data integrity is even more important in a multi-user database system because the data is shared, for without appropriate controls it would be possible for one user to update the database incorrectly which could cause problems for other users. It is worth noting that most commercial products used to be somewhat weak in their support for integrity controls. More recently however this is changing with more recent versions of DBMS products, ORACLE V7 for example, giving more and more support. This is a consequence of the standardisation of SQL where SQL2 does support certain types of integrity.

3.16 Defining Data Integrity

Data integrity means that data remains stable, secure, and accurate. It is maintained by internal constraints known as integrity rules that are invisible to users.

In this section, we will discuss types of data integrity, the methods available to protect data, and look at a few scenarios where you might apply data integrity rules.

As a database developer, protecting the data in the database is one of your most important responsibilities, perhaps the most important. The current generation of database engines, including Jet, provide a powerful array of tools to assist you in making sure the data in your database is accurate and consistent. Although no amount of programming can prevent every type of error that could be introduced, you should use the tools available to you to do whatever you can to guarantee the validity of the data.

The types of data integrity can be broken down into four basic areas:

- Entity Integrity - No duplicate rows.
  - Domain Integrity - The values in any given column fall within an accepted range.
- Referential Integrity - Foreign key values point to valid rows in the referenced table.
- User-defined Integrity - The data complies with applicable business rules.

3.16.1 Integrity Rules

The relational model uses integrity rules to create database tables that are accurate and consistent. Although data integrity is often maintained by building constraints into the application programs, many developers apply these constraints at the modeling level, requiring less work from the program itself. Integrity rules can be applied to models in two ways:

- By building tables that store information in only one place.

This method ensures that changes in data are automatically updated across the system and will not be redundant.

- By using normalisation techniques to enforce constraints with domains and keys.

The four types of integrity rules used in data modeling are:

As described above, some of these rules are built or defined in the actual modeling structure, while others can be specified in a DBMS application such as Access. However, not all DBMS applications handle integrity rules the same way. Following are the definitions of the various types of integrity rules and how they might be enforced in a DBMS application.

3.16.2 Relational Integrity Rules

There are two integrity rules associated with the relational model:

- Entity Integrity
- Referential Integrity

These two rules are general rules which apply to every database that claims to conform to the relational model prescription and have to do respectively with primary keys and foreign keys. The rules refer to database states. There is nothing solid said about how to avoid incorrect database states and many commercial DBMS support stopping the execution of DB updates which would lead to violations of these states.

3.16.3 Referential Integrity

The referential integrity rule for the relational model states that if base relation R2 includes a foreign key FK matching the primary key PK of some base relation R1 then every value of FK in R2 must

(a) Be equal to the value of PK in some tuple of R1; or

(b) Be wholly null i.e. each attribute in that FK must be null.

Referential integrity states that a given nonnull foreign key value must have a matching primary key value somewhere in the referenced relation. For example, when adding a row to a relation containing a foreign key, the relation containing the referenced primary key must have a matching value. In addition, when a relation containing a primary key references a relation with a foreign key, primary key values cannot be changed or deleted in the primary key relation. This would cause the foreign key value to be orphaned, destroying referential integrity. Microsoft Access addresses referential integrity by restrictions or cascades. By specifying "Enforce Referential Integrity" in Access, deletions or changes to primary key records containing foreign key references are disallowed, while designating "Cascades" automatically updates deletions or changes.

Referential integrity can be automatically set in Access in the following screen:

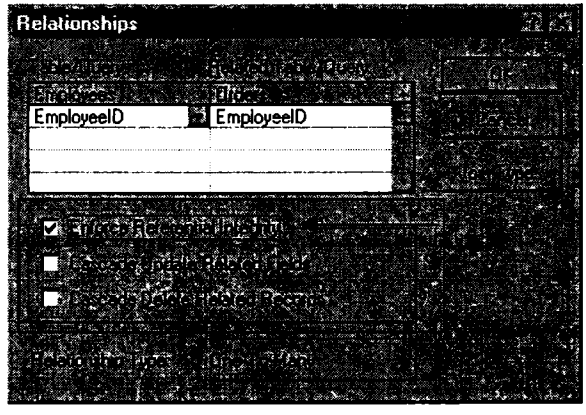


Fig. 3.23

Let us consider what happens when you start manipulating the records involved in the order entry system. You can edit the customer information at will without any ill effects, but what would happen if you needed to delete a customer? If the customer has orders, the orders will be orphaned. Clearly you can't have an order placed by a non-existent customer, so you must have a means in place to enforce that for each order, there is a corresponding customer. This is the basis of enforcing referential integrity. There are two ways that you can enforce the validity of the data in this situation. One is by cascading deletions through the related tables; the other is by preventing deletions when related records exist.

Database applications have several choices available for enforcing referential integrity, but if possible, you should let the database engine do its job and handle this for you. The latest advanced database engines allow you to use declarative referential integrity. You specify a relationship between tables at design time, indicating if updates and deletes will cascade through related tables. If cascading updates are enabled, changes to the primary key in a table are propagated through related tables. If cascading deletes are enabled, deletions from a table are propagated through related tables.

Looking again at our order entry system, if cascading updates are enabled, a change to the CustID for a Customers table record would change all of the related OrdCustID values in the Orders table. If cascading deletes are enabled, deleting a record from Customers would delete any related records in the Orders table. In contrast, if cascading updates or deletes are not enabled, you would be prevented from changing the primary key or deleting a record from Customers if any related records exist in the Orders table.

Moreover, keep in mind that if you have enforced referential integrity in the relationship between Orders and OrderDetails, this relationship can also have an effect on your ability to manage records in Customers. Just as you can't delete a customer with orders, neither can you delete an order with detail items. The result is passed along as far as necessary. If you cascade deletes from Customers to Orders, but not from Orders to OrderDetails, you will be prevented from deleting a record in Customers if there are any Orders records which have related OrderDetails records.

Before you go ahead and enable cascading deletes on all your relationships, keep in mind that this can be a dangerous practice in some situations. Let us say you have a table called States, which lists the U.S.P.S two letter state abbreviation for each of the states in the country, along with the full name of the state. You use this table as a lookup table and to enforce the validity of the state entered in the Customers table. If you define a relationship between the States table and the Customers table with cascading deletes enabled, then delete a record from States, you will delete all Customers table records where the customer is located in that state. In most cases, I have chosen to let my applications handle cascading deletes. This gives me a bit of a buffer against errors in the application and helps to prevent the loss of data. The database engine will prevent deletions if related records exist, forcing me to account for those records explicitly.

Foreign key values point to valid rows in the referenced table. A foreign key is a value in one table that references, or points to, a related row in another table. It's absolutely imperative that referential integrity constraints be enforced. While it is possible (likely, in fact) that foreign key values may be null, they should never be invalid. If a foreign key is entered, it must reference a valid row in the related table. If you allow unenforced references to exist, you are inviting chaos in your data. Referential or relational integrity is really a special form of domain integrity. The domain of a foreign key value is all of the valid primary key values in the related table.

Depending on the database engine you are using, you may have several choices available for how and in what manner referential integrity constraints will be enforced. Many database engines (including Jet) allow you to use Declarative Referential Integrity rather than triggers

or application code to enforce constraints. With this approach, you define a relationship between the columns in two tables and the database engine enforces the constraint for inserts, updates, and deletes.

Assuming a one-to-many relationship, here are the constraints which will be imposed:

- When a row is inserted on the many side, a foreign key that is entered (pointing to a row on the one side) must reference a valid row.
- If the foreign key is updated on the many side, the new value must point to a valid row on the many side.
- A row on the one side cannot be deleted if related rows exist on the many side.
- The primary key from the one side cannot be updated if related rows exist on the many side.

Some database engines also allow you to specify how to handle changes to the one side of the relationship:

Cascading Updates

If cascading updates are specified, a change to the primary key on the one side of a relationship will be propogated through related rows on the many side so that all foreign keys which point to the row on the one side are updated.

Cascading Deletes

If cascading deletes are specified, deleting a row on the one side will delete any related rows on the many side.

Microsoft's Jet engine allows you to specifiy either cascading updates, deletes, or both.

Use caution when specifying cascading deletes. This can be a dangerous practice if you are using a lookup table to enforce a domain. Consider what would happen if you had a table of several thousand names and address related to a table of the 50 U.S. states (using the table of states to enforce that the state entered in the address is valid), then deleted California from the table of states. If cascading deletes were requested, you would delete not only the California row from the states table, but also delete all rows from the name and address table in the state of California.

Also note that if you are using some type of autonumbering column as a primary key, there is generally no point in specifying cascading updates, since this type of column typically can't be updated anyway.

This applies to both Jet Counter fields and MS SQL Server IDENTITY columns.

3.16.4 Entity Integrity

Whether the primary key is simple or composite, entity integrity states that no value in the primary key field can be null. Null values are defined as "empty or containing no value," as opposed to "0." Since primary key column values must be unduplicated in order to determine a unique record, a null primary key value makes no identification and returns no corresponding value. DBMS products such as Access automatically enforce entity integrity by prohibiting null values in primary key fields.

Entity integrity can be automatically set in Access in the following screen:

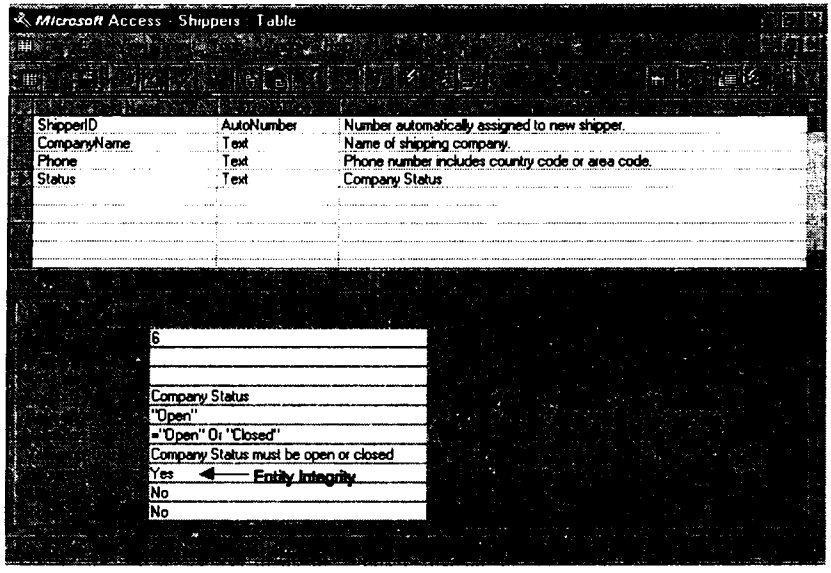


Fig. 3.23:

3.16.5 Domain Integrity

Domain integrity ensures that the values in columns of a relation are legal according to the physical and logical domain definitions. These domain descriptions are defined in the DBMS application. For instance, the StudentID attribute domains might be:

Physical: data type "numeric"; length "4 characters"

Logical: "the range of numbers between 1000 and 4999"

Therefore, the field would only accept input of a four-digit number between 1000 and 4999.

3.16.6 Entity Integrity

This simply means that in any given table, every row is unique. In a properly normalized, relational database, it is of particular importance to avoid duplicate rows in a table because users will expect that when a row is updated, there are no other rows that contain the same data. If there are duplicate rows, a user may update one of several duplicates and expect that the data has been updated for all instances, when in fact there are duplicates elsewhere that have not been updated. This will lead to inconsistencies in the data.

Entity integrity is normally enforced through the use of a primary key or unique index. However, it may at times be possible to have a unique primary key for a table and still have duplicate data. For example, if you have a table of name and address information, you may enter a row for the name "Robert Smith" while another user enter a row for the same individual but enters the name as "Bob Smith". No form of primary key or unique index would be able to trap this

type of violation. The only solution here is to provide the user a means of performing a search for existing data before new rows are created. (Don't be fooled into thinking that you can get away with putting a unique index on a combination of first, middle, and last names to avoid duplicates or your design will collapse when you need to enter rows for two people named John David Smith.) Another method of finding this type of situation before the data is entered is by using a "soft" search algorithm, such as a Soundex search.

Soundex is an algorithm that will produce a code which describes a phonetic equivalent of a word. Soundex codes are widely used with name searches because they will detect matches even if names are misspelled. Microsoft SQL Server comes with a built-in function to determine the Soundex code for a word. You can also build your own if the database engine you are using does not provide one. The algorithum is publicly available almost everywhere. A search of the Web for the keyword "Soundex" should produce the algorith within the first few matches.

In case of Domain Integrity:

- The values in any given column fall within an accepted range.
- It is important to make sure that the data entered into a table is not only correct, but appropriate for the columns it is entered into. The validity of a domain may be as broad as specifying only a data type (text, numeric, etc.) or as narrow as specifying just a few available values. Different database engines will have varying means of enforcing the validity of the domain of a column, but most will permit the entry of at least a simple Boolean expression that must be True for the value in a column to be accepted (Jet calls this a "Validation Rule"). If the range of acceptable values is large or changes frequently, you can also use a lookup table of valid entries and define a referential integrity constraint between the column and the lookup table. Although this approach is more complex to implement, it allows you to modify the domain without changing the table design. (Changing the table design often requires that you have exclusive access to the table - something which can be nearly impossible with a production database in use by even a modest number of users.)

In addition to single column rules, the domain of an entry may be dependent on two or more columns in a table. For example, the range of valid entries for a sub-category column may be dependent on the value entered in an associated category column. Unfortunately, it can be difficult or impossible to express this type of rule as a simple Boolean expression. If you are fortunate enough to be working with a database engine which provides the capability of writing insert, update and delete triggers you can code complex logic to handle this type of validation.

Unfortunately, Jet does not provide the capability of writing trigger code for data events. If you need to force some code to run when a row is inserted, updated, or deleted, you will need to go through a few hoops to do it. One method that we have applied is to revoke all permissions except Read on the table, then write functions to insert, update, or delete rows which use either an owner access query or a priviledged workspace to change the data.

Let's look at some examples of columns which could have domain integrity rules applied:

Column	Data Type	Domain
Social Security Number	Text	In the format xxx-xx-xxxx, where x is a number from 0 to 9. I tend to use text rather than numeric columns for formatted data such as this even if the value contains only numbers. This not only avoids the possibility of performing mathematical operations on the data, but saves the trouble of formatting it each time you need to present it if you save the data with the formatting in the column.
GradeLevel	Text	Exists in the following list: Freshman, Sophmore, Junior, Senior. This type of list is probably stable enough that a "hardcoded" list of values is sufficient.
Denomination	Text	Exists in the following list: Catholic, Episcopalian, Lutheran, Methodist, etc. This type of list is probably long enough and dynamic enough that it would be better handled using a lookup table of valid entries. To do this, you would need to create a table that holds a unique list of valid denominations and define a relationship between the lookup table and the Denomination column.
Insurer	Text	If the Insured column is True, then Insurer is not null. This type of multi-column validation typically can be defined at the table level with most database engines. With Jet, for example, the expression might be: If Insured Then Not Is Null (Insurer) Where Insured is a Boolean column which cannot be null.
Salary	Numeric	Falls within the range defined by the job title.

Unless you have trigger capability, this can be difficult to implement. A workaround is to use a lookup table of valid salaries by job title and a multiple-field relationship between the table containing the salary and the lookup table.

One thing to remember when working with domain integrity rules is that you not only need to understand the design of the database, you also need to understand the type of expected data and the business rules which apply to it. In many cases, you will have several choices available to you for the manner in which you implement a rule.

Data Type

Simple data type rules, such as "Value is numeric" should always be defined by specifying the appropriate data type for the column when you create it. Remember when working with numeric values to specify the appropriate type of number. In most cases, you will have a choice of whole number ranges (byte, integer, long integer), floating point (single and double precision), and financial. Be sure to use a financial data type (money, currency, etc.) when working with financial data. Most database engines use special data formats to provide greater consistency and accuracy than the equivalent floating point data type.

Formatted Data

For columns such as social security numbers, phone numbers, and other values which can be guaranteed to be in a fixed format, it is better to use a formatting rule and store the data as text. In general, you should prefer to use text data types for formatted data even if the values are all numerals. While you can use a numeric data type and perhaps save a few bytes of storage, you will need to deal with formatting issues each time you present the data. Additionally, by storing the data as text, you can allow the database engine (rather than your application) to enforce that the data is entered in the correct format.

Note: If you are building an application for international use or which contains international data, be sure to provide flexibility in the rules to account for the locality. Values such as phone numbers, postal codes, etc., vary from country to country.

Nullability

This can nearly always be defined as a Boolean expression. An exception would be if the nullability of a column is defined by the value in another column. In this case, the possible range of values for the second column would determine if you can use a simple expression or would need to define a lookup table.

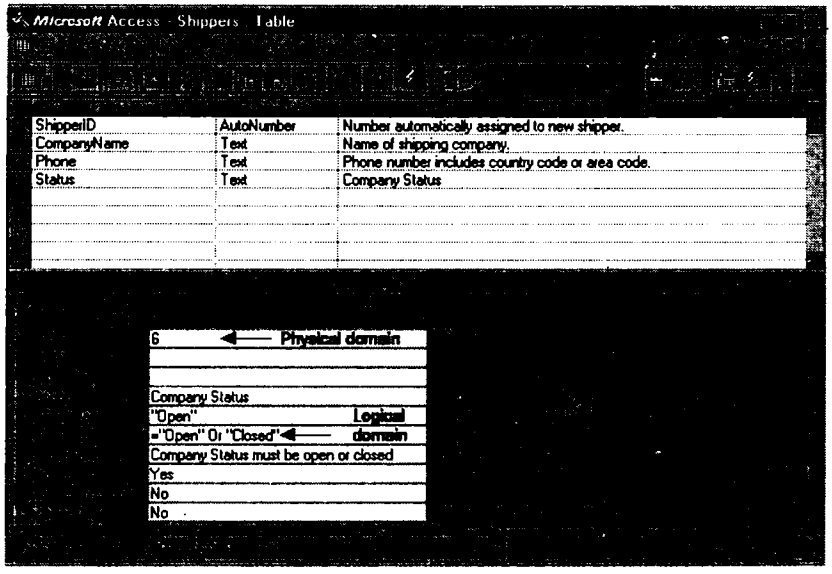
Value List

Unless we have a very high degree of certainty that the list will not change, We will use a lookup table. Examples of lists which could be "hardcoded" would be items such as military rank, states or provinces within the U.S. and Canada, school grade level, etc. If there is a possibility that the list is or could easily become dynamic or excessively long, you could prefer to use a lookup table and a referential integrity constraint. Keep in mind that data which appears stable when you begin to build a design could easily become dynamic if the business rules change.

Remember that whenever possible, you should let the database engine do the job of enforcing the integrity of a domain.

How domains are described in Ms-Access

Physical and logical attribute domains can be described in Microsoft Access DBMS under "Field Properties" and "Data Type."



3.16.7 User-defined Integrity

The data complies with applicable business rules. User-defined integrity is a kind of "catch-all" for rules which don't fit neatly into one of the other categories. Although all of the other types of data integrity can be used to enforce business rules, all business rules may not be able to be enforced using entity, domain, and referential integrity. The types of rules that apply will of course vary by application. If possible, you should take advantage of the database engine to apply whatever constraints it is capable of, but some may need to be enforced using the application code.

A textbook example of a business rule which must be enforced is a funds transfer in a banking application. If a customer wishes to transfer funds from a savings account to a checking account, you must deduct the withdrawl from the savings account and add the deposit to the checking account. If you successfully record the withdrawl without recording the deposit, the customer will have "lost" the amount (and not be pleased). If you record the deposit without the withdrawl, the customer will get a "free" deposit (and the bank will be unhappy). To enforce this type of rule, you will need to use a Transaction in your application code.

Users can define integrity rules according to business policies. For example, the fact that "a student must register for 1996 Fall Quarter on or before September 16, 1996" is a business policy of Mercer. Some DBMS products allow the definition of user-defined rules.

Some types of user-defined integrity can be set in Access in the following screen:

Referential Integrity

Often we wish to ensure that a value appearing in a relation for a given set of attributes also appears for another set of attributes in another relation. This is called referential integrity.

3.17 Integrity Constraints

1. Integrity constraints provide a way of ensuring that changes made to the database by authorized users do not result in a loss of data consistency.
2. We saw a form of integrity constraint with E-R models:
   - key declarations: stipulation that certain attributes form a candidate key for the entity set.
   - form of a relationship: mapping cardinalities 1-1, 1-many and many-many.
3. An integrity constraint can be any arbitrary predicate applied to the database.
4. They may be costly to evaluate, so we will only consider integrity constraints that can be tested with minimal overhead.

3.17.1 Domain Constraints

1. A domain of possible values should be associated with every attribute. These domain constraints are the most basic form of integrity constraint. They are easy to test for when data is entered.
2. Domain types
   - Attributes may have the same domain, e.g. cname and employee- name.
   - It is not as clear whether bname and cname domains ought to be distinct.
   - At the implementation level, they are both character strings.
   - At the conceptual level, we do not expect customers to have the same names as branches, in general.
   - Strong typing of domains allows us to test for values inserted, and whether queries make sense. Newer systems, particularly object-oriented database systems, offer a rich set of domain types that can be extended easily.
3. The check clause in SQL-92 permits domains to be restricted in powerful ways that most programming language type systems do not permit.
   - The check clause permits schema designer to specify a predicate that must be satisfied by any value assigned to a variable whose type is the domain.

Examples:

```
create domain hourly-wage numeric(5,2)
constraint wage-value-test check(value >= 4.00)
```

Note that "constraint wage-value-test" is optional (to give a name to the test to signal which constraint is violated).

```
create domain account-number char(10)
constraint account-number-null-test
check(value not null)
create domain account-type char(10)
constraint account-type-test
check(value in ("Checking", "Saving"))
```

3.18 Normalisation

Normalisation is a subject that is made overly confusing by most literature on the subject that we have read. In a nutshell, its simply the process of distilling the structure of the database to the point where you have removed repeating groups of data into separate tables. In our example, we have normalized customers and orders by creating a separate table for the orders. If you look around, you can probably find a list of books on normalisation theory as long as your arm. Read on if you wish, but the bottom line is that you need to design your database to be efficient and reliable. At times you may need to sacrifice normalisation to practicality.

Normalisation is essentially the process of taking a wide table with lots of columns but few rows and redesigning it as several narrow tables with fewer columns but more rows. A properly normalized design allows you to use storage space efficiently, eliminate redundant data, reduce or eliminate inconsistent data, and ease the data maintenence burden. Before looking at the forms of normalisation, you need to know one cardinal rule for normalizing a database:

- You must be able to reconstruct the original flat view of the data.
- If you violate this rule, you will have defeated the purpose of normalizing the design.

If you look at the Customers table, you can see that it isn't really necessary to include the CustCity and CustState fields since a US ZIP Code uniquely defines a city and state in the US. If you were to fully normalize the Customers table, you would need to remove the CustCity and CustState fields and create a table, perhaps called ZIPCodes, which included these fields, then include only the CustZIP field and join the Customers table to the ZIPCodes table in order to reconstruct the full address. The problem with this is that you add the overhead of an additional join in every query where you need to have the full address available.

There aren't any hard and fast rules for when to stop normalizing a database. You need to make your own choices based on the practicality of the data structures and the performance trade-offs involved. If possible, you should at least try to design the application so that you can restructure the data to accomodate normalizing or denormalizing the tables.

Building a database structure is a process of examining the data which is useful and necessary for an application, then breaking it down into a relatively simple row and column format. There are two points to understand about tables and columns that are the essence of any database:

- Tables store data about an entity

An entity may be a person, a part in a machine, a book, or any other tangible or intangible object, but the primary consideration is that a table only contain data about one thing.

- Columns contain the attributes of the entity

Just as a table will contain data about a single entity, each column should only contain one item of data about that entity. If, for example, you are creating a table of addresses, there's no point in having a single column contain the city, state, and postal code when it is just as easy to create three columns and record each attribute separately.

One method that is helpful when initially breaking down the data for tables and columns is to examine the names that are used. I use a plural form of a noun for table names (Authors, Books, etc.), and a noun or noun and adjective for column names (FirstName, City, etc.). If you find that you are coming up with names that require the use of the word "and" or the use of two nouns, it's an indication that you haven't gone far enough in breaking down the data. An exception to this would be a table that serves as middle of a many to many relationship, such as BookAuthors, etc.

Before continuing on to discuss normalisation, we will look at some of the common flaws in database designs and the problems they cause. To illustrate these problems, I'll use the following sample table, which I'll call simply "Ugly":

Table 3.1: Ugly

Student Name	Advisor Name	Course ID1 Course Description 1 Course Instructor Name 1 Course ID2 Course Description 2 Course Instructor Name 2
Sanjay Sharma	Ashu Jain	VB1 Intro to Visual Basic Parveen Gupta DAO1 Intro to DAO Programming Jatin Verma
Manish Suri	Gaurav Kumar	DAO1 Intro to DAO Programming Ravi Kalra VBSQL1 Client/Server Programming with VBSQL Anurag Jain
Karan Joshi	Hari Pal	API1 API Programming with VB Ajay Verma OOP1 Object Oriented Programming in VB Akash Tyagi
Mukesh Goel	Vinay Khanna	VB1 Intro to Visual Basic Aman Kumar API1 API Programming with VB Dev Kohli

Let us look at some of the problems with this structure:

Repeating Groups

The course ID, description, and instructor are repeated for each class. If a student needs a third class, you need to go back and modify the table design in order to record it. While you could add CourseID3, CourseID4, CourseID5, etc., along with the associated description and instructor fields, no matter how far you take it there may one day be someone who wants one more class. Additionally, adding all those fields when most students would never use them is a waste of storage.

Inconsistent Data

Let us say that after entering these rows, you discover that Bruce McKinney's course is actually title "Intro to Advanced Visual Basic". In order to reflect this change, you would need to examine all the rows and change each individually. This introduces the potential for errors if one of the changes is omitted or done incorrectly.

Delete Anomalies

If you no longer wished to track Joe Garrick's Intro to DAO class, you would need to delete two students, two advisors, and one additional instructor in order to do it. If you remove the first two rows of the table, all of the data is deleted with the reference to the course.

Insert Anomalies

Perhaps the department head wishes to add a new class - let's call it "Advanced DAO Programming" - but hasn't yet set up a schedule or even an instructor. What would you enter for the student, advisor, and instructor names?

As you can see, this single flat table has introduced a number of problems - all of which can be solved by normalizing the table design. Do not be misled into thinking that normalisation is the answer to all your problems in developing a database design. As you will see later, there may times when it's prudent to denormalize a structure. There's a variety of other problems that can be introduced into your data as well as an infinite variety of complex business rules which may need to be applied.

The normal forms defined in relational database theory represent guidelines for record design. The guidelines corresponding to first through fifth normal forms are presented here, in terms that do not require an understanding of relational theory. The design guidelines are meaningful even if one is not using a relational database system. We present the guidelines without referring to the concepts of the relational model in order to emphasize their generality, and also to make them easier to understand. Our presentation conveys an intuitive sense of the intended constraints on record design, although in its informality it may be imprecise in some technical details.

The normalisation rules are designed to prevent update anomalies and data inconsistencies. With respect to performance tradeoffs, these guidelines are biased toward the assumption that all non- key fields will be updated frequently. They tend to penalize retrieval, since data which may have been retrievable from one record in an unnormalized design may have to be retrieved from several records in the normalized form. There is no obligation to fully normalize all records when actual performance requirements are taken into account.

3.19 Benefits of Normalisation

Normalisation is a data analysis method used during the design stage of relational data modeling. The components of normalisation are referred to as normal forms, a progressive series of rules that can be applied to simplify and refine each relation. Normalisation is not a part of the relational model, but a separate concept that can be applied to it.

The goal of normalisation is to define database tables that can be updated and modified with predictable results. Often, relations contain problems called anomalies, which can result in data redundancy and inconsistency. In the 1970s, relational theorists defined and classified these anomalies and created normal forms to resolve them. Initially, Codd classified first, second, and third normal forms; others, such as Boyce/Codd, fourth, and fifth normal forms, were defined later. First, second, third, and fourth normal forms will be explored in this text. Boyce/Codd will be described briefly.

Designers should be careful not to apply normalisation techniques too soon in modeling, for instance, in the object modeling stage. Remember, the purpose of object modeling is to capture the entities of a business and their relationships. Subtyping and decomposing these entities properly will eliminate most anomalies early in the design. Normalisation should only be applied to tables at the relational modeling stage if further refinement is needed.

The ultimate benefit of normalisation is data integrity, which can be assured because normal forms:

- Simplify entities by creating relations with one theme;
- Build tables that can be easily joined with other tables to produce information;
- Reduce redundant data across relations (since normalisation attempts to store each data value in only one place.);
- Avoid lost data by requiring that each field contain atomic values and by designating appropriate primary and foreign keys;
- Reduce modification anomalies, such as deletion, insertion, and update anomalies; and
- Define relation constraints that are a logical consequence of domains and keys.

3.19.1 Example of Deletion Anomaly

This anomaly occurs when deleting a row in a relation causes the loss of other important information or creates orphans in another relation.

RELATION = STU-ACT

StuID	Activity	Fee
100	Diving	200
150	Softball	50
175	Racquetball	50
200	Softball	50

Deletion of Student 1001 also deletes the fact that diving costs $200.

3.19.2 Example of Insertion Anomaly

This anomaly occurs when a row cannot be inserted into a relation without having facts about another entity.

RELATION = STU-ACT

StuID	Activity	Fee
1001	Diving	200
1005	Softball	50
1006	Racquetball	45
1010	Softball	50

The fact that Swimming costs $25 cannot be added to the table until a student is enrolled.

3.19.3 Example of Update Anomaly

This anomaly occurs when information must be changed in more than one field or relation to update the database. If this data is not changed in each location, the relation will contain inconsistent data.

RELATION = STU-BLDG

StuID	Building	Rent
100	Russell	1500
150	Baldwin	1000
200	Russell	1500
250	Monarch	1200

If the rent for Russell Building is increased to $1700, the data must be changed in every row containing Russell.

3.19.4 Use of Functional Dependencies and Keys in Normal Forms

In a database, the value of one attribute or combination of attributes is used to look up a value of another attribute. Normalisation techniques use functional dependencies and keys to create the most efficient database structures possible in order to obtain these values.

As previously described, a functional dependency is an association between the columns of a relation. For example, consider the following example of this relation from the student registration model:

RELATION = ENROLLED STUDENT

STUDENT ID	STUDENT NAME	STUDENT Phone
1001	Sapna Jain	011-3318113
1002	Raman Bhardwaj	011-3388026
1003	Sandhya Rajput	011-5146048

Here, one value of the attribute Student ID can determine a unique value from the attributes Student Name and/or Student Phone. For example, Student ID 1001 determines that the Student Name is John Black and the Student Phone is 445-9385. Thus, Student Name and Student Phone are functionally dependent on Student ID. Another way of saying this is that Student ID functionally determines Student Name and Student Phone. Therefore, a given value of one attribute will always return the same value of another attribute.

In the above example, Student ID is a determinant and the primary key. This is not always true. A relation can contain a determinant that does not have unique values because functional dependency actually addresses how attributes are related. Remember, however, that a primary key value identifies the values of an entire row in a relation. Therefore, a primary key attribute must have unique values.

3.19.5 Functional Dependency and Determinants

The example of functional dependency and determinants shown in the text was extremely simple. The following might provide more insight.

This relation represents a partial schedule of college classes offered for the Fall 1996 Quarter. However, certain facts must be known about a relation in order to properly identify its functional dependencies. The context of this relation is that for Fall 1996 Quarter:

- A class can be offered more than once at a school.
- A class can be offered at more than one school.
- A specific class will not be offered at the same school on the same day.

ClassID	School	ClsStart	ClsEnd	Capacity
INSY311	Douglas	9/15/96	12/15/96	20
ENGL102	Atlanta	9/13/96	12/13/96	25
INSY495	Griffin	9/15/96	12/15/96	15
ENGL102	Douglas	9/15/96	12/15/96	20
ENGL102	Douglas	9/11/96	12/11/96	22

Primary key = ClassID-School

The functional dependencies in this relation are:

- ClsStart is functionally dependent on ClassID-School (or ClassID- School functionally determines ClsStart).
- Capacity is functionally dependent on ClassID-School (or ClassID-School functionally determines Capacity).
- ClsEnd is functionally dependent on ClsStart (or ClsStart functionally determines ClsEnd).

This can be written another way:

```
ClassID-School → ClsStart
ClassID-School → Capacity
ClsStart → ClsEnd
```

The arrow points from the determinant to the functional dependency.

Functional dependency simplifies the structure of relations and increases understanding of user's data. The concepts of functional dependencies, determinants, and primary keys might be confusing until seen in their application to normal forms.

3.19.6 Why Normalise

Database normalisation can essentially be defined as the practice of optimizing table structures. Optimization is accomplished as a result of a thorough investigation of the various pieces of data that will be stored within the database, in particular concentrating upon how this data is interrelated. An analysis of this data and its corresponding relationships is advantageous because it can result both in a substantial improvement in the speed in which the tables are queried, and in decreasing the chance that the database integrity could be compromised due to tedious maintenance procedures.

Before delving further into the subject of db normalisation, allow me to introduce a few terms that frequently arise when discussing this subject. To better illustrate the meaning of the respective terms, we will allude to a hypothetical database which contains information about a school scheduling system.

Thus far the only thing that we have really learnt about database normalisation is that it provides for table optimization through the investigation of entity relationships. But why is this necessary? In this section, we will elaborate a bit upon why normalisation is necessary when creating commercial database applications.

Essentially, table optimization is accomplished through the elimination of all instances of data redundancy and unforeseen scaleability issues.

3.20 Redundancy

Data redundancy is exactly what you think it is; the repetition of data. One obvious drawback of data repetition is that it consumes more space and resources than is necessary. Consider the following table:

Table 3: Poorly Defined Table

student_id	class_name	time	location	professor_id
999-40-9876	Math 148 MWF	11:30	Rm. 432	prof1 45
999-43-0987	Physics 113 TR	1:30	Rm. 12	prof1 43
999-42-9842	Botany 42 F	12:45	Rm. 9	prof1 67
999-41-9832	Matj 148 MWF	11:30	Rm. 432	prof1 45

Basically this table is a mapping of various students to the classes found within their schedule. Seems logical enough, right?

Actually, there are some serious issues with the choice to store data in this format. First of all, assuming that the only intention of this table is to create student-class mappings, then there really is no need to repeatedly store the class time and professor ID. Just think that if there are 30 students to a class, then the class information would be repeated 30 times over!

Moreover, redundancy introduces the possibility for error. You might have noticed the name of the class found in the final row in the table (Matj 148). Given the name of the class found in the first row, chances are that Matj 148 should actually be Math 148! While this error is easily identifiable when just four rows are present in the table, imagine finding this error within the rows representing the 60,000 enrolled students at my alma mater, The Ohio State University. Chances that you will find these errors are unlikely, at best. And the cost of even attempting to find them will always be high.

3.21 Unforeseen Scaleability Issues

Unforeseen scaleability issues generally arise due to lack of forethought pertaining to just how large a database might grow. Of course, as a database grows in size, initial design decisions will continue to play a greater role in the speed of and resources allocated to this database. For example, it is typically a very bad idea to limit the potential for expansion of the information that is to be held within the database, even if there are currently no plans to expand. For example, structurally limiting the database to allot space for only three classes per student could prove deadly if next year the school board decides to permit all students to schedule three classes. This also works in the opposite direction; What if the school board subsequently decides to only allow students to schedule two classes? Have you allowed for adequate flexibility in the design so as to easily adapt to these new policies?

The remedy to these problems is through the use of a process known as database normalisation. A subject of continued research and debate over the years, several general rules have been formulated that layout the process one should follow in the quest to normalize a database.

Many of your databases will be small, with one or two tables. But as you become braver, tackling bigger projects, you may start finding that the design of your tables is proving problematic. The SQL you write starts to become unwieldy, and data anomalies start to creep in. It is time to learn about database normalisation, or the optimization of tables.

Let us begin by creating a sample set of data. Imagine we are working on a system to keep track of employees working on certain projects.

Project number	Project name	Employee number	Employee name	Rate category	Hourly rate
1023	Madagascar travel site	11	Rajeev Kohli	A	$60
12			Manoj Malhotra	B	$50
16			Anil Chawla	C	$40
1056	Online estate agency	11	Rajeev Kohli	A	$60
17			Prem Bhardwaj	B	$50

A problem with the above data should immediately be obvious. Tables in relational databases, which would include most databases you will work with, are in a simple grid, or table format. Here, each project has a set of employees. So we couldn't even enter the data into this kind of table. And if we tried to use null fields to cater for the fields that have no value, then we cannot use the project number, or any other field, as a primary key (a primary key is a field, or list of fields, that uniquely identify one record). There is not much use in having a table if we can't uniquely identify each record in it. So, our solution is to make sure that each field has no sets, or repeating groups. Now we can place the data in a table.

employee_project table

Project number	Project name	Employee number	Employee name	Rate category	Hourly rate
1023	Madagascar travel site	11	Rajeev Kohli	A	$60
1023	Madagascar travel site	12	Manoj Malhotra	B	$50
1023	Madagascar travel site	16	Anil Chawla	C	$40
1056	Online estate agency	11	Rajiv Kohli	A	$60
1056	Online estate agency	17	Prem Bhardwaj	B	$50

Notice that the project number cannot be a primary key on it's own. It does not uniquely identify a row of data. So, our primary key must be a combination of project number and employee number. Together these two fields uniquely identify one row of data. (Think about it. You would never add the same employee more than once to a project. If for some reason this could occur, you would need to add something else to the key to make it unique).

Database Normalisation

Database Normalisation - Part 2

So, now our data can go in table format, but there are still some problems with it. We store the information that code 1023 refers to the Madagascar travel site 3 times! Besides the waste of space, there is another serious problem. Look carefully at the data below.

employee_project table

Project number	Project name	Employee number	Employee name	Rate category	Hourly rate
1023	Madagascar travel site	11	Rajeev Kohli	A	$60
1023	Madagascar travel site	12	Manoj Malhotra	B	$50
1023	Madagascat travel site	16	Anil Chawla	C	$40
1056	Online estate agency	11	Rajeev Kohli	A	$60
1056	Online estate agency	17	Prem Bhardwaj	B	$50

Did you notice anything strange in the data above? Madagascar is misspelt in the 3rd record. Now imagine trying to spot this error in a table with thousands of records! By using the structure above, the chances of the data being corrupted increases drastically.

The solution is simply to take out the duplication. What we are doing formally is looking for partial dependencies, ie fields that are dependent on a part of a key, and not the entire key. Since both project number and employee number make up the key, we look for fields that are dependent only on project number, or on employee number.

We identify two fields. Project name is dependent on project number only (employee_number is irrelevant in determining project name), and the same applies to employee name, hourly rate and rate category, which are dependent on employee number. So, we take out these fields, as follows:

employee_project table

Project number	Employee number
1023	11
1023	12
1023	16
1056	11
1056	17

Clearly we cannot simply take out the data and leave it out of our database. We take it out, and put it into a new table, consisting of the field that has the partial dependency, and the field it is dependent on. So, we identified the employee name, hourly rate and rate category as being dependent on employee number. The new table will consist of employee number as a key, and employee name, rate category and hourly rate, as follows:

Employee table

Employee number	Employee name	Rate category	Hourly rate
11	Rajeev Kohli	A	$60
12	Manoj Malhotra	B	$50
16	Anil Chawla	C	$40
17	Prem Bhardwaj	B	$50

And the same for the project data.

Project table

Project number	Project name
1023	Madagascar travel site
1056	Online estate agency

Note the reduction of duplication. The text "Madagascar travel site" is stored once only, not for each occurrence of an employee working on that project. The link is made through the key, the project number. Obviously there is no way to remove the duplication of this number without losing the relation altogether, but it is far more efficient storing a short number repeatedly, than a large piece of text.

Database Normalisation - Part 3

We are still not perfect. There is still room for anomalies in the data. Look carefully at the data below:

Employee table

Employee number	Employee name	Rate category	Hourly rate
11	Rajeev Kohli	A	$60
12	Manoj Malhotra	B	$50
16	Anil Chawla	C	$40
17	Prem Bhardwaj	B	$40

The problem above is that Monique Williams has been awarded an hourly rate of $40, when she is actually category B, and should be earning $50 (In the case of this company, the rate category - hourly rate relationship is fixed. This may not always be the case). Once again we are storing data redundantly: the hourly rate - rate category relationship is being stored in its entirety for each employee. The solution, as before, is to remove this excess data into its own table. Formally, what we are doing is looking for transitive relationships, or relationships where a non-key attribute is dependent on another non-key relationship. Hourly rate, while being in one sense dependent on Employee number (we probably identified this dependency earlier, when looking for partial dependencies) is actually dependent on Rate category. So, we remove it, and place it in a new table, with its actual key, as follows:

Employee table

Employee number	Employee name	Rate category
11	Rajeev Kohli	A
12	Manoj Malhotra	B
16	Anil Chawla	C
17	Prem Bhardwaj	B

Rate table

Rate category	Hourly rate
A	$60
B	$50
C	$40

We have cut down once again. It is now impossible to mistakenly assume rate category "B" is associated with an hourly rate of anything but $50. These relationships are only stored in once place – our new table, where it can be ensured they are accurate.

Database Normalisation - Part 4

Let us run again through the example we have just done, this time without the data tables to guide us. After all, when you are designing a system, you usually won't have test data available at this stage. The tables were there to show you the consequences of storing data in unnormalized tables, but without them we can focus on dependency issues, which is the key to database normalisation.

In the beginning, the data structure we had was as follows:

- Project number
- Project name
- 1-n Employee numbers (1-n indicates that there are many occurrences of this field - it is a repeating group)
- 1-n Employee names
- 1-n Rate categories
- 1-n Hourly rates

So, to begin the normalisation process, we start by moving from zero normal form to 1st normal form.

The definition of 1st normal form

- There are no repeating groups
- All the key attributes are defined
- All attributes are dependent on the primary key

So far, we have no keys, and there are repeating groups. So we remove the repeating groups, and define the primary key, and are left with the following:

```
Employee project table
Project number - primary key
Project name
Employee number - primary key
Employee name
Rate category
Hourly rate
```

This table is in 1st normal form.

Database Normalisation - Part 5

A table is in 2nd normal form if

- it is in 1st normal form
- it includes no partial dependencies (where an attribute is dependent on only a part of a primary key).

So, we go through all the fields. Project name is only dependent on Project number. Employee name, Rate category and Hourly rate are dependent only on Employee number. So we remove them, and place these fields in a separate table, with the key being that part of the original key they are dependent on. So, we are left with the following 3 tables:

```
Employee project table
  Project number - primary key
  Employee number - primary key
Employee table
  Employee number - primary key
  Employee name
  Rate category
  Hourly rate
Project table
  Project number - primary key
  Project name
```

The table is now in 2nd normal form. Is it in 3rd normal form?

Database Normalisation - Part 6

The definition of 3rd normal form

- It's in 2nd normal form
- It contains no transitive dependencies (where a non-key attribute
- Is dependent on another non-key attribute).

We can narrow our search down to the Employee table, which is the only one with more than one non-key attribute. Employee name is not dependent on either Rate category or Hourly rate, the same applies to Rate category, but Hourly rate is dependent on Rate category. So, as before, we remove it, placing it in it's own table, with the attribute it was dependent on as key, as follows:

```
Employee project table
  Project number - primary key
  Employee number - primary key
Employee table
  Employee number - primary key
  Employee name
  Rate Category
Rate table
  Rate category - primary key
  Hourly rate
Project table
  Project number - primary key
  Project name
```

These tables are all now in 3rd normal form, and ready to be implemented. There are other normal forms - Boyce-Codd normal form, and 4th normal form, but these are very rarely used for business applications. In most cases, tables in 3rd normal form are already in these normal forms anyway.

Database Normalisation - Part 7

Before you rush off and start normalizing everything, a word of warning. No process is better than good old common sense. Take a look at this example.

```
Customer table
  Number - primary key
  Name
  Address
  Zip Code
  Town
```

What normal form is this table in? Giving it a quick glance, we see no repeating groups, and a primary key defined, so it is at least in 1st normal form. There's only one key, so we needn't even look for partial dependencies, so it is at least in 2nd normal form. How about transitive dependencies? Well, it looks like Town might be determined by Zip Code. And in most parts of the world that's usually the case. So we should remove Town, and place it in a separate table, with Zip Code as the key? No! Although this table is not technically in the 3rd normal form, removing this information is not worth it. Creating more tables increases the load slightly, slowing processing down. This is often counteracted by the reduction in table sizes, and redundant data. But in this case, where the town would almost always be referenced as part of the address, it isn't worth it. Perhaps a company that uses the data to produce regular mailing lists of thousands of customers should normalize fully. It always comes down to how the data is going to be used. Normalisation is just a helpful process that usually results in the most efficient table structure, and not a rule for database design.

3.22 Forms of Normalisation

Six normal forms have been formally defined. As a rule, each normal form builds on the one before. While it is important to normalize relations to the highest form possible, the designer should strive at least for third normal form when building the relational model.

The process of normalisation involves eliminating problems in a relation by decomposing it into two or more separate relations. This decomposition does not result in any loss of information. On the contrary, often additional information is provided in building new tables.

When using normalisation techniques, understanding the context (or meaning) and the environment of each relation and its attributes is essential. Otherwise, it might be difficult to determine whether a relation meets the requirements for a specific normal form. A different perspective of a particular attribute can change dependency properties as the normal form is applied. For example, whether a student can have more than one major can change how a relation is normalized. (This is illustrated in the description of the first normal form.)

Another important consideration is whether normalisation is always necessary. Normalisation techniques are guidelines to be used for checking the usefulness and accuracy of relational models. But the ultimate purpose of modeling is to create a well-structured, efficient database. Just as redundant data can slow the performance of the database, so can an overabundance of tables. When analyzing the relational model, use common sense. Relations should definitely be normalized if necessary, but consider the trade-offs.

Relational database theorists have divided normalisation into several rules called normal forms.

3.22.1 First Normal Form (1NF)

A relation is considered to be in 1NF only if it meets the requirements of a relation and each column contains atomic (single) values. An attribute that contains multiple values in one record can result in lost data. However, a table in 1NF often has problems that can result in modification anomalies. For most models, relations should be tested beyond 1NF.

Example of 1st normal form

Consider the relation STUDENT where a student can have only one major. StuID is the primary key. Since it has a simple primary key, this relation is also in 2NF.

To be in 1NF, the table of data must meet the requirements of a relation.

RELATION = STUDENT

StuID	StuName	Major
1001	Jones	Accounting
1005	Phillips	Science
1006	Stevens	Art
1010	Barber	Business

Fig. 3.29

Consider the same relation in a different context: A student can have more than one major. Since a student can have more than one major, the primary key must be StuID-Major. With a composite primary key, the relation must be checked for 2NF.

RELATION = STUDENT StuID StuName Major

StuID	StuName	Major
1001	Jones	Business
1001	Jones	Accounting
1005	Phillips	Science
1006	Stevens	Art
1010	Barber	English
1010	Barber	Business

Fig. 3.30

Primary Key = StuID-Major

1NF Definition

By default, all tables in a relational database are in 1NF as the underlying domain(s) contain atomic or simple, nidivisible values only. The values of any attribute in a tuple must be a single value, not a set, from the domain of that attribute. This disallows relations within relations or nested relations.

Another Example of First Normal Form: No repeating groups.

What we are looking for is repeating groups of columns. The purpose is to reduce the width of the table. This is done by taking the groups of columns and making a new table where the table is defined using the columns that repeat. Rather than having additional columns, the resulting table has more rows.

OK, so what is a repeating group? Let us look at the columns in our sample table Ugly:

- StudentName
- AdvisorName
- CourseID1
- CourseDescription1
- CourseInstructorName1
- CourseID2
- CourseDescription2
- CourseInstructorName2

In the example, columns for course information have been duplicated to allow the student to take two courses. The problem occurs when the student wants to take three course or more. While you could go ahead and add CourseID3, etc., to the table, the proper solution is to remove the repeating group of columns to another table.

If you have a set of columns in a table with field names that end in numbers xx1, xx2, xx3, etc., it's a clear warning signal that you have repeating groups in the table. A common exception

to this would be a table of street addresses, where you might have AddressLine1, AddressLine2, etc., rather than using a single field for multiple line addresses.

Let us revisit the design to handle the repeating groups:

Table 3.2: First Normal Form

Students	Student Courses
StudentID	StudentName
AdvisorName	SCStudentID
SCCourseID	SCCourseDescription
	SCCourseInstructorName

The primary keys are shown in *italics*.

We have divided the tables so that the student can now take as many courses as he wants by removing the course information from the original table and creating two tables: one for the student information and one for the course list. The repeating group of columns in the original table is gone, but we can still reconstruct the original table using the StudentID and SCStudentID columns from the two new tables. The new field SCStudentID is a foreign key to the Students table.

3.22.2 Second Normal Form (2NF)

A relation can be in 2NF only if it is in 1NF and every nonkey attribute is fully dependent on the primary key. 2NF only applies to relations in 1NF that have composite primary keys. 1NF relations with simple primary keys are automatically in 2NF (there can be no partial dependence on a simple primary key). However, in a 1NF relation that has a composite primary key, each nonkey attribute must depend on the entire primary key, not just one or two columns, to meet 2NF requirements.

To take a relation from 1NF to 2NF, it should be decomposed into two or more tables. Each table should contain an appropriate primary key and attributes that apply to that key.

Consider the relation ACTIVITY where a student can take more than one activity and an activity can have only one fee. To be in 2NF the relation the RDBMS must be in 1NF, and nonkey attributes cannot be partially dependent on the primary key.

RELATION = STU-ACT	
StuID	**Activity**
1001	Diving
1005	Softball
1005	Swimming
1006	Swimming
1010	Softball
Primary key = StuID-Activity	

RELATION = ACT-FEE	
Activity	**Fee**
Diving	150
Softball	50
Swimming	25
Primary key = Activity	

Fig. 3.31

Fee is the only nonkey attribute. A partial dependency occurs because Fee is actually functionally determined by Activity, not by StuID. Consider the possible anomalies:

- If line 1 were deleted, the fact that Diving costs $150 would be lost (deletion).
- The activity "Basketball" at $45 could not be added unless a student enrolled (insertion).
- If the cost of Softball changed to $60, each entry of Softball would have to be changed (update).

Other considerations: The relation has more than one theme and contains redundant data.

Solution:

Reevaluate each relation considering the anomalies above. Decomposing the relations into two separate relations removes anomalies, reduces redundancy, and allows easy updates.

RELATION = STU-ACT	
StuID	**Activity**
1001	Diving
1005	Softball
1005	Swimming
1006	Swimming
1010	Softball
Primary key = StuID-Activity	

RELATION = ACT-FEE	
Activity	**Fee**
Diving	150
Softball	50
Swimming	25
Primary key = Activity	

Fig. 3.32

2NF Definition

According to Date's book, a relation R is in 2NF if it is in 1NF and every nonkey attribute is fully functionally dependent on the primary key of R.

An alternative and better definition of 2NF is that a relation R is in 2NF if it is in 1NF and every non-prime attribute of R is fully functionally dependent on each candidate key of R. If a relation is not in 2NF it can be normalised into a number of 2NF relations using a reversible nonloss decomposition which is equivalent.

Another Example of Second Normal Form

No nonkey attributes depend on a portion of the primary key.

Second Normal Form really only applies to tables where the primary key is defined by two or more columns. The essense is that if there are columns which can be identified by only part of the primary key, they need to be in their own table.

Let us look at the sample tables for an example. In the StudentCourses table, the primary key is the combination of SCStudentID and SCCourseID. However, the table also contains the SCCourseDescription and the SCCourseInstructorName columns. These columns are only dependent on the SCCourseID column. In other words, the description and instructor's name will be the same regardless of the student. How do we resolve this problem? Let us revisit the sample tables.

Table 3.3: Second Normal Form

Students	Student Courses	Courses
StudentID	SCStudentID	CourseID
StudentName	SCCourseID	CourseDescription
AdvisorName		CourseInstructorName

What we have done is to remove the details of the course information to their own table Courses. The relationship between students and courses has at last revealed itself to be a many-to- many relationship. Each student can take many courses and each course can have many students. The StudentCourses table now contains only the two foreign keys to Students and Courses.

We are almost done normalizing this small sample, but before taking the last step, lets add a little more detail to the sample tables to make them look something more like the real world.

Table 3.4: Detail Columns Added

Students	Student Courses	Courses
StudentID	SCStudentID	CourseID
StudentName	SCCourseID	CourseDescription
StudentPhone	CourseInstructorName	
StudentAddress	CourseInstructorPhone	
StudentCity		
StudentState		
StudentZIP		
AdvisorName		
AdvisorPhone		

3.22.3 Third Normal Form (3NF)

A relation is in 3NF only if it is in 2NF and has no transitive dependencies. In other words, all nonkey attributes must be mutually independent (not dependent on any other nonkey attribute). If one nonkey attribute is dependent on another, the relation possibly contains data about more than one theme, contradicting the normalisation's "one theme per table" concept. Therefore, nonkey attributes should be functionally determined by the primary key only.

To take a relation from 2NF to 3NF, it should be decomposed into two or more tables with each containing only applicable attributes, or the conflicting attribute should be placed in a more appropriate table.

Consider the following relation where a student can live in only one building and a building can charge only one rental rate.

To be in 3NF the relation :

- Must be in 2NF; and
- Can have no transitive dependencies between nonkey attributes.

3NF Definition

According to Date's book, a relation R is in 3NF if:

- It is in 2NF and every nonkey attribute is non-transitive dependent on the primary key.
- All the nonkey attributes are mutually independent and fully functionally dependent on the primary key.
- This definition can lead to inconsistencies in the question of whether a BCNF relation is in 3NF and 2NF also. By (b) above the nonkey attributes and not mutually exclusive.
- Thus an alternative definition of 3NF is preferred.

3NF Alternative Definition

In this definition of 3NF, a relation R is in 3NF if it is in 2NF and none of the non prime attributes are transitivily dependent on any candidate keys.

This definition of 3NF is better than the other one as it does not assume that there is only one candidate key. However, there are still some inadequacies with this form of 3NF which led to the defn of BCNF.

3NF Mutual Independence

Mutual independence between attributes in the definition of 3NF means that none of the attributes are functionally dependent on any of the others.

Another Example of a Third Normal Form

- No attributes depend on other nonkey attributes.

This means that all the columns in the table contain data about the entity that is defined by the primary key. The columns in the table must contain data about only one thing. This is really an extension of Second Normal Form - both are used to remove columns that belong in their own table.

To complete the normalisation, we need to look for columns that are not dependent on the primary key of the table. In the Students table, we have two data items about the student's advisor: the name and phone number. The balance of the data pertains only to the student and so is appropriate in the Students table. The advisor information, however, is not dependent on the student. If the student leaves the school, the advisor and the advisor's phone number will remain the same. The same logic applies to the instructor information in the Courses table. The data for the instructor is not dependent on the primary key CourseID since the instructor will be unaffected if the course is dropped from the curriculum.

Let us complete the normalisation for the sample tables.

Table 3.5: Third Normal Form

Students	Advisors	Instructors	StudentCourses	Courses
StudentID	AdvisorID	InstructorID	SCStudentID	CourseID
StudentName	AdvisorName	InstructorName	SCCourseID	CourseDescription
StudentPhone	AdvisorPhone	InstructorPhone	CourseInstructorID	
StudentAddress				
StudentCity				
StudentState				
StudentZIP				
StudentAdvisorID				

There is one other potential modification that could be made to this design. If you look at the Advisors and Instructors tables, you can see that the columns are essentially the same: a name and phone number. These two tables could be combined into a single common table called Staff or Faculty. The advantage of this approach is that it can make the design simpler by using one less table. The disadvantage is that you may need to record different additional details for instructors than you would record for advisors. One possible way of resolving this conflict would be to go one further step and create a Staff table that records basic information that any organisation would retain about an employee, such as name, address, phone, Social Security number, date of birth, etc. Then you would have the Advisors and Instructors tables contain foreign keys to the Staff table for appropriate individual along with any additional details that might need to be stored specifically for that particular role. The benefit of this approach is that if a member of the staff is both an advisor and an instructor (this would often be the case in a college or university), the basic information would not need to be duplicated for both roles.

Don't go overboard with Third Normal Form or you'll wreak havoc on performance. If you look at the Students table, you can see that in fact any city and state in the U.S. could be identified by the ZIP code. However, it may not be practical to design the database so that every time you need to get an address you have to join the row from Students to a table containing the (approximately) 65,000 ZIP codes in the U.S. A general guideline is that if you routinely run queries which join more than four tables, you may need to consider denormalizing the design.

3NF Inadequacies

3NF in either definition cannot handle relations with multiple candidate keys where these candidate keys are composite and overlapping.

Such situations do not tend to occur too often in practice, but where they do, Boyce-Codd normal form (BCNF) has been defined to eliminate possible update anomalies.

3.22.4 Boyce-Codd Normal Form (BCNF)

BCNF was developed to overcome some problems with 3NF. A relation meets the requirements of BCNF if every determinant is a candidate key. Remember, a determinant is an attribute on which another attribute is fully functionally dependent.

BCNF Definition

A relation R is in BCNF if and only if every determinant is a candidate key. This is a simpler definition than 3NF or the alternative 3NF as it does not involve 1NF, or 2NF or transitive dependencies. Any relation R can be nonloss decomposed into an equivalent collection of BCNF relation.

3.22.5 Fourth Normal Form (4NF)

A relation meets the requirements of 4NF only if it is in 3NF (or BCNF) and has no multivalued dependencies. Multivalued dependencies exist only in relations containing at least three attributes. The problem arises when a single value of the primary key returns multiple records for two or more attributes that are mutually independent. Often this mean that the relation contains multiple themes. Relations with multivalued dependencies can cause considerable data redundancy and can make data updates difficult.

To take a relation from 3NF to 4NF, it must be decomposed into two or more relations. Each relation should address only one theme.

Consider the following relation where a student can have more than one major and can take more than one activity.

To be in 4NF the relation :

- Must be in 3NF, and
- Can have no multivalued dependencies.

RELATION = STUDENT

StuID	Major	Activity
100	Math	Skiing
100	Business	Skiing
100	Math	Softball
100	Business	Softball
150	Art	Racquetball

Fig. 3.35

Notice that a single value of the primary key returns multiple records for two or more mutually independent attributes. StuID 100 has two majors (math and business) and two activities (skiing and softball). The problem exists because each occurrence of Major must be recorded with each occurrence of Activity.

Consider the possible anomalies:

- If line 5 were deleted, the fact that Art and Racquetball exist would be lost (deletion).
- Neither a new Major nor Activity could be added unless a student enrolled in both a Major and an Activity (insertion).
- If student 100 changed Majors or Activities, each entry would have to be changed (update).

Other considerations: The relation has more than one theme and contains considerable redundant data.

Solution:

Reevaluate each relation considering the anomalies above. Decomposing the relations into two separate relations removes anomalies, reduces redundancy, and allows easy updates.

RELATION = STU-MAJOR	
StuID	**Major**
100	Math
100	Business
150	Art
Primary key = StuID-Major	

RELATION = STU-ACT	
StuID	**Activity**
100	Skiing
100	Softball
150	Racquetball
Primary key = StuID-Activity	

Fig. 3.36

4NF Definition

If a relation R has three or more attributes grouped as A, B, or C, and

$R \rightarrow\rightarrow R.B$ and $R \rightarrow\rightarrow R.C$

holds (i.e. there are multivalued dependencies present) then R is not in 4NF. R is in 4NF if and only if it is in BCNF and all multivalued dependencies are functional dependencies.

3.22.6 5NF Definition

A relation R is in 5NF (also called project/join normal form) if and only if every join dependency in R is a consequence of the candidate keys of R i.e. is not implied by the candidate keys of R. If R is in 5NF it is also in 4NF.

Discovering join dependencies is not easy as unlike functional dependencies and multivalued dependencies they do not have a straightforward real world interpretation.

Additionally, for a database to be in second normal form, it must also be in first normal form, and for a database to be in third normal form, it must meet the requirements for both first and second normal forms. There are also additional forms of normalisation, but these are rarely applied. In fact, it may at times be practical to violate even the first three forms of normalisation.

3.23 Normalisation Theory

In normalisation theory, there are a hierarchy of normal forms into which base tables should be put. Each normal form has a set of satisfiable criteria based on functional dependence and full functional dependence. Starting with first normal form, tables can be decomposed as:

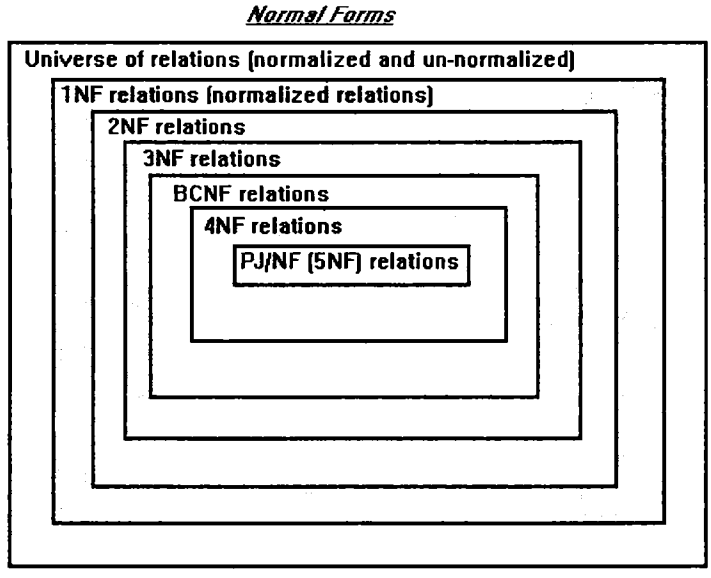


Fig. 3.37: Normal Forms

3.23.1 Functional Dependence

In the definition of normal forms in relational databases, given relation R, then attribute y of R is functionally dependent on attribute x of R, or R.x functionally determines R.y (denoted R.x $\rightarrow$ R.y) if and only if each R.x has associated with it precisely one R.y where x and y may be composite attributes. If R.x is the primary key of R, or even if it is a candidate key, then by definition all R.y's must be functionally dependent on R.x. Functional dependence is not the same as full functional dependence, and is a property of the meaning or the semantics of data.

3.23.2 Functional Determinant

A functional determinant is an attribute or a composite attribute in one table in a relational database, whose values are functionally dependent on another possibly composite attributes of that table. There is no requirement in the definition of functional dependence that a functional determinant has to be a candidate key, though all functional determinants are at least candidate keys.

3.23.3 Composite Attributes

A composite attribute in a relational database is a combination of two or more attributes, or columns, from a single base table, which may functionally determine another, possibly composite, attribute or may be a functional determinant.

3.23.4 Multivalued Dependencies

1NF disallows a set or list of values as a value for an attribute in a tuple and if we have two or more multivalued independent attributes in the same base relation, we have to repeat every value of one attribute for every value of the other. Such a dependency is called a multivalued dependency and because redundant information is stored, a relation with a multivalued dependency can have problems of update anomalies.

Multivalued dependencies can exist on relations which have three or more relations, and we say

R.A $\rightarrow\rightarrow$ R.B

if and only if the set of R.B values match the pairs (A values, C values) in R and R.A $\rightarrow\rightarrow$ R.C also holds. R.A, R.B, and R.C, may be composite. Multivalued dependencies may be eliminated by nonloss decomposition into equivalent 4NF relations.

3.23.5 General Update Anomalies

One goal of schema design in normalisation is to minimise the storage space that the base relations (files) occupy. Attribute values pertaining to a particular value should not be repeated, but should appear only once. If given values are repeated more than once then this can cause update anomalies. These can be classified into insertion anomalies, deletion anomalies, and modification anomalies.

3.23.6 Review of Normal Forms

- FIRST NORMAL FORM

First normal form deals with the "shape" of a record type.

Under first normal form, all occurrences of a record type must contain the same number of fields.

First normal form excludes variable repeating fields and groups. This is not so much a design guideline as a matter of definition. Relational database theory doesn't deal with records having a variable number of fields.

- SECOND AND THIRD NORMAL FORMS

Second and third normal forms deal with the relationship between non-key and key fields.

Under second and third normal forms, a non-key field must provide a fact about the key, us the whole key, and nothing but the key. In addition, the record must satisfy first normal form.

We deal now only with "single-valued" facts. The fact could be a one-to-many relationship, such as the department of an employee, or a one-to-one relationship, such as the spouse of an employee. Thus the phrase "Y is a fact about X" signifies a one-to-one or one-to-many relationship between Y and X. In the general case, Y might consist of one or more fields, and so might X. In the following example, QUANTITY is a fact about the combination of PART and WAREHOUSE.

- Second Normal Form

Second normal form is violated when a non-key field is a fact about a subset of a key. It is only relevant when the key is composite, i.e., consists of several fields. Consider the following inventory record:

PART	WAREHOUSE	QUANTITY	WAREHOUSE-ADDRESS

The key here consists of the PART and WAREHOUSE fields together, but WAREHOUSE-ADDRESS is a fact about the WAREHOUSE alone. The basic problems with this design are:

- The warehouse address is repeated in every record that refers to a part stored in that warehouse.
- If the address of the warehouse changes, every record referring to a part stored in that warehouse must be updated.

Because of the redundancy, the data might become inconsistent, with different records showing different addresses for the same warehouse.

If at some point in time there are no parts stored in the warehouse, there may be no record in which to keep the warehouse's address.

To satisfy second normal form, the record shown above should be decomposed into (replaced by) the two records:

PART	WAREHOUSE	QUANTITY	WAREHOUSE	WAREHOUSE-ADDRESS

When a data design is changed in this way, replacing unnormalized records with normalized records, the process is referred to as normalisation. The term "normalisation" is sometimes used relative to a particular normal form. Thus a set of records may be normalized with respect to second normal form but not with respect to third.

The normalized design enhances the integrity of the data, by minimizing redundancy and inconsistency, but at some possible performance cost for certain retrieval applications. Consider an application that wants the addresses of all warehouses stocking a certain part. In the unnormalized form, the application searches one record type. With the normalized design, the application has to search two record types, and connect the appropriate pairs.

Third Normal Form

The third normal form is violated when a non-key field is a fact about another non-key field, as in

EMPLOYEE	DEPARTMENT	LOCATION

The EMPLOYEE field is the key. If each department is located in one place, then the LOCATION field is a fact about the DEPARTMENT in addition to being a fact about the EMPLOYEE. The problems with this design are the same as those caused by violations of second normal form:

- The department's location is repeated in the record of every employee assigned to that department.
- If the location of the department changes, every such record must be updated.
- Because of the redundancy, the data might become inconsistent, with different records showing different locations for the same department.
- If a department has no employees, there may be no record in which to keep the department's location.

To satisfy the third normal form, the record shown above should be decomposed into the two records:

EMPLOYEE	DEPARTMENT	DEPARTMENT	LOCATION

To summarize, a record is in second and third normal forms if every field is either part of the key or provides a (single- valued) fact about exactly the whole key and nothing else.

- Functional Dependencies

In relational database theory, second and third normal forms are defined in terms of functional dependencies, which correspond approximately to our single-valued facts. A field Y is "functionally dependent" on a field (or fields) X if it is invalid to have two records with the same X-value but different Y-values. That is, a given X-value must always occur with the same Y-value. When X is a key, then all fields are by definition functionally dependent on X in a trivial way, since there can't be two records having the same X value.

There is a slight technical difference between functional dependencies and single-valued facts as we have presented them. Functional dependencies only exist when the things involved have unique and singular identifiers (representations). For example, suppose a person's address is a single-valued fact, i.e., a person has only one address. If we don't provide unique identifiers for people, then there will not be a functional dependency in the data:

PERSON	ADDRESS
John Smith	123 Main St., New York
John Smith	321 Center St., San Francisco

Although each person has a unique address, a given name can appear with several different addresses. Hence we do not have a functional dependency corresponding to our single-valued fact.

Similarly, the address has to be spelled identically in each occurrence in order to have a functional dependency. In the following case the same person appears to be living at two different addresses, again precluding a functional dependency.

PERSON	ADDRESS
John Smith	123 Main St., New York
John Smith	123 Main Street, NYC

We are not defending the use of non-unique or non-singular representations. Such practices often lead to data maintenance problems of their own. We do wish to point out, however, that functional dependencies and the various normal forms are really only defined for situations in which there are unique and singular identifiers. Thus the design guidelines as we present them are a bit stronger than those implied by the formal definitions of the normal forms.

For instance, we as designers know that in the following example there is a single-valued fact about a non-key field, and hence the design is susceptible to all the update anomalies mentioned earlier.

EMPLOYEE	FATHER	FATHER'S-ADDRESS
Arun Sethi	Prem Sethi	123, Gole Mkt., New Delhi.
Varun Sethi	Prem Sethi	123 Gole Mkt., New Delhi.
Atul Roy	Suresh Roy	A/II, 321 Janak Puri, New Delhi

However, in formal terms, there is no functional dependency here between FATHER'S-ADDRESS and FATHER, and hence no violation of third normal form.

- FOURTH AND FIFTH NORMAL FORMS

Fourth and fifth normal forms deal with multi-valued facts. The multi-valued fact may correspond to a many-to-many relationship, as with employees and skills, or to a many-to-one relationship, as with the children of an employee (assuming only one parent is an employee). By "many-to-many" we mean that an employee may have several skills, and a skill may belong to several employees.

Note that we look at the many-to-one relationship between children and fathers as a single-valued fact about a child but a multi-valued fact about a father.

In a sense, fourth and fifth normal forms are also about composite keys. These normal forms attempt to minimize the number of fields involved in a composite key, as suggested by the examples to follow.

- Fourth Normal Form

Under fourth normal form, a record type should not contain two or more independent multi-valued facts about an entity. In addition, the record must satisfy third normal form.

The term "independent" will be discussed after considering an example.

Consider employees, skills, and languages, where an employee may have several skills and several languages. We have here two many- to-many relationships, one between employees and skills, and one between employees and languages. Under fourth normal form, these two relationships should not be represented in a single record such as:

EMPLOYEE	SKILL	LANGUAGE

Instead, they should be represented in the two records:

EMPLOYEE	SKILL	EMPLOYEE	LANGUAGE

Note that other fields, not involving multi-valued facts, are permitted to occur in the record, as in the case of the QUANTITY field in the earlier PART/WAREHOUSE example.

The main problem with violating fourth normal form is that it leads to uncertainties in the maintenance policies. Several policies are possible for maintaining two independent multi-valued facts in one record:

1. A disjoint format, in which a record contains either a skill or a language, but not both:

EMPLOYEE	SKILL	LANGUAGE
Sumit	cook	
Sumit	type	
Sumit		Hindi
Sumit		English
Sumit		French

This is not much different from maintaining two separate record types. (We note in passing that such a format also leads to ambiguities regarding the meanings of blank fields. A blank SKILL could mean the person has no skill, or the field is not applicable to this employee, or the data is unknown, or, as in this case, the data may be found in another record).

2. A random mix, with three variations:

   a. Minimal number of records, with repetitions:

EMPLOYEE	SKILL	LANGUAGE
Sumit	cook	Hindi
Sumit	type	English
Sumit	type	French

   b. Minimal number of records, with null values:

EMPLOYEE	SKILL	LANGUAGE
Sumit	cook	Hindi
Sumit	type	English
Sumit		French

   c. Unrestricted:

EMPLOYEE	SKILL	LANGUAGE
Sumit	cook	French
Sumit	type	
Sumit		English
Sumit	type	Hindi

3. A "cross-product" form, where for each employee, there must be a record for every possible pairing of one of his skills with one of his languages:

EMPLOYEE	SKILL	LANGUAGE
Sumit	cook	French
Sumit	cook	English
Sumit	cook	Hindi
Sumit	type	French
Sumit	type	English
Sumit	type	Hindi

Other problems caused by violating fourth normal form are similar in spirit to those mentioned earlier for violations of second or third normal form. They take different variations depending on the chosen maintenance policy:

- If there are repetitions, then updates have to be done in multiple records, and they could become inconsistent.
- Insertion of a new skill may involve looking for a record with a blank skill, or inserting a new record with a possibly blank language, or inserting multiple records pairing the new skill with some or all of the languages.
- Deletion of a skill may involve blanking out the skill field in one or more records (perhaps with a check that this doesn't leave two records with the same language and a blank skill), or deleting one or more records, coupled with a check that the last mention of some language hasn't also been deleted. Fourth normal form minimizes such update problems.
- Independence

We mentioned independent multi-valued facts earlier, and we now illustrate what we mean in terms of the example. The two many-to- many relationships, employee:skill and employee:language, are "independent" in that there is no direct connection between skills and languages. There is only an indirect connection because they belong to some common employee. That is, it does not matter which skill is paired with which language in a record; the pairing does not convey any information. That's precisely why all the maintenance policies mentioned earlier can be allowed.

In contrast, suppose that an employee could only exercise certain skills in certain languages. Perhaps Smith can cook French cuisine only, but can type in French, English, and Hindi. Then the pairings of skills and languages becomes meaningful, and there is no longer an ambiguity of maintenance policies. In the present case, only the following form is correct:

EMPLOYEE	SKILL	LANGUAGE
Sumit	cook	French
Sumit	type	French
Sumit	type	English
Sumit	type	Hindi

Thus the employee:skill and employee:

language relationships are no longer independent. These records do not violate fourth normal form. When there is an interdependence among the relationships, then it is acceptable to represent them in a single record.

- Multivalued Dependencies

For readers interested in pursuing the technical background of fourth normal form a bit further, we mention that fourth normal form is defined in terms of multivalued dependencies, which correspond to our independent multi-valued facts. Multivalued dependencies, in turn, are defined essentially as relationships which accept the "cross-product" maintenance policy mentioned above. That is, for our example, every one of an employee's skills must appear paired with every one of his languages. It may or may not be obvious to the reader that this is equivalent to our notion of independence: since every possible pairing must be present, there is no "information" in the pairings. Such pairings convey information only if some of them can be absent, that is, only if it is possible that some employee cannot perform some skill in some language. If all pairings are always present, then the relationships are really independent.

We should also point out that multivalued dependencies and fourth normal form apply as well to relationships involving more than two fields. For example, suppose we extend the earlier example to include projects, in the following sense:

- An employee uses certain skills on certain projects.
- An employee uses certain languages on certain projects.

If there is no direct connection between the skills and languages that an employee uses on a project, then we could treat this as two independent many-to-many relationships of the form EP:S and EP:L, where "EP" represents a combination of an employee with a project. A record including employee, project, skill, and language would violate fourth normal form. Two records, containing fields E,P,S and E,P,L, respectively, would satisfy the fourth normal form.

- Fifth Normal Form

The fifth normal form deals with cases where information can be reconstructed from smaller pieces of information that can be maintained with less redundancy. Second, third, and fourth normal forms also serve this purpose, but the fifth normal form generalizes to cases not covered by the others.

We will not attempt a comprehensive exposition of fifth normal form, but illustrate the central concept with a commonly used example, namely one involving agents, companies, and products. If agents represent companies, companies make products, and agents sell products, then we might want to keep a record of which agent sells which product for which company. This information could be kept in one record type with three fields:

AGENT	COMPANY	PRODUCT
Sumit	Ford	car
Sumit	GM	truck

This form is necessary in the general case. For example, although agent Sumit sells cars made by Ford and trucks made by GM, he does not sell Ford trucks or GM cars. Thus we need the combination of three fields to know which combinations are valid and which are not.

But suppose that a certain rule was in effect: if an agent sells a certain product, and he represents a company making that product, then he sells that product for that company.

AGENT	COMPANY	PRODUCT
Sumit	Ford	car
Sumit	Ford	truck
Sumit	GM	car
Sumit	GM	truck
Vikram	Ford	car

In this case, it turns out that we can reconstruct all the true facts from a normalized form consisting of three separate record types, each containing two fields:

AGENT	COMPANY
Sumit	Ford
Sumit	GM
Vikram	Ford

COMPANY	PRODUCT
Ford	car
Ford	truck
GM	car
GM	truck

AGENT	PRODUCT
Sumit	car
Sumit	truck
Vikram	car

These three record types are in fifth normal form, whereas the corresponding three-field record shown previously is not.

Roughly speaking, we may say that a record type is in fifth normal form when its information content cannot be reconstructed from several smaller record types, i.e., from record types each having fewer fields than the original record. The case where all the smaller records have the same key is excluded. If a record type can only be decomposed into smaller records which all have the same key, then the record type is considered to be in fifth normal form without decomposition. A record type in fifth normal form is also in fourth, third, second, and first normal forms.

The fifth normal form does not differ from fourth normal form unless there exists a symmetric constraint such as the rule about agents, companies, and products. In the absence of such a constraint, a record type in fourth normal form is always in fifth normal form.

One advantage of the fifth normal form is that certain redundancies can be eliminated. In the normalized form, the fact that Smith sells cars is recorded only once; in the unnormalized form it may be repeated many times.

It should be observed that although the normalized form involves more record types, there may be fewer total record occurrences. This is not apparent when there are only a few facts to record, as in the example shown above. The advantage is realized as more facts are recorded, since the size of the normalized files increases in an additive fashion, while the size of the unnormalized file increases in a multiplicative fashion. For example, if we add a new agent who sells x products for y companies, where each of these companies makes each of these products, we have to add x+y new records to the normalized form, but xy new records to the unnormalized form.

It should be noted that all three record types are required in the normalized form in order to reconstruct the same information. From the first two record types shown above we learn that Vikram represents Ford and that Ford makes trucks. But we can't determine whether Vikram sells Ford trucks until we look at the third record type to determine whether Vikram sells trucks at all.

The following example illustrates a case in which the rule about agents, companies, and products is satisfied, and which clearly requires all three record types in the normalized form. Any two of the record types taken alone will imply something untrue.

AGENT	COMPANY	PRODUCT
Sumit	Ford	car
Sumit	Ford	truck
Sumit	GM	car
Sumit	GM	truck
Vikram	Ford	car
Vikram	Ford	truck
Bobby	Ford	car
Bobby	GM	car
Bobby	Totota	car
Bobby	Totota	bus

AGENT	COMPANY
Sumit	Ford
Sumit	GMs
Vikram	Ford
Bobby	Ford
Bobby	GM
Bobby	Toyota

COMPANY	PRODUCT
Ford	car
Ford	truck
GM	car
GM	truck
Toyota	car
Toyota	bus

AGENT	PRODUCT
Sumit	car
Sumit	truck
Vikram	car
Vikram	truck
Bobby	car
Bobby	bus

Observe that:

- Vikarm sells cars and GM makes cars, but Vikram does not represent GM.
- Bobby represents Ford and Ford makes trucks, but Bobby does not sell trucks.
- Bobby represents Ford and Bobby sells buses, but Ford does not make buses.

the fourth and the fifth normal forms both deal with combinations of multivalued facts. One difference is that the facts dealt with under the fifth normal form are not independent, in the sense discussed earlier. Another difference is that, although the fourth normal form can deal with more than two multivalued facts, it only recognizes them in pairwise groups. We can best explain this in terms of the normalisation process implied by the fourth normal form. If a record violates fourth normal form, the associated normalisation process decomposes it into two records, each containing fewer fields than the original record. Any of these violating the fourth normal form is again decomposed into two records, and so on until the resulting records are all in fourth normal form. At each stage, the set of records after decomposition contains exactly the same information as the set of records before decomposition.

In the present example, no pairwise decomposition is possible. There is no combination of two smaller records which contains the same total information as the original record. All three of the smaller records are needed. Hence an information-preserving pairwise decomposition is not possible, and the original record is not in violation of fourth normal form. The fifth normal form is needed in order to deal with the redundancies in this case.

- Unavoidable Redundancies

Normalisation certainly doesn't remove all redundancies. Certain redundancies seem to be unavoidable, particularly when several multivalued facts are dependent rather than independent. The example show it seems unavoidable that we record the fact that "Sumit can type" several times. Also, when the rule about agents, companies, and products is not in effect, it seems unavoidable that we record the fact that "Sumit sells cars" several times.

- Inter-Record Redundancy

The normal forms discussed here deal only with redundancies occurring within a single record type. The fifth normal form is considered to be the "ultimate" normal form with respect to such redundancies'.

Other redundancies can occur across multiple record types. For the example concerning employees, departments, and locations, the following records are in the third normal form in spite of the obvious redundancy:

EMPLOYEE	DEPARTMENT	DEPARTMENT	LOCATION

EMPLOYEE	LOCATION

In fact, two copies of the same record type would constitute the ultimate in this kind of undetected redundancy.

Inter-record redundancy has been recognised for some time, and has recently been addressed in terms of normal forms and normalisation.

3.24 Normalisation Guidelines

Given a relation R in 1NF and a set of functional dependencies multivalued dependencies and join dependencies we systematically reduce these to a collection of smaller, more desirable relations by taking projections, thus eliminating any possible update anomalies.

These are guidelines only, and database designers do not have to conform to these guidelines. There may be practical reasons based on anticipated usage of a database, for not normalisating a database schema to as high a normal form as it can go.

3.24.1 Database Schema Design

The purpose of database schema design is to decide on a suitable logical structure, and this is usually based on the principles behind normalisation theory. The basic axiom is that one semantic real-world fact should be stored in one place if possible in order to avoid update anomalies. Normalisation theory suggests a set of guidelines for good database design.

3.24.2 Clustering

Clustering involves trying to store logically related records physically close together on the disk. This is an important factor for performance. A relational DBMS supports such clustering by allowing the creation of clustered indexes – this is why it needs to know about pages and files. The DBA informs the DBMS what type of clustering to support. The DBA should be allowed to vary the clusterings in mid-file, but all this should be transperent to the users. However, if data independence is to be achieved, any such changes in physical clustering should not require and concomitant changes in application programs. Knowledge of how the data is to be used is essential to producing a good physical database design.

3.25 Advantages of Optimisation

The major advantage of query optimisation is that users do not have to worry about how best to state their queries (i.e. how to phrase requests in order to get the best performance out of the system), and also there is a real possibility that the optimiser will do better than a human programmer in terms of using the DBMS efficiently. There are several reasons for this:

- The optimiser will have more information available than the user.
- Database statistics may change
- The optimiser is a program
- The optimiser embodies the skills of the best programmers
- All of the above serve as evidence in support of the claim that optimisability i.e. the fact that relational database requests are optimisable, is in fact a strength of relational systems.

3.26 Indexing

In order to get acceptable performance from a database application, you not only need to write efficient code, you also need to provide the database engine with the tools it needs to do its job well. The most important step you can take in that direction is the proper use of indexes.

3.26.1 Introduction to Indexing

Like an index in a book, an index on a table provides a means to rapidly locate specific information – but a table index is used by the database engine rather than the user. The proper use of indexes can yield significant performance improvements in a database application because indexes can assist the database engine in avoiding costly disk i/o when searching a table for specific data. Most importantly, proper indexing can prevent the database engine from using the most costly of database operations: the table scan.

Database engines often employ a form of a binary search algorith when retrieving rows from a table. If you are familiar with the binary search, you know that a binary search can isolate a specific item in a collection of thousands of items with only a handful of comparisons. In a database application, comparisons become seeks in a disk file (or a disk cache if you are lucky). File seeks are inherently expensive since even the fastest disks on the market today are orders of magnitude slower than RAM operations. A primary concern for improving performance is to minimize the number of disk reads when querying the database. Well chosen indexes are the key to reducing file i/o operations because they reduce the number of data pages which must be read to locate a row in a table. When searching for a row, the database engine can read a small number of pages in the index rather than reading every row in the table until it finds a match.

It is important to remember, however, that just as an index will improve read performance, it will degrade write performance. When a new row is written to a table, the database engine must not only write the row, it must also update any associated indexes. If you decided to maximize read performance by indexing every column in a table and creating a multiple column index for every possible search or sort, performance of inserts, updates, and deletes will come to a grinding halt.

3.26.2 Types of Indexes

Nearly every database engine available will support two fundamental types of indexes.

- Unique Indexes

  There are no duplicate entries in a unique index. Unique indexes are most often used for the primary key of a table.
- Non-unique Indexes

  Non-unique indexes may have duplicate values and are used anywhere than an index will provide a performance improvement in the application.

Although it is the most common use, a unique index does not necessarily need to be the primary key of a table. A technique we have used to simplify query designs is to specify a unique index on a combination of columns that could serve as a primary key, then add an autonumbering column to the table and specify that column as the primary key. This makes queries simpler because only one column needs to be joined. Additionally, it can help eliminate redundancy since only one value would need to be stored as a foreign key to the table. I've never analysed the performance impact of adding an extra column for this purpose, but in some situations, a performance cost (if one in fact exists) is worth the price in the simplicity of the design.

Some database engines can also create clustered or non-clustered indexes.

- Clustered Index - In a clustered index, the actual rows in the table reside on the leaf pages of the index.
- Non-clustered Index - In a non-clustered index, the leaf pages of the index are pointers to the data pages containing the rows in the table.

Clustered indexes can offer significant performance advantages over non-clustered indexes if your database engine supports them. Since the actual table data resides at the lowest level of the index, there is always one less seek involved in the file. However, keep in mind that there can only be one clustered index per table, so if a clustered index is to be used, it must be chosen carefully. Additionally, dropping or changing a clustered index is a costly operation because the data in the table must be entirely rewritten.

Another specification that is sometimes available is ascending or descending indexes. This is as simple as it sounds. In an ascending index, the index is written in ascending order. In a descending index, its written in descending order. Not all database engines can create descending indexes (SQL Server doesn't), but most can use an ascending index to speed up a descending sort.

One more consideration in creating an index is the nullability of the index. This may be driven by the index definition or simply inherited from the definitions of the underlying columns. It should be noted, however, that the treatment of indexes which allow nulls can vary from one database engine to another. For example, if you create a unique index in a Jet database which allows nulls, Jet will essentially ignore the null values from the index and allow any number of rows with null entries. Contrast that with MS SQL Server, where if an index allows nulls, only one null entry is permitted in a unique index. This applies only to primarily to multicolumn indexes since it would be rare to create a nullable unique index on a single column. To illustrate this, let's look at an example of the difference.

Column A	Column B	SQL Server Result	Jet Result
1	1	Allowed - unique entry	Allowed - unique entry
1	Null	Allowed - unique entry	Allowed - unique entry
2	1	Allowed - unique entry	Allowed - unique entry
2	Null	Allowed - unique entry	Allowed - unique entry
2	Null	Disallowed - duplicate value in unique index	Allowed - duplicate null ignored

3.26.3 What to Index

The following list indicates the types of data which are good candidates for indexing.

- Columns used in joins

  Joins will almost always benefit from having an index available on both sides of the join.
- Columns used in a query WHERE clause

If a column is part of the selection criteria for a frequently run query, an index may improve query performance. For less frequently used queries, you will need to consider the cost in terms of inserts, updates, and deletes against the gain in select queries.

- Columns used in a query ORDER BY clause

Sort performance can be improved substantially by indexing the

- Columns used in the ORDER BY clause of the query. However, sorts are inherently slow unless a clustered index is used for the sort (in which case the data will be stored in the sorted order).
- Columns used in a query GROUP BY clause

Grouping operations will be enhanced by indexing especially in situations where the range of values being grouped is small in relation to the number of rows in the table.

3.26.4 What Not to Index

This list indicates columns that may not benefit from indexing.

- Tables with a small number of rows

If a table has only a handful of rows, the query optimizer for the database engine might determine that a table scan is more efficient than using an index. In this case, the index would only serve to slow down inserts, updates, and deletes in the table.

- Columns with a wide range of values

If the data in a column is different for nearly every row (such as a table of addresses) the index may not be useful because the database engine might determine that scanning the table is more economical than traversing such a large index. An exception would be using a clustered index on a column or columns that are used in sorts. If a clustered index is created using the same sorting order as a query, the data would be stored in sorted order in the table.

- Tables with heavy transaction loads but limited decision support load

If a table has a lot of insert, update, and delete activity but there are few SELECT queries run against it, added indexes will probably result in a net penalty in overall performance of the application.

- Columns not used in queries

Columns which are rarely retrieved do not need to be indexed since the index only enhances performance for SELECT queries where the column is part of the WHERE, GROUP BY, or ORDER BY clause. Even if the column is part of many queries but is never included in criteria, sorting, etc., it will not benefit from an index.

3.27 Oracle Realational Databases

A database is a structured collection of data. Data refers to the characteristics of people, things, and events. Oracle stores each data item in its own field. For example, a person's first name, date of birth, and their postal code are each stored in separate fields. The name of a field usually reflects its contents. A postal code field might be named POSTAL-CODE or PSTL_CD. Each DBMS has its own rules for naming the data fields.

A field has little meaning unless it is seen within the context of other fields. The postal code T6G 2H1, for example, expresses nothing by itself. To what person or business does it belong? The postal code field is informative only after it is associated with other data. In Oracle, the fields relating to a particular person, thing, or event are bundled together to form a single, complete unit of data, called a record (it can also be referred to as a row or an occurrence). Each record is made up of a number of fields. No two fields in a record can have the same field name.

During an Oracle database design project, the analysis of your business needs identifies all the fields or attributes of interest. If your business needs change over time, you define any additional fields or change the definition of existing fields.

3.27.1 Oracle Tables

Oracle stores records relating to each other in a table. For example, all the records for employees of a company would be stored by Oracle in one table, the employee table. A table is easily visualized as a tabular arrangement of data, not unlike a spreadsheet, consisting of vertical columns and horizontal rows.

EMPLOYEE

EMPL_ID	NAME_FIRST	NAME_LAST	DEPT	POSITION	SALARY
1	Himanshu	Sharma	1	Owner	29
4	Deepak	Choudhary	2	Excavator	13
5	Raman	Kumar	2	Laborer	12
7	Ish	Mjumdar	3	Laborer	11
11	Rashmi	Garg	1	Secretary	12

A table consists of a number of records. The field names of each record in the table are the same, although the field values may differ. Every employee record has a salary field, called SALARY. The values in the SALARY field can be different for each employee.

Each field occupies one column and each record occupies one row. In each column of the table, you put a specific category of information for the employees, such as their ID number, first name, and position. Each row in the table contains the information relating to a specific employee, together as one record. Each record is a unique entry and is independent of any other record in the table. The EMPLOYEE table, for example, contains records for Raman Kumar and Rashmi Garg. Although both records are part of the EMPLOYEE table, the data contained within them is independent of each other. There is no relationship between Raman's and Rashmi's salaries.

After the analysis of the business requirements, the database design team defines the necessary tables. Different tables are created for the various groups of information. An EMPLOYEE table is created for employee information, a DEPARTMENT table is created for department information. Related tables are grouped together to form a database. For example, a personnel or human resources application database includes both the EMPLOYEE and DEPARTMENT tables and all other tables involved in the application.

3.27.2 Primary Keys

Every table in Oracle has a field or a combination of fields that uniquely identifies each record in the table. This unique identifier is called the primary key, or simply the key. The primary key provides the means to distinguish one record from all the others in a table. It allows the user and the database system to identify, locate, and refer to one particular record in the table.

The database design team determines the best candidate field for the primary key. The employee's first and last names together could be a primary key, that is until a new employee with the same name is hired. Then the key would no longer be unique. Sometimes the design team has to define a new ID number or code field, just so that a table has a primary key. For the EMPLOYEE table, the primary key would likely be the employee ID number.

Once a table has been assigned a primary key, Oracle won't allow more than one record in the table with the same value for the primary key. No two employees can have the same ID number.

3.27.3 Relational Databases in Oracle

Sometimes all the information of interest to a business operation can be stored in one table. For example, let us say the only data you need to maintain about your office supplies is a description of each item, its supplier, and the quantity on hand. It would be enough to have one office supply table with those data items as the fields. More often, though, business applications involve many tables. In a typical personnel application, there might be one table for employees, another for information about their hours of work, and another for the departments in the company.

Oracle makes it very easy to link the data in multiple tables: matching an employee to the department in which they work is one example. This is what makes Oracle a relational database management system, or RDBMS. It stores data in two or more tables and enables you to define relationships between the tables. The link is based on one or more fields common to both tables.

For example, the following diagram represents part of the EMPLOYEE table and the entire DEPARTMENT table:

EMPLOYEE		DEPARTMENT			
EMPL_ID	NAME_LAST	DEPT	DEPT_NO	DESCRIPTION	LOCATION
1	Sharma	1	1	Administration	New Delhi
4	Choudhary	2	2	Quarry	New Delhi |
5	Kumar	2	3	Stockpile	Noida
7	Majumdar	3			
11	Garg	1			

There is a department number field in both the EMPLOYEE and DEPARTMENT tables. In the EMPLOYEE table, the department number represents the department in which the employee works. In the DEPARTMENT table, the department number represents a valid department within the business. In both tables, they are department numbers; in essence, the contents of the DEPT field in the EMPLOYEE table represents the same thing as the contents of the DEPT_NO field in the DEPARTMENT table. It is not necessary that the linking fields have the same field names. What's important is their value and what they represent.

The business is divided into departments. The departments are identified and stored in the DEPARTMENT table. Each department is assigned a department number. The relationship between the EMPLOYEE and DEPARTMENT tables is based on the department number. Each employee works in one specific department. The employee's department number is stored in the DEPT field of the EMPLOYEE table. An employee cannot be assigned to a department that is not defined in the DEPARTMENT table. A department can be defined in the DEPARTMENT table, yet have no employees assigned to it.

3.27.4 Foreign Key

Remember that every table in ORACLE has a primary key a field or fields making each record unique. In the employee table, the primary key is the employee ID number, and it is stored in the EMPL_ID field. In the DEPARTMENT table, the department number is the primary key and is stored in the DEPT_NO field.

The department number is also stored in a field in the EMPLOYEE table - the DEPT field. The department number field links the EMPLOYEE table to the DEPARTMENT table. This relationship is based on the department number field. Employee Flintstone works in department number 2, Gravel works in department 1.

When a field in one table matches the primary key of another table, the field is referred to as a foreign key. A foreign key is a field or a group of fields in one table whose values match those of the primary key of another table. You can think of a foreign key as the primary key of a foreign table. In the personnel database example, the DEPT field in the EMPLOYEE table is a foreign key. The DEPT_NO field is still the primary key of the DEPARTMENT table.

3.27.5 Lookup Table

When a foreign key exists in a table, the foreign key's table is sometimes referred to as a lookup table. The DEPARTMENT table in our example is a lookup table for the EMPLOYEE table. The value of an employee's department can be looked up in the DEPARTMENT table.

In Oracle, on-line screens or forms, have the ability to display a list of lookup table entries. The employee form could have a lookup list of departments from the DEPARTMENT table. The user simply highlights the preferred department, and Oracle automatically enters that department into the employee's record. This provides a means of ensuring that only valid data is entered in the database. A user can only select a listed department. Whether or not a lookup list is used in a form, Oracle will still check for a valid department when one is entered. However, if the form displays the list of possible departments, the user is less likely to enter an invalid department number.

Not only does Oracle allow you to link multiple tables, it also maintains consistency between them. It can prevent you from deleting a department which still has employees in it. Or if you change an employee's ID number, then all records of their work hours will also reflect that change. Ensuring that the data among related tables is correctly matched is referred to as maintaining referential integrity. This also applies to the previous example where a user enters an invalid department number for an employee. Oracle won't accept a department number that isn't in the DEPARTMENT table.

Oracle's relational databases represent a new and exciting database technology and philosophy on campus. As the Oracle development projects continue to impact on University applications, more and more users will realize the power and capabilities of relational database technology.

3.28 A Relational Database in Action

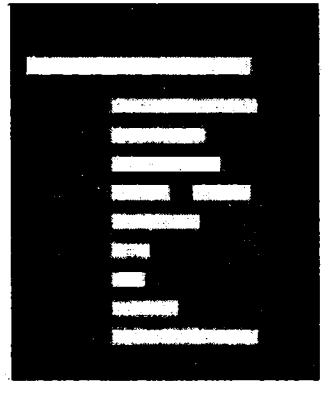
Whenever you or a customer needs information, chances are that information is coming from a database. Here's how such a database works, using the example of finding a book at a Web commerce site.

STEP 1:

You submit a Web form asking whether a particular book is in stock.

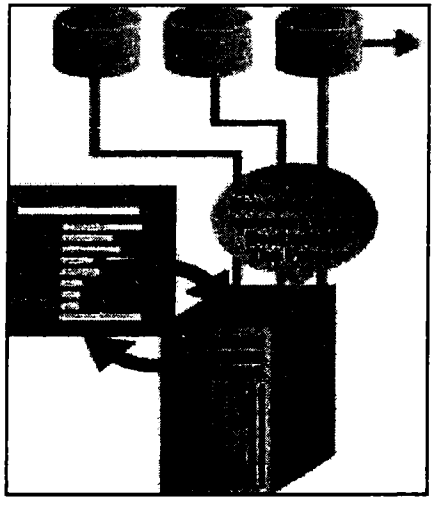
STEP 2:

The form request is sent across the internet to a Web server.



STEP 3:

A query program on the Web server extracts fields from the Web form to construct a database query, which it sends to the database.



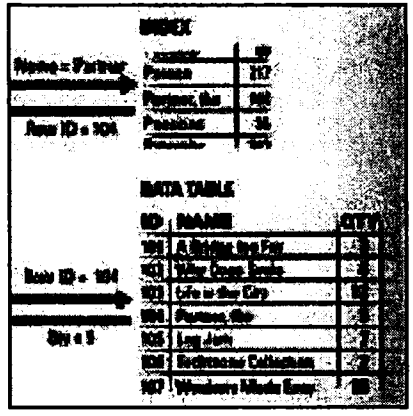
STEP 4:

The database network listener

1. Receives your query over the network; the query parser
2. Checks your query for correctness and breaks it into simple steps the database can run; the query optimizer
3. Modifies your query's steps so they happen in the fastest way possible; and the database worker thread
4. get an execution plan from the optimizer and begins work.

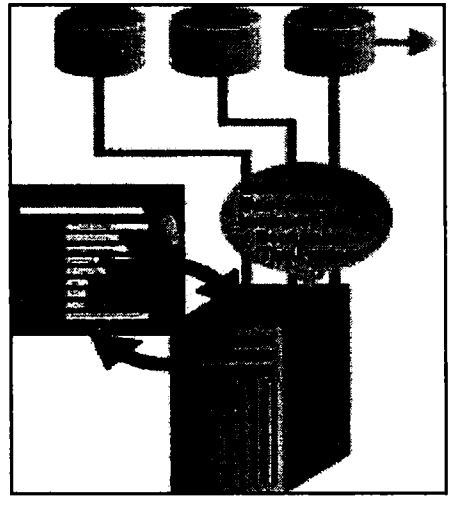
STEP 5:

The worker thread looks up the book's name in the index to find which row in the inventory table to retrieve; the worker thread looks up the correct book in the inventory table and gets the quantity, which is 5.



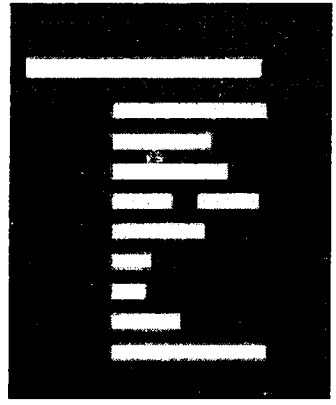
STEP 6:

The worker thread returns "5" to the listener, which returns that information to the program that submitted the query.



STEP 7:

The query program formats an HTML page containing available inventory and returns it to the Web server, which sends the page to you.



STEP 8:

You see real-time inventory information on your screen.

3.29 Domains

In his original definition of the relational database, Codd applied the term domain from relational mathematics to mean the column of a relational table.

In Codd's more recent work, he adopts the term attribute for the individual column of a relation.

Codd now clarifies the meaning of domain to be a rule or list establishing the set of values which may occur in any column derived from that domain:

- "All positive integers"
- "Geographic place names"
- "Gender as F or M"

3.30 Structure of Relational Database

A relational database consists of a collection of tables, each having a unique name. A row in a table represents a relationship among a set of values. Thus a table represents a collection of relationships.

There is a direct correspondence between the concept of a table and the mathematical concept of a relation. A substantial theory has been developed for relational databases.

3.30.1 Basic Structure

Fig. 3.2 shows the deposit and customer tables for our banking example.

bname	account#	ename	balance
Downtown	101	Johnaon	500
Lougheed_Mall	215	Smith	700
SFU	102	Hayea	100
SFU	301	Adams	1300

ename	street	ecity
Johnaon	Pankler	Vaneonver
Smith	North	Burnahy
Hayen	Omita	Burnahy
Adams	No.3 Road	Richmond
Jonea	Oak	Vaneonver

Fig. 3.2: The deposit and customer relations.

It has four attributes. For each attribute there is a permitted set of values, called the domain of that attribute e.g. the domain of bname is the set of all branch names.

Let D_1 denote the domain of bname, D_3 and D_4, and the remaining attributes' domains respectively. Then, any row of deposit consists of a four-tuple (v_1, v_2, v_3, v_4) where

$$v_1 \in D_1, v_2 \in D_2, v_3 \in D_3, v_4 \in D_4$$

In general, deposit contains a subset of the set of all possible rows.

That is, deposit is a subset of

$$D_4$$

In general, a table of n columns must be a subset of

$$\times_{i=1}^{n} D_i \quad \text{(all possible rows)}$$

Mathematicians define a relation to be a subset of a Cartesian product of a list of domains. You can see the correspondence with our tables. We will use the terms relation and tuple in place of table and row from now on.

Some more formalities:

- let the tuple variable t refer to a tuple of the relation r We say $t \in r$ to denote that the tuple t is in relation r.

Then r [bname] = r [1] = the value of r on the bname attribute.

So r [bname] = r [1] = "Downtown",

and [cname] = r [3] = "Johnson".

We will also require that the domains of all attributes be indivisible units. A domain is atomic if its elements are indivisible units. For example, the set of integers is an atomic domain. However, the set of all sets of integers is not. This because integers do not have subparts, but sets do - the integers comprising them. We could consider integers non-atomic if we thought of them as ordered lists of digits.

3.30.2 Database Scheme

We distinguish between a database scheme (logical design) and a database instance (data in the database at a point in time). A relation scheme is a list of attributes and their corresponding domains.

The text uses the following conventions:

- Italics for all names
- Lowercase names for relations and attributes
- Names beginning with an uppercase for relation schemes
- These notes will do the same.

For example, the relation scheme for the deposit relation:

Deposit-scheme = (bname, account#, cname, balance)

We may state that deposit is a relation on scheme Deposit-scheme by writing deposit(Deposit-scheme).

If we wish to specify domains, we can write:

(bname: string, account#: integer, cname: string, balance: integer).

Note that customers are identified by name. In the real world, this would not be allowed, as two or more customers might share the same name. Fig. 3.3 shows the E-R diagram for a banking enterprise.

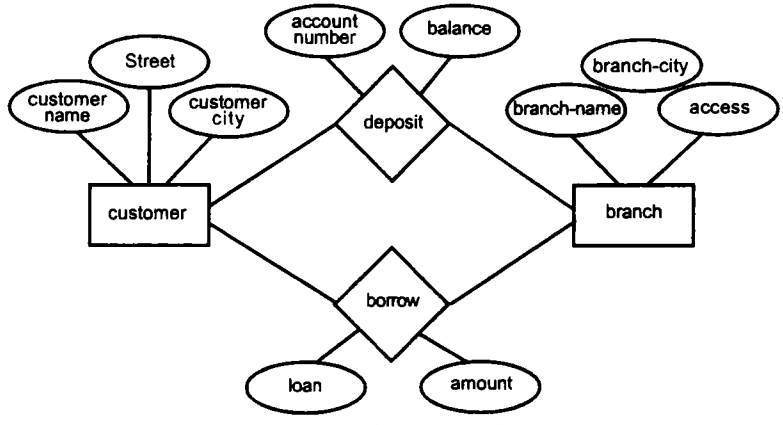


Fig. 3.3: E-R Diagram for the Banking Enterprise

The relation schemes for the banking example used throughout the text are:

- **Branch-scheme = (bname, assets, bcity)**
- **Customer-scheme = (cname, street, ccity)**
- **Deposit-scheme = (bname, account#, cname, balance)**
- **Borrow-scheme = (bname, loan#, cname, amount)**

Note: Some attributes appear in several relation schemes (e.g. bname, cname). This is legal, and provides a way of relating tuples of distinct relations.

Why not put all attributes in one relation?

Suppose we use one large relation instead of customer and deposit:

Account-scheme = (bname, account#, cname, balance, street, ccity)

If a customer has several accounts, we must duplicate her or his address for each account.

If a customer has an account but no current address, we cannot build a tuple, as we have no values for the address. We would have to use null values for these fields. Null values cause difficulties in the database. By using two separate relations, we can do this without using null values.

3.31 Query Languages

A query language is a language in which a user requests information from a database. These are typically higher-level than programming languages.

They may be one of:

- Procedural, where the user instructs the system to perform a sequence of operations on the database. This will compute the desired information.
- Nonprocedural, where the user specifies the information desired without giving a procedure for obtaining the information.
- A complete query language also contains facilities to insert and delete tuples as well as to modify parts of existing tuples.

3.32 The Relational Algebra

The relational algebra is an alternative and an equivalent to the relational calculus as the manipulative part of the relational model. There are eight operators in the relational algebra used to build relations and manipulate the data. They are:

1. Select
2. Project
3. Product
4. Union
5. Intersection
6. Difference
7. Join
8. Divide

The output of each of the above is another relation and thus it is possible to nest and combine operators. The five operations projection, product, union, difference, and selection, are all primitive. The other three can be defined in terms of these. There are several reasons for defining data manipulation as a relational algebra.

3.32.1 Relational Algebra Reasons

We define data manipulation as a relational algebra not just to support data retrieval but also to allow the writing of expressions (using the algebra) which in turn can be used for retrieval, for defining scope for updates, for defining virtual data which may form part of a users view, for defining access rights, integrity, etc.

Thus expressions are a high-level symbolic representation which can be subject to transformation rules to get equivalent expressions – this in essence means that relational algebraic expressions are a convenient basis for query optimisation.

The relational algebra is a procedural query language. It uses six fundamental operations:

- select (unary)
- project (unary)
- rename (unary)
- cartesian product (binary)
- union (binary)
- set-difference (binary)

Several other operations are defined in terms of the fundamental operations:

- set-intersection
- natural join
- division
- assignment

Operations produce a new relation as a result.

3.32.2 Formal Definition of Relational Algebra

A basic expression consists of either:

- A relation in the database.
- A constant relation.

General expressions are formed out of smaller subexpressions using

$\sigma_p(E_1)$

Fig. 3.4: Select (p a predicate)

$\Pi_s(E_1)$

Fig. 3.5: Project (S a list of Attributes)

$\rho_x(E_1)$

Fig. 3.6: Rename (x a Relation Name)

$E_1 \cup E_2$

Fig. 3.7: Union

$E_1 - E_2$

Fig. 3.8: Set Difference

$\rho_x(E_1)$

Fig. 3.9: Cartesian Product

Additional Operations

Additional operations are defined in terms of the fundamental operations. They do not add power to the algebra, but are useful to simplify common queries.

The Set Intersection Operation

Set intersection is denoted by g1.gif , and returns a relation that contains tuples that are in both of its argument relations. It does not add any power as :

$$r \cap s = r - (r - s)$$

To find all customers having both a loan and an account at the SFU branch, we write

$$\Pi_{cname}(\sigma_{bname="SFU"}(borrow)) \cap \Pi_{cname}(\sigma_{bname="SFU"}(deposit))$$

3.32.3 The Natural Join Operation

Often we want to simplify queries on a cartesian product.

For example, to find all customers having a loan at the bank and the cities in which they live, we need borrow and customer relations:

$$\Pi_{borrow.cname,ecity}(\sigma_{borrow.cname=customer.cname}(borrow \times customer))$$

Our selection predicate obtains only those tuples pertaining to only one cname.

This type of operation is very common, so we have the natural join, denoted by a sign g5.gif

Natural join combines a cartesian product and a selection into one operation. It performs a selection forcing equality on those attributes that appear in both relation schemes. Duplicates are removed as in all relation operations.

To illustrate, we can rewrite the previous query as :

$$\Pi_{cname}(\sigma_{bname="SFU"}(borrow)) \cap \Pi_{cname}(\sigma_{bname="SFU"}(deposit))$$

The resulting relation is shown in Fig.

ename	ecity
Smith	Burnaby
Hayes	Burnaby
Jones	Vanvuver

Joining borrow and customer relations.

We can now make a more formal definition of natural join.

Consider R and S to be sets of attributes. We denote attributes appearing in both relations by $R \cap S$. We denote attributes in either or both relations by $R \cup S$.

Consider two relations $r(R)$ and $s(S)$.

The natural join of r and s, denoted by $r \bowtie s$ is a relation on scheme $R \cup S$. It is a projection onto $R \cup S$ of a selection on $r \times s$ where the predicate requires $r.A = s.A$ for each attribute A in $R \cap S$.

Formally,

$$r \bowtie s = \Pi_{R \cup S}(\sigma_{r.A_1 = s.A_1 \wedge r.A_2 = s.A_2 \wedge \ldots \wedge r.A_n = s.A_n}(r \times s))$$

where

$$R \cap S = \{A_1, A_2, \ldots, A_n\}$$

To find the assets and names of all branches which have depositors living in Stamford, we need customer, deposit and branch relations:

$$A$$
$$\bowtie$$

Note that $\bowtie$ is associative.

To find all customers who have both an account and a loan at the SFU branch:

$$\Pi_{cname}(\sigma_{bname = "SFU"}(borrow \bowtie deposit))$$

This is equivalent to the set intersection version we wrote earlier. We see now that there can be several ways to write a query in the relational algebra.

If two relations $r(R)$ and $s(S)$ have no attributes in common, then $R \cap S = \emptyset$, and $r \bowtie s = r \times s$.

SELECT JOIN Example

A join on an SQL SELECT statement is, loosely speaking, a query in which data is retrieved from more than one table. The ability to join two or more tables is one of the most powerful features of relational systems. Many different types of join can be made. Examples are given of a simple equijoin, and a greater-than join.

3.32.4 Join

In relational algebra the join of relation A on attribute X with relation B on attribute Y yeilds the set of all tuples 't' such that 't' is a concatenation of a tuple 'a' belonging to A and tuple 'b' belonging to B and the predicate 'a.X comp b.Y' evaluates to true (attributes A.X and B.Y sould be defined on the same domain). If the 'comp' operator is '=' then it is an equi- join and thus it must include two identical attributes. If one of these is removed (using a projection) then the result is a natural join - this is the most important kind of join. JOINs are represented

in SQL by the SELECT statement. Joins are written in shorthand form e.g. A join B, (or A |><| B) and can be .regresentee diagramatically as follows:

Join Dependencies

Join dependencies arise where there is no lossless join decomposition into two relations, but there is a lossless join decomposition into more than two relations. Such dependencies are eliminated by decomposition into 5NF. Join dependencies do not have a simple real-world analogy.

3.32.5 The Division Operation

Division, denoted, is suited to queries that include the phrase "for all".

Suppose we want to find all the customers who have an account at all branches located in Brooklyn. Strategy: think of it as three steps. We can obtain the names of all branches located in Brooklyn by:

$$r_1 = \Pi_{bname}(\sigma_{bcity="Brooklyn"}(branch))$$

We can also find all cname, bname pairs for which the customer has an account. Now we need to find all customers who appear in r_2 with every branch name in $r_1 = \Pi_{bname}(\sigma_{bcity="Brooklyn"}(branch))$

The divide operation provides exactly those customers:

$$r_2 = \Pi_{cname,bname}(deposit)$$

which is simply $r_2 \div r_1$ Formally, Let $r(R)$ and $s(S)$ be relations.

Let $S \subseteq R$

The relation $r \div s$ is a relation on scheme $R - S$

A tuple t is in $r \div s$ if for every tuple t in s there is a tuple t_r in r satisfying both of the following:

$$t_r[S] = t_s[S] \qquad (3.2.1)$$

$$t_r[R - S] = t[R - S] \qquad (3.2.2)$$

These conditions say that the portion of a tuple is in if and only if there are tuples with the portion and the portion in for every value of the portion in relation . We will look at this explanation in class more closely.

The division operation can be defined in terms of the fundamental operations.

3.32.6 The Assignment Operation

Sometimes it is useful to be able to write a relational algebra expression in parts using a temporary relation variable (as we did with and in the division example).

The assignment operation, denoted, works like assignment in a programming language.

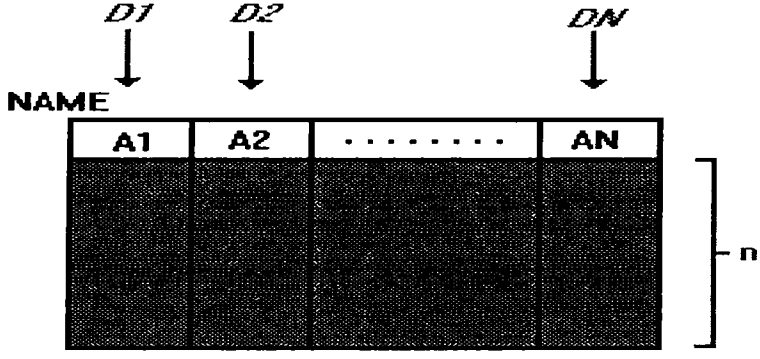
We could rewrite our division definition as:

- No extra relation is added to the database, but the relation variable created can be used in subsequent expressions. Assignment to a permanent relation would constitute a modification to the database.

3.33 Relational Tables Overview

A relational DBMS is a database where data is a collection of time-varying, normalised, independent relations of assorted degree and cardinalities. Each table or relation exists on domain D1 to DN and consists of :

- A set of attributes A1 to AN such that attribute Ai corresponds to domain Di
- A set of n-tuples or entries in the table.



where 'n' in this case is the degree of the relation. Primary keys also exist on tables.

The entire information content of the database is represented as explicit data values each of which is atomic. There are no links or pointers connecting tables and thus the representation of relationships is as data in another table.

3.34 RDBMS Criteria

A DBMS implementation is considered to be a relational DBMS if it supports users viewing the data as tables and nothing else, and at least the operators select, project, and join either explicitly as operators or as parts of other functions with no reference to physical access.

We can justify this subset of the full model because select, project and join are a useful subset of the full algebra and a system with tables and no operators does not provide the real flavour of relational systems. Also a system that provides the relational operators but requires physical predefinition of physical access paths to support them does not provide the physical data independence of a true relational system. Finally, a system merely executing exact operations requested by the user could not possibly provide query optimisation and would therefore be slow.

3.35 Query Optimisation Overview

Query optimisation is essential if a DBMS is to achieve acceptable performance and efficiency. Relational database systems based on the relational model and relational algebra have the strength that their relational expressions are at a sufficiently high level so query optimisation is feasible in the first place; in non-relational systems, user requests are low level and optimisation is done manually by the user - the system cannot help. Hence systems which implement optimisation have several advantages over systems that do not.

The optimisation process itself involves several stages, which involves the implementation of the relational operators. A different approach to query optimisation, called semantic optimisation has recently been suggested.

3.35.1 Internal Representation for Queries

The first stage in the optimisation process in a relational database is to convert the query to some internal representation that is more suitable for machine manipulation, thus eliminating purely external-level considerations (e.g. SQL syntax). The internal representation language must be rich enough to represent all possible queries in the system's query language. It should also be neutral as far as possible, in the sense that it should not prejudice any subsequence optimisation choices. The internal form that is typically chosen is some kind of abstract syntax tree or query tree.

It is convenient to assume that the internal representation employs one of the formalisms we are already familiar with - namely, the relational algebra or relational calculus which must be extended to be able to represent all SQL which is more powerful than it (e.g. aggregate functions etc.)

3.35.2 Stages of Query Optimisation

We can identify four broad stages in the overall query optimisation process, as follows:

- Cast the query into some internal representation
- Convert to canonical form
- Choose candidate low-level procedures
- Generate query plans and choose cheapest

3.35.3 Convert to Canonical Form

The second stage in the query optimisation process in relational databases is to convert the internal representation of the query into some equivalent canonical form i.e. converting a query into a version which is equivalent to all other versions, but with unnecessary parts omitted and which is more efficient then the original in some respect. This canonical form is obtained using a subset of known well-defined algebraic transformation rules.

3.35.4 Canonical Form Definition

The notion of canonical form into which a query in a relational database is converted, is central to many branches of mathematics and related disciplines. It can be defined as follows. Given a set of Q objects (say queries) and a notion of equivalence among those objects (say the notion

of q1 and q2 are equivalent if and only if they produce the same result), subset C of Q is said to be of canonical forms of Q (under the stated definition of equivalence) if and only if every object q in Q is equivalent to just one object c in C. The object c is said to be the canconical form for the object q. All 'interesting' properties that apply to an object q also apply to its canconical form c; thus it is sufficient to study just the small set C of canconical forms, not the large set Q, in order to prove a variety of 'interesting' results.

3.35.5 Convert to Canonical form

Algebraic Transformation Rules

The canonical form for a query during the query optimisation process is obtained using a subset of known well-defined relational algebraic transform rules. These include the following:

- Transform a sequence of restrictions.
- In a sequence of projections, all but the last can be ignored.
- Transform a restriction of a projection.
- Transformations based on the relational algebra.
- Commuting.
- Conjunctive normal form for predicate expressions.

3.35.6 Advantages of Optimisation

The major advantage of query optimisation is that users do not have to worry about how best to state their queries (i.e. how to phrase requests in order to get the best performance out of the system), and also there is a real possibility that the optimiser will do better than a human programmer in terms of using the DBMS efficiently. There are several reasons for this:

- The optimiser will have more information available than the user;
- Database statistics may change;
- The optimiser is a program; and
- The optimiser embodies the skills of the best programmers.

All of the above serve as evidence in support of the claim that optimisability i.e. the fact that relational database requests are optimisable, is in fact a strength of relational systems.

3.36 The Relational Model vs. the E-R Model

Why use an object model and a logical data model? Basically, models are a liaison between users and database developers. Remember, the primary purpose of modeling is to represent user data in a way that is meaningful. But how is meaning determined? In an abstract way, the object model provides meaning to business objects. In this phase, terms are defined, and perspectives are explored. The relational model more explicitly defines objects by applying business policies and restrictions and by evaluating data structures. For instance, in the student registration model, the prerequisites for a particular class can be defined in order to determine whether a student is eligible to register for that class (business policy). Accurate interpretations can result in useful applications and more intelligent database systems. For this reason, user perspective must be an ongoing consideration aduring the modeling phase of database design.

3.37 The Tuple Relational Calculus

1. The tuple relational calculus is a nonprocedural language. (The relational algebra was procedural.) We must provide a formal description of the information desired.
2. A query in the tuple relational calculus is expressed as:

$$\{t \mid P(t)\}$$

i.e. the set of tuples t_s for which predicate is true.

3. We also use the notation
   * $t[a]$ to indicate the value of tuple t_s on attribute a.
   * $t \in r$ to show that tuple t_s is in relation r.

Formal Definitions

1. A tuple relational calculus expression is of the form

$$\{t \mid P(t)\}$$

where P is a formula.

Several tuple variables may appear in a formula.

2. A tuple variable is said to be a free variable unless it is quantified by a $\exists$ or a $\forall$. Then it is said to be a **bound variable.**
3. A formula is built of atoms. An atom is one of the following forms:
   * $s \in r$, where s is a tuple variable, and r is a relation ($\notin$ is not allowed).
   * $s[x]\ \theta$ where s and u are tuple variables, and x and y are attributes, and θ is a comparison operator $(<, \leq, =, \neq, >, \geq)$.
   * $s[x]\ \theta\ u[y]$, where c is a constant in the domain of attribute x.
4. A Formulae are built up from atoms using the following rules:
   * An atom is a formula.
   * If P is a formula, then so are $\neg P$ and (P).
   * If P_1 and P_2 are formulae, then so are $P_1 \vee P_2$ and $P_1 \Rightarrow P_2$.
   * If $P(s)$ is a formula containing a free tuple variable, then

$\exists s \in r(P(s))$ and $\forall s \in r(P(s))$

are formulae also.

5. Note some equivalences:

$$P_1 \wedge P_2 = \neg(\neg P_1 \vee \neg P_2)$$

$$\forall t \in r(P(t)) = \neg \exists t \in r(\neg P(t))$$

$$P_1 \Rightarrow P_2 = \neg P_1 \vee P_2$$

3.37.1 Safety of Expressions

1. A tuple relational calculus expression may generate an infinite expression, e.g.

$$\{t \mid \neg(t \in borrow)\}$$

2. There are an infinite number of tuples that are not in borrow! Most of these tuples contain values that do not appear in the database.
3. Safe Tuple Expressions

We need to restrict the relational calculus a bit.

- The domain of a formula P, denoted dom(P), is the set of all values referenced in P.
- These include values mentioned in P as well as values that appear in a tuple of a relation mentioned in P. So, the domain of P is the set of all values explicitly appearing in P or that appear in relations mentioned in P.
- $dom(t \in borrow \wedge t[amount] < 1200)$ is the set of all values appearing in borrow.
- $dom(\{t \mid \neg(t \in borrow)\})$ is the set of all values appearing in borrow.

We may say an expression $\{t \mid P(t)\}$ is safe if all values that appear in the result are values from dom(P).

4. A safe expression yields a finite number of tuples as its result.

Otherwise, it is called unsafe.

3.37.2 Expressive Power of Languages

1. The tuple relational calculus restricted to safe expressions is equivalent in expressive power to the relational algebra.

The Domain Relational Calculus

1. Domain variables take on values from an attribute's domain, rather than values for an entire tuple.

Expressive Power of Languages

1. All three of the following are equivalent:
   * The relational algebra.
   * The tuple relational calculus restricted to safe expressions.
   * The domain relational calculus restricted to safe expressions.

3.37.3 Modifying the Database

1. Up until now, we have looked at extracting information from the database. We also need to add, remove and change information. Modifications are expressed using the assignment operator.

Deletion

1. Deletion is expressed in much the same way as a query. Instead of displaying, the selected tuples are removed from the database. We can only delete whole tuples.

In relational algebra, a deletion is of the form

$$r \leftarrow r - E$$

where r is a relation and E is a relational algebra query.

Tuples r in for which E is true are deleted.

2. Some examples:
   1. Delete all of Smith's account records.

$$deposit \leftarrow deposit - \sigma_{cname = \text{"}Smith\text{"}}(deposit)$$

   2. Delete all loans with loan numbers between 1300 and 1500.

$$deposit \leftarrow deposit - \sigma_{loan\# \geq 1300 \wedge loan\# \leq 1500}(deposit)$$

   3. Delete all accounts at Branches located in Needham.

$$r \leftarrow r - E$$

Insertions

1. To insert data into a relation, we either specify a tuple, or write a query whose result is the set of tuples to be inserted. Attribute values for inserted tuples must be members of the attribute's domain.
2. An insertion is expressed by

$$r \leftarrow r \cup E$$

where r is a relation and E is a relational algebra expression.

3. Some examples:
   1. To insert a tuple for Smith who has $1200 in account 9372 at the SFU branch.

$$deposit \leftarrow deposit \cup \{(\text{"}SFU\text{"}, 9372, \text{"}Smith\text{"}, 1200)\}$$

   2. To provide all loan customers in the SFU branch with a $200 savings account.

$$r_1 \leftarrow (\sigma_{bname = \text{"}SFU\text{"}}(borrow))$$
$$r_2 \leftarrow \Pi_{bname, loan\#, cname}(r_1)$$
$$deposit \leftarrow deposit \cup (r_2 \times \{(200)\})$$

1. Updating

Updating allows us to change some values in a tuple without necessarily changing all. We use the update operator, δ, with the form

$$\delta_{A \leftarrow E}(r)$$

where r is a relation with attribute A, which is assigned the value of expression E.

The expression E is any arithmetic expression involving constants and attributes in relation r.
Some examples:

1. To increase all balances by 5 percent.

$$\delta_{balance \leftarrow balance*1.05}(deposit)$$

This statement is applied to every tuple in deposit.

2. To make two different rates of interest payment, depending on the balance amount:

$$\delta_{balance \leftarrow balance*1.05}(\sigma_{balance > 10000}(deposit))$$

$$\delta_{balance \leftarrow balance*1.05}(\sigma_{balance \leq 10000}(deposit))$$

Note: In this example, the order of the two operations is important. (Why?)

Views

1. We have assumed up to now that the relations we are given are the actual relations stored in the database.
2. For security and convenience reasons, we may wish to create a personalized collection of relations for a user.
3. We use the term view to refer to any relation, not part of the conceptual model, that is made visible to the user as a "virtual relation".
4. As relations may be modified by deletions, insertions and updates, it is generally not possible to store views. This is because views must then be recomputed for each query referring to them.

3.38 View Definition

1. A view is defined using the create view command:

create view v as <query expression>

where <query expression> is any legal query expression.

The view created is given the name .

2. To create a view all-customer of all branches and their customers:

create view *all-customer* **as**

$$\Pi_{bname,cname}(deposit) \cup \Pi_{bname,cname}(borrow)$$

3. Having defined a view, we can now use it to refer to the virtual relation it creates. View names can appear anywhere a relation name can.
4. We can now find all customers of the SFU branch by writing

$$\Pi_{cname}(\sigma_{bname="SFU"}(all - customer))$$

3.38.1 Updates Through Views and Null Values

1. Updates, insertions and deletions using views can cause problems. The modifications on a view must be transformed to modifications of the actual relations in the conceptual model of the database.

2. An example will illustrate: consider a clerk who needs to see all information in the borrow relation except amount.

Let the view loan-info be given to the clerk:

$$\textbf{create view } \textit{loan-info} \textbf{ as}$$
$$\Pi_{bname,loan\#,cname}(borrow)$$

3. Since SQL allows a view name to appear anywhere a relation name may appear, the clerk can write:

$$\textit{loan-info} \leftarrow \textit{loan-info} \cup \{(\text{"SFU"}, 3, \text{"Ruth"})\}$$

This insertion is represented by an insertion into the actual relation borrow, from which the view is constructed.

However, we have no value for amount. A suitable response would be:

- Reject the insertion and inform the user.
- Insert ("SFU",3,"Ruth",null) into the relation.

The symbol null represents a null or place-holder value. It says the value is unknown or does not exist.

4. Another problem with modification through views: consider the view

$$\textbf{create view } \textit{branch-city} \textbf{ as}$$
$$\Pi_{bname,city}(borrow \bowtie customer)$$

This view lists the cities in which the borrowers of each branch live.

Now consider the insertion

$$\textit{branch-city} \leftarrow \textit{branch-city} \cup \{(\text{"Brighton"}, \text{"Woodside"})\}$$

Using nulls is the only possible way to do this.

If we do this insertion with nulls, now consider the expression the view actually corresponds to:

RELATION = STU-ACT

StuID	Activity	Fee
100	Diving	200
150	Softball	50
175	Racquetball	50
200	Softball	50

As comparisons involving nulls are always false, this query misses the inserted tuple.

To understand why, think about the tuples that got inserted into borrow and customer. Then think about how the view is recomputed for the above query.

3.39 Extension of the Codd Rules and Features

In 1993, E.F. Codd & Associates published a white paper, commissioned by Arbor Software (now Hyperion Solutions), entitled 'Providing OLAP (On-line Analytical Processing) to User-Analysts: An IT Mandate'. Dr Codd was, of course, very well known as a respected database researcher from the 1960s through to the late 1980s and is credited with being the inventor of the relational database model. Unfortunately, his OLAP rules proved to be controversial due to being vendor-sponsored, rather than mathematically based. It is also unclear how much involvement Dr Codd himself had with this activity, but it seems likely that his role was very limited. Several of the rules seem to have been invented by the sponsoring vendor, not Dr Codd. The white paper should therefore be regarded as a vendor-published brochure (which it is) rather than as an academic paper (which it is not). Note that this paper was not published by Codd and Date, and Chris Date has never endorsed Codd's work in this area.

The OLAP white paper included 12 rules, which are now well known (and available for download from vendors' Web sites). They were followed by another six (much less well known) rules in 1995 and Dr Codd also restructured the rules into four groups, calling them 'features'. The features are briefly described and evaluated here, but they are now rarely quoted and little used.

Basic Features B

F1: Multi-dimensional Conceptual View (Original Rule 1). Few would argue with this feature; like Dr Codd, we believe this to be the central core of OLAP. Dr Codd included 'slice and dice' as part of this requirement.

F2: Intuitive Data Manipulation (Original Rule 10). Dr Codd preferred data manipulation to be done through direct actions on cells in the view, without recourse to menus or multiple actions. One assumes that this is by using a mouse (or equivalent), but Dr Codd did not actually say so. Many products fail on this, because they do not necessarily support double clicking or drag and drop. The vendors, of course, all claim otherwise. In our view, this feature adds little value to the evaluation process. We think that products should offer a choice of modes (at all times), because not all users like the same approach.

F3: Accessibility: OLAP as a Mediator (Original Rule 3). In this rule, Dr Codd essentially described OLAP engines as middleware, sitting between heterogeneous data sources and an OLAP front-end. Most products can achieve this, but often with more data staging and batching than vendors like to admit.

F4: Batch Extraction Vs Interpretive (New). This rule effectively required that products offer both their own staging database for OLAP data as well as offering live access to external data. We agree with Dr Codd on this feature and are disappointed that only a minority of OLAP products properly comply with it, and even those products do not often make it easy or automatic. In effect, Dr Codd was endorsing multidimensional data staging plus partial pre-calculation of large multidimensional databases, with transparent reach-through to underlying detail. Today, this would be regarded as the definition of a hybrid OLAP, which is indeed becoming a popular architecture, so Dr Codd has proved to be very perceptive in this area.

F5: OLAP Analysis Models (New). Dr Codd required that OLAP products should support all four analysis models that he described in his white paper (Categorical, Exegetical, Contemplative and Formulaic). We hesitate to simplify Dr Codd's erudite phraseology, but we would describe these as parameterized static reporting, slicing and dicing with drill down, 'what if?' analysis and goal seeking models, respectively. All OLAP tools in this Report support the first two (but some other claimants do not fully support the second), most support the third to some degree (but probably less than Dr Codd would have liked) and few support the fourth to any usable extent. Perhaps Dr Codd was anticipating data mining in this rule?

F6: Client Server Architecture (Original Rule 5). Dr Codd required not only that the product should be client/server but that the server component of an OLAP product should be sufficiently intelligent that various clients could be attached with minimum effort and programming for integration. This is a much tougher test than simple client/server, and relatively few products qualify. We would argue that this test is probably tougher than it needs to be, and we prefer not to dictate architectures. However, if you do agree with the feature, then you should be aware that most vendors who claim compliance, do so wrongly. In effect, this is also an indirect requirement for openness on the desktop. Perhaps Dr Codd, without ever using the term, was thinking of what the Web would deliver? Or perhaps he was anticipating a widely accepted API standard, which OLE DB for OLAP is becoming.

F7: Transparency (Original Rule 2). This test was also a tough but valid one. Full compliance means that a user of, say, a spreadsheet should be able to get full value from an OLAP engine and not even be aware of where the data ultimately comes from. To do this, products must allow live access to heterogeneous data sources from a full function spreadsheet add-in, with the OLAP server engine in between. Although all vendors claimed compliance, many did so by outrageously rewriting Dr Codd's words. Even Dr Codd's own vendor-sponsored analyses of Essbase and (then) TM/1 ignore part of the test. In fact, there are a few products that do fully comply with the test, including Analysis Services, Express, and Holos, but neither Essbase nor iTM1 (because they do not support live, transparent access to external data), in spite of Dr Codd's apparent endorsement. Most products fail to give either full spreadsheet access or live access to heterogeneous data sources. Like the previous feature, this is a tough test for openness.

F8: Multi-User Support (Original Rule 8). Dr Codd recognised that OLAP applications were not all read-only and said that, to be regarded as strategic, OLAP tools must provide concurrent access (retrieval and update), integrity and security. We agree with Dr Codd, but also note that many OLAP applications are still read- only. Again, all the vendors claim compliance but, on a strict interpretation of Dr Codd's words, few are justified in so doing.

Special Features S

F9: Treatment of Non-Normalized Data (New). This refers to the integration between an OLAP engine and denormalized source data. Dr Codd pointed out that any data updates performed in the OLAP environment should not be allowed to alter stored denormalized data in feeder systems. He could also be interpreted as saying that data changes should not be allowed in what are normally regarded as calculated cells within the OLAP database. For example, Essbase allows this, and Dr Codd would perhaps disapprove.

F10: Storing OLAP Results: Keeping Them Separate from Source Data (New). This is really an implementation rather than a product issue, but few would disagree with it. In effect, Dr Codd was endorsing the widely held view that read-write OLAP applications should not be implemented directly on live transaction data, and OLAP data changes should be kept distinct from transaction data. The method of data write-back used in Microsoft Analysis Services is the best implementation of this, as it allows the effects of data changes even within the OLAP environment to be kept segregated from the base data.

F11: Extraction of Missing Values (New). All missing values are cast in the uniform representation defined by the Relational Model Version 2. We interpret this to mean that missing values are to be distinguished from zero values. In fact, in the interests of storing sparse data more compactly, a few OLAP tools such as iTM1 do break this rule, without great loss of function.

F12: Treatment of Missing Values (New). All missing values to be ignored by the OLAP analyser regardless of their source. This relates to Feature 11, and is probably an almost inevitable consequence of how multidimensional engines treat all data.

Reporting Features R

F13: Flexible Reporting (Original Rule 11). Dr Codd required that the dimensions can be laid out in any way that the user requires in reports. We would agree, and most products are capable of this in their formal report writers. Dr Codd did not explicitly state whether he expected the same flexibility in the interactive viewers, perhaps because he was not aware of the distinction between the two. We prefer that it is available, but note that relatively fewer viewers are capable of it. This is one of the reasons that we prefer that analysis and reporting facilities be combined in one module.

F14: Uniform Reporting Performance (Original Rule 4). Dr Codd required that reporting performance be not significantly degraded by increasing the number of dimensions or database size. Curiously, nowhere did he mention that the performance must be fast, merely that it be consistent. In fact, our experience suggests that merely increasing the number of dimensions or database size does not affect performance significantly in fully pre-calculated databases, so Dr Codd could be interpreted as endorsing this approach— which may not be a surprise given that Arbor Software sponsored the paper. However, reports with more content or more on-the-fly calculations usually take longer (in the good products, performance is almost linearly dependent on the number of cells used to produce the report, which may be more than appear in the finished report) and some dimensional layouts will be slower than others, because more disk blocks will have to be read. There are differences between products, but the principal factor that affects performance is the degree to which the calculations are performed in advance and where live calculations are done (client, multidimensional server engine or RDBMS). This is far more important than database size, number of dimensions or report complexity.

F15: Automatic Adjustment of Physical Level (Supersedes Original Rule 7). Dr Codd required that the OLAP system adjust its physical schema automatically to adapt to the type of model, data volumes and sparsity. We agree with him, but are disappointed that most vendors fall far short of this noble ideal. We would like to see more progress in this area and also in the related area of determining the degree to which models should be pre- calculated (a major issue that

Dr Codd ignores). The Panorama technology, acquired by Microsoft in October 1996, broke new ground here, and users can now benefit from it in Microsoft Analysis Services.

Dimension Control D

F16: Generic Dimensionality (Original Rule 6). Dr Codd took the purist view that each dimension must be equivalent in both its structure and operational capabilities. This may not be unconnected with the fact that this is an Essbase characteristic. However, he did allow additional operational capabilities to be granted to selected dimensions (presumably including time), but he insisted that such additional functions should be grantable to any dimension. He did not want the basic data structures, formulae or reporting formats to be biased towards any one dimension. This has proven to be one of the most controversial of all the original 12 rules. Technology focused products tend to largely comply with it, so the vendors of such products support it. Application focused products usually make no effort to comply, and their vendors bitterly attack the rule. With a strictly purist interpretation, few products fully comply. We would suggest that if you are purchasing a tool for general purpose, multiple application use, then you want to consider this rule, but even then with a lower priority. If you are buying a product for a specific application, you may safely ignore the rule.

F17: Unlimited Dimensions & Aggregation Levels (Original Rule 12). Technically, no product can possibly comply with this feature, because there is no such thing as an unlimited entity on a limited computer. In any case, few applications need more than about eight or ten dimensions, and few hierarchies have more than about six consolidation levels. Dr Codd suggested that if a maximum must be accepted, it should be at least 15 and preferably 20; we believe that this is too arbitrary and takes no account of usage. You should ensure that any product you buy has limits that are greater than you need, but there are many other limiting factors in OLAP products that are liable to trouble you more than this one. In practice, therefore, you can probably ignore this requirement.

F18: Unrestricted Cross-dimensional Operations (Original Rule 9). Dr Codd asserted, and we agree, that all forms of calculation must be allowed across all dimensions, not just the 'measures' dimension. In fact, many products which use only relational storage are weak in this area. Most products, such as Essbase, with a multidimensional database are strong. These types of calculations are important if you are doing complex calculations, not just cross tabulations, and are particularly relevant in applications that analyse profitability.

3.40 Past and Future of Relational Databases

Today's database market is dominated by products based on the Relational Model of data, which is now over thirty years old. The challenge of the Object Model of data has faded, and the economic case for adopting the hybrid Object/ Relational technology is unproven.

So is the Relational Model the last word in database architecture? Lazy Software believes not, and offers a powerful alternative in the Associative Model of Data, which is implemented by SentencesO, its database management system for the Internet and beyond.

Since the Relational Model was conceived, PCs and the Internet have extended the role of computers into new areas that depend on the PC's multimedia capability. The 1980s saw the arrival of a new generation of databases based on the Object Model of data, that were better able to manage complex multimedia data structures such as image and sound.

Analysts forecast that object databases would rapidly replace relational databases, but their sales have fallen far short of predictions. In fact, the Object Model was never intended to compete with the Relational Model. As a result, in many respects it is inferior to the Relational Model for transaction processing. Moreover, the market has not bought into the need for specialised technology solely to store multimedia files.

The most visible limitation of the Relational Model has been its inability to handle complex data, but importance of this has been exaggerated. The Relational Model has some far more significant limitations that the market has not yet challenged:

- If your programs are so expensive, why can they do only one job?
- If your applications are so vital, why can't they work like you do?
- If your customers are so important, why store the same data for each one?
- If your databases are so critical, why can't they work together?

Review Question for Unit 3

Objective type questions:

1. The relational model's structure is defined by:
   (a) Relations, tuples, and attributes.
   (b) Files, records, and fields.
   (c) Tables, rows, and columns.
   (d) All of the above.
   (e) None of the above.
2. The relational model is important because:
   (a) It can be used to express DBMS-independent designs, since its constructs are broad and general.
   (b) It is the basis for an important category of DBMS products.
   (c) Both A and B.
   (d) None of the above.
3. The expression X Y is read as:
   (a) "X functionally determines Y."
   (b) "X determines Y."
   (c) "Y is dependent on X."
   (d) All of the above.
   (e) None of the above.

4. A modification anomaly occurs when:
   (a) Undesirable consequences result from changing data in a table.
   (b) Functional dependencies must be modified.
   (c) Relations are normalized.
   (d) All of the above.
   (e) None of the above.
5. Normal forms are nested, which means that:
   (a) A relation in 1NF is also in 2NF and 3NF.
   (b) A relation in Boyce-Codd NF is also in 4NF and 5NF.
   (c) A relation in 3NF is also in 2NF and 1NF.
   (d) None of the above.
6. Normalisation:
   (a) Is used because not all relations are equal.
   (b) Is a process for converting a relation that has certain problems to two or more relations that do not have these problems.
   (c) Can be used as a guideline for checking the desirability and correctness of relations.
   (d) All of the above.
   (e) None of the above.

CHAPTER 4

STANDARD QUERY LANGUAGE

4.1 History of SQL

When IBM began implementing Dr. E. F. Codd's relational database model in the early 1970's, Donald Chamberlin and others at IBM's research division developed a prototype language called Structured English Query Language, or SEQUEL for short. Later, IBM expanded and revised the language, dubbing it SEQUEL/2. SEQUEL/2 became the application programming interface (API) for IBM's first relational database system prototype, System/R. For legal reasons, IBM eventually changed the name SEQUEL/2 to SQL. That is why many people and reference books today, like this magazine, use the pronunciation "sequel" and spell it out as Structured Query Language.

SQL allows users to access data in relational database management systems, such as Oracle, Sybase, Informix, Microsoft SQL Server, Access, and others, by allowing users to describe the data the user wishes to see. SQL also allows users to define the data in a database, and manipulate that data. This page will describe how to use SQL, and give examples.

The relational model was first introduced in a paper published in 1970 by Dr. E. F. Codd. It provided a mathematical basis for structuring, manipulating, and controlling data, and it abstracted data from any physical implementation. Prior to that time, database users had to know how records were physically linked in order to access them. When the physical structure of a database was changed (for performance reasons, for example), programmers had to rewrite their applications, even if the logical structure remained the same. In the relational model, data is separated from application programs. In fact, the model said nothing whatsoever about how data should be stored or accessed. It dealt exclusively with how data was seen from the user's point of view. The relational model significantly changed the way people thought about managing data. It transformed database technology from an art into a science and revolutionized the industry.

4.2 Relational Database Management Systems

The sole purpose of a DBMS is to manage data. If we measure quantity of DBMSs by the number of installations, then by far the most common kind of DBMS is a relational DBMS. Some commercial relational DBMSs are Oracle, Informix and Microsoft Access. Postgres is a research prototype DBMS which supports relational and extended relational data models.

A relational DBMS manages data which are modeled as tables, such as the two tables illustrated below.

The Employees table below describes the employees of a company, listing for each employee their ID within the company, their name, and the ID of the department where they work. The Departments table describes the company's departments by listing each department's ID, the department's name, and the building where the department is located. We have included a small amount of sample data.

- Employees

EmpID	Name	DeptID
123	Sergio	CS
234	Lara	Comm
135	Elayne	Comm
124	Dick	CS

- Departments

DeptID	Name	Bldg
Comm	Communications	Lincoln
CS	Computer Science	PCAT

Typically, the DBMS stores its data on disk. The user can write an SQL statement and submit it to a DBMS, which then will retrieve the appropriate data from disk and return it to the user. This interaction is depicted in the Fig. 4.1 below.

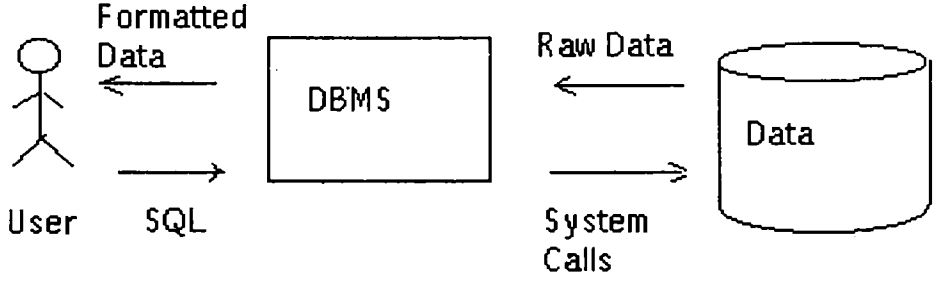


Fig. 4.1

SQL has many more features than data retrieval, but we will cover only retrieval here.

4.3 SQL: The Universal Database Language

SQL (pronounced "ess-que-el") stands for Structured Query Language. SQL is used to communicate with a database. According to ANSI (American National Standards Institute), it is the standard language for relational database management systems. SQL statements are used to perform tasks such as update data on a database, or retrieve data from a database. Some common relational database management systems that use SQL are: Oracle, Sybase, Microsoft SQL Server, Access, Ingres, etc. Although most database systems use SQL, most of them also have their own additional proprietary extensions that are usually only used on their system. However, the standard SQL commands such as "Select", "Insert", "Update", "Delete", "Create", and "Drop" can be used to accomplish almost everything that one needs to do with a database. This tutorial will provide you with the instruction on the basics of each of these commands as well as allow you to put them to practice using the SQL Interpreter.

As we said, SQL (Structured Query language) is the language of choice for most modern multi-user, relational databases. That is because SQL provides the syntax and idioms (language) you need to talk to (query) relational databases in a standardized, cross- platform/product way (structured).

The beauty of SQL is that it idiomizes the relational model. Rather than refer to data as a set of pointers, SQL provides predefined procedures to allow you to use any value in a table to relate other tables in a database. So long as a database is structured using the relational model, SQL will be a natural fit because SQL was designed to make sense in a relational system. SQL by its very design is a language that can be used to talk about relating tables.

SQL databases (most modern relational databases), as you will recall, are composed of a set of row/column-based "tables", indexed by a "data dictionary". To access data in the tables, you simply use SQL to navigate the system and produce "views" based on search criteria defined in the SQL query.

Let us now step back for a moment and look at each of these terms.

Relational database system contains one or more objects called tables. The data or information for the database are stored in these tables. Tables are uniquely identified by their names and are comprised of columns and rows. Columns contain the column name, data type, and any other attributes for the column. Rows contain the records or data for the columns. Here is a sample able called "weather". city, state, high, and low are the columns. The rows contain the data for this table:

Weather

City	State	High	Low
Bhopal	M.P.	42	7.2
NewDelhi	Delhi	45	5.0
Mumbai	Maharastra	44	14.4
Kolkata	W. Bengal	46	8.9
Chennai	Tamilnadu	40	18.3

SQL is often cited as being the lingua franca of relational database management systems. Certainly no other database language has found such wide acceptance among such a broad range of products. Since it was first standardized in 1986, SQL has become universally adopted. Even nonrelational database systems support a SQL interface. But unlike other computer languages such as C or COBOL, which are the exclusive domain of programmers, SQL is employed by a variety of professionals. Programmers, database administrators, and business analysts alike use SQL to access information. A working knowledge of the language is valuable to anyone who uses a database.

SQL is a special-purpose, nonprocedural language that supports the definition, manipulation, and control of data in relational database management systems. It is a special-purpose language, because you can use it only for handling databases; you can't write general-purpose applications with it. (To write an application, you have to embed SQL in some other language, and it is frequently used that way.) That is why SQL is also known as a data sublanguage. A sublanguage can be used with application languages, but it is not a full-fledged application language. Also, a full-featured application language usually includes semantics for procedures, whereas SQL is nonprocedural. It doesn't specify how something should be done, it just specifies what should be done. In other words, SQL is concerned with results rather than procedures.

By far the most important feature of SQL is that it provides access to relational databases. That is so fundamental to SQL that many people think the terms SQL database and relational database are synonymous. But as you will soon see, they're not. In fact, the SQL-92 standard doesn't even mention the term relation.

A Simple SQL Statement

Here is an SQL statement which retrieves names of employees in the department whose ID is CS:

```
SELECT Employees.Name
FROM Employees
WHERE Employees.DeptID = "CS"
```

We will begin by trying to understand this SQL statement. To understand any SQL statement, it is best to begin with the FROM clause.

```
FROM Employees
```

means that the data retrieved will originate from the Employees table.

Next, consider the WHERE clause.

```
WHERE Employees.DeptID = "CS"
```

means that the data retrieved will come from rows whose DeptID column has the value CS. Thus the data retrieved by this SQL statement will be derived from the rows.

```
EmpID Name Dept
123 Sergio CS
124 Dick CS
```

Finally, the SELECT clause specifies that data retrieved by this SQL statement is from the Name column. Thus this SQL query will retrieve the table.

```
Name
Sergio
Dick
```

Some observations:

1. The terms Employees.Name and Employees.DeptID use Employees as a qualifier. These qualifiers are necessary since the column names DeptID and Name appear in both tables. However, it is good programming practice to use qualifiers even if they are not necessary.
2. SQL uses = for the equality relation, not ==. Other possible relations are >, <, <> (not equal), >= and <= .
3. The meanings of terms in SQL clash with their meanings as logical operators. This table should highlight the problems and help you avoid confusion.

Meaning in SQL Meaning as a Logical Operator

SELECT Retrieve certain columns s : Retrieve rows satisfying a condition

PROJECT NONE p : Retrieve certain columns WHERE Retrieve rows satisfying a condition None

4. The official SQL language is defined by an international standards committee. DBMS vendors typically advertise that they conform to some or all of the current standard.
5. The SQL standard does not specify case sensitivity, so it is up to vendors. Most vendors provide case insensitive SQL. Capitalization of keywords is just a user friendly artifact.
6. The order of SELECT, FROM, WHERE clauses is specified by the standard.
7. More complex WHERE conditions can be used in SQL. Any boolean combination of conditions can be used.

4.4 SQL: Three Types of Statements

Now we are ready to begin writing some SQL code. If you have access to a SQL database and would like to try the examples as we go along, make sure to log on as a user with full privileges and to use an interactive SQL facility. (If your SQL database is on a network, you will have to talk to your database administrator about getting privileges.) If you don't have access to a SQL database, don't worry: the examples are easy to follow. You don't need to try them out to understand what's going on.

The only way to make anything happen in SQL is to execute a SQL statement. There are several types of SQL statements, but they generally fall into three categories:

- Data definition language (DDL) statements
- Data manipulation language (DML) statements
- Data control statements

In a way, SQL is like three languages rolled into one.

4.5 SQL Tables

We have already discussed the concept of tables in the last part, but let's just refresh our memory in terms of how tables relate to SQL. A table is a systematic way to store data. For the most part, a table is just like a spreadsheet. Tables are composed of rows (records) and each row is composed of columns (fields).

Employee Table

EmployeeID Number	Employee Name	Employe Phone	Salary
001	Sharmila	3316697	90,000
002	Sushmita	3736789	40,000
003	Sangita	6789011	50,000

How the tables are stored by the database you are using does not really make a difference for you. The beauty of SQL is that it works independently of the internal structure of the database. The tables could be stored as simple flat files on a local PC or as complex, networked, compressed, encrypted and proprietary data structures. All you need to know is the table's name. If you know the name, you can use SQL to call up the table. We will look at manipulating tables in detail a bit later. But first, let's look at the data dictionary.

A Sample Database

Okay, let is define a simple relational database that we can use to practice with...

We will define a database called "MY_COMPANY" with four tables, "CLIENTS", "EMPLOYEES", "PRODUCTS", and "SALES". These tables will look something like the following:

EMPLOYEES Table

EMP_NUM	EMP_NAME	EMP_COMMISSION	EMP_SALARY
001	Lim Li Chuen	10%	90,000 USD
002	Lim Sing Yuen	20%	40,000 USD
003	Loo Soon Keat	20%	50,000 USD

CLIENTS Table

C_NUM	C_NAME	C_ADDR	C_CITY	C_STATE	C_ZIP	C_PHONE
001	Jason Lim	100 W	10th St	LA	CA 90027	456-7890
002	Rick Tan	21 Jack	St LA	CA	90031	649-2038
003	Stephen	Petersen 1029#	A Kent Ave.	LA	CA 90102	167-3333

PRODUCTS Table

P_NUM	P_QUANTITY	P_PRICE
001	104	99.99
002	12	865.99
003	2000	50.00

SALES Table

S_NUM	P_NUM	S_QUANITY	S_AMOUNT	E_NUM	C_NUM
001	001	1	99.99	101	102
002	001	2	199.98	102	101
003	002	1	865.99	101	103

4.6 Creating Databases

Okay, as we have said before, it would be too difficult for us to cover how to install and conFig.ure all of the myriad of relational databases around, so it is your job to get something installed on your local system or to arrange with your systems administrator to give you access to an existing database system.

Specifically, to get a database working, you will 1) install the application on the host computer (insert disk A, click install.exe), 2) create a database according to the instructions of the database application you install, 3) populate your database with tables, and 4) populate your tables with data.

In the case of Access and many other database systems around these days, the process of creating databases and tables is pretty much handled by Wizards and GUI tools. Thus, you rarely need to use SQL for such operations. More likely, you would choose something like "FILE|NEW DATABASE" from the main menu.

Below is an example of the Wizard used by Microsoft Access. In this case, we have chosen "File|New Database" from the main menu at the top of the application window and then we double click "Blank Database". Of course, although Access and other database systems offer template databases for your convenience, we will create our own for practice.

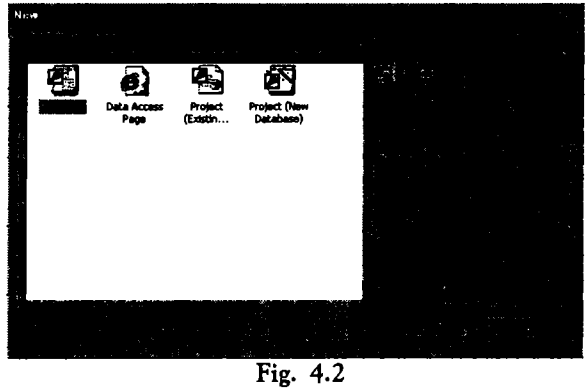


Fig. 4.2

In Access, as will be the case in most databases, once you create a database, you will then be asked to define the database structure.

Specifically, you will be asked to define some tables in your database (and perhaps other more advanced tools like Macros, Indexes, Views, Forms, Queries, etc). Below is a screen-shot from Access that offers a series of database definition tabs.

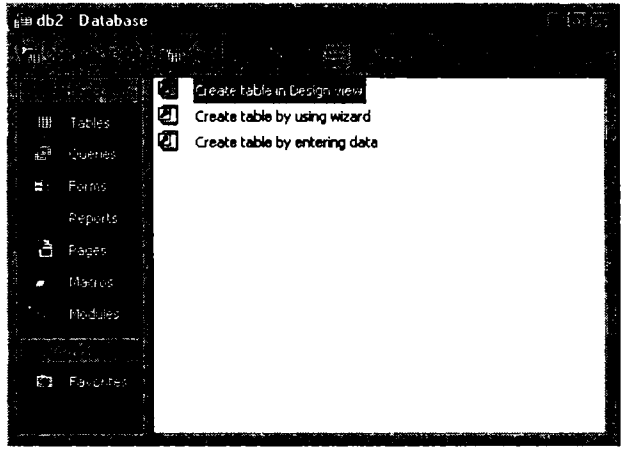


Fig. 4.3

To create a table, simply choose "New" from the "Tables" tab and follow instructions for defining your fields. Nothing could be simpler. Hopefully, now that you understand what the database is doing in the background, you will be easily able to understand what you need to do to get it working.

However, even though GUIs are pretty swank these days, it is probably a good idea to learn the SQL to which is being used in the background to create the database. Specifically, you use the CREATE command to create a database such as in the following.

```
CREATE DATABASE DATABASE_NAME;
```

We might use the following code to create a database called MY_COMPANY.

```
CREATE DATABASE MY_COMPANY;
```

4.7 Data Definition Language Statements

Let us start with one of the fundamental DDL statements in SQL: the CREATE TABLE statement. There are several types of tables in SQL, but the two basic types are

- Base tables
- Viewed tables (or views or virtual tables).

Base tables are actual tables, and views are "virtual" tables; they are derived from base tables but appear as actual tables to the user.

4.7.1 Create Table

The create table statement is used to create a new table. Here is the format of a simple create table statement:

- create table "tablename"
- ("column1" "data type",
- "column2" "data type",
- "column3" "data type");

The CREATE TABLE statement is used to create base tables. The CREATE TABLE statement requires a table name and a list of columns along with their corresponding data types. There are several optional elements that you can also include in the CREATE statement, but for the moment, let's just deal with the required ones. The basic syntax is as follows:

```
CREATE TABLE table (column datatype);
```

CREATE and TABLE are SQL keywords; table, column, and datatype are placeholders for values that you supply. Notice that column and datatype are enclosed in parentheses. Generally in SQL, parentheses are used to group items together. Here, they are required to enclose the list of column definitions. The semicolon at the end is a statement separator. Use it at the end of each SQL statement.

To create a new table, enter the keywords create table followed by the table name, followed by an open parenthesis, followed by the first column name, followed by the data type for that column, followed by any optional constraints, and followed by a closing parenthesis. It is important to make sure you use an open parenthesis before the beginning table, and a closing parenthesis after the end of the last column definition. Make sure you seperate each column definition with a comma. All SQL statements should end with a ";".

The table and column names must start with a letter and can be followed by letters, numbers, or underscores - not to exceed a total of 30 characters in length. Do not use any SQL reserved keywords as names for tables or column names (such as "select", "create", "insert", etc).

Let us try an example. Suppose you want to keep track of all of your appointments in an appointment table. You could create such a table in SQL by entering the following statement:

```
CREATE TABLE APPOINTMENTS (APPOINTMENT_DATE DATE);
```

Executing this statement creates a table named APPOINTMENTS consisting of a single column named APPOINTMENT_DATE with a data type of DATE. At this point, there are no rows, since the table is empty. (The CREATE TABLE statement only defines tables; the INSERT statement, discussed later, populates them.)

APPOINTMENTS and APPOINTMENT_DATE are called identifiers in SQL, because they identify specific database objects, in this case a particular table and column, respectively. There are two types of identifiers in SQL: regular and delimited. Delimited identifiers are enclosed in double quotation marks and are case sensitive; regular identifiers are neither delimited nor case sensitive. Here we will use only regular identifiers.

Identifiers must conform to certain rules. Regular identifiers can contain letters of the alphabet (any alphabet, not just Latin), numeric digits, and underscores. (If you're familiar with Japanese Hiragana syllabary or Chinese, you can also include syllables and ideographs.) Identifiers can't contain any punctuation, spaces, or special characters (such as #, @, %, or !), nor can they begin with a digit or an underscore. Although it's possible to use some SQL keywords as identifiers, common sense should tell you not to try this. Also, because an identifier identifies something, it has to be unique within its namespace: You can't create a table in a database if a table of that name already exists, and you can't have two columns of the same name within a table. Remember that to SQL, APPOINTMENTS and Appointments are the same. Varying the case won't make a regular identifier unique.

Technically, although you need only one column to create a table, to be at all useful most tables have more than one column. Let us change our syntax diagram slightly to reflect the optional columns:

```
CREATE TABLE table (
column datatype
[ { , column datatype } ] ) ;
```

We used square brackets to denote optional elements and curly braces to denote an element that can be repeated any number of times. (You wouldn't include these special characters in the actual SQL statement.) The syntax diagram now shows that we can include any number of columns. Notice the comma before the second column placeholder. Whenever you specify a list of items in SQL, you have to separate the items with commas.

Since our single-column APPOINTMENTS table isn't very realistic, let us create a more useful one. (We will have to give it a different name, though, because we already have an APPOINTMENTS table in the database.)

```
CREATE TABLE APPOINTMENTS2 (
    APPOINTMENT_DATE DATE,
    APPOINTMENT_TIME TIME,
    DESCRIPTION VARCHAR ( 256 ) ) ;
```

Here we have defined a table called APPOINTMENTS2. As in our first example, it has a column called APPOINTMENT_DATE, to record the date of the appointment, and APPOINTMENT_TIME, to record the time. DESCRIPTION is a variable-length string of text of up to 256 characters. We used the VARCHAR (short for CHARACTER VARYING) data type, because we weren't sure how much space we would need, but we knew we didn't want more than 256 characters. Character strings as well as some other data types take length assignments. These assignments have to be enclosed in parentheses to the right of the data type.

You may have noticed that we changed the spacing in our two examples. The first time, the entire statement was written in one continuous string, but the second time, we broke the line after the opening parenthesis and moved each column definition to a new line. There's no rule in SQL that says we had to do that. Listing columns in a stack just improves readability. SQL is a free-form language to some extent. You can break up the lines or add indentation and spaces however you see fit. Use whatever spacing is easiest to read and understand.

For example, you might see the following SQL code to create a table called PRODUCTS with three columns in the MY_COMPANY database we just created. Note that the three columns would be P_NUM which would be an integer value and could not be null, the P_QUANTITY which would also accept integers as values, and the P_PRICE column which would accept decimal numbers with 8 digits before and 2 digits after the decimal point.

```
CREATE TABLE PRODUCTS (P_NUM INT NOT NULL,
                                    P_QUANTITY INT,
                                    P_PRICE DECIMAL(8,2))
        IN DATABASE MY_COMPANY;
```

Notice that as we mentioned before, when you create a table, you must specify the data type for each column. Notice also that you may use the "NOT NULL" keyword to tell the database that it should not allow any NULL values to be added to the column.

As a final note, I would like to mention that you can also typically create Views, Indexes, and Synonyms, however, those topics are beyond the scope of this tutorial since you will most likely not be doing database administration types of activities. For most web-development work, it is simply enough to define some tables.

4.7.2 SQL Data Types

Data types specify what the type of data can be for that particular column. If a column called "Last_Name", is to be used to hold names, then that particular column should have a "varchar" (variable-length character) data type.

Type	Alias	Description
CHARACTER	CHAR	Contains a string of characters. Usually, these fields will have a specified maximum length that is defined when the table is created.
NUMERIC	NONE	Contains a number with a specified number of decimal digits and scale (indicating a power to which the value should be multiplied) defined at the table creation.
DECIMAL	DEC	Similar to NUMERIC except that it is more proprietary.
INTEGER	INT	Only accepts integers
SMALLINT	NONE	Same as INTEGER except that precision must be smaller than INT precisions in the same table.
FLOAT	NONE	Contains floating point numbers
DOUBLE	PRECISION	NONE Like FLOAT but with greater precision

It is important to note that not all databases will implement the entire list and that some will implement their own data types such as calendar or monetary types. Some fields may also allow a NULL value in them even if NULL is not exactly the correct type.

Okay, we will explain data types when we actually start using them, so for now, let us go on to some real examples of doing things with SQL. Let us log on to a database and start executing queries using SQL.

The following data types are usually supported by SQL though the SQL standards are supporting more than these:

Numeric data:	INTEGER	signed integer 31 bits
	SMALLINT	signed integer 15 bits
	DECIMAL(p,q)	signed number p digits, q decimals
	FLOAT(p)	floating point number, p bits precision
String data:	CHAR(n)	fixed length string, of n-8 bits
	VARCHAR(n)	varying length string, up to n-8 bits
	GRAPHIC(n)	fixed length string, n-16 bits
	VARGRAPHIC(n)	varying length string n-16 bits
Date/time data:	DATE	date (yyyymmmdd)
	TIME	time (hhmmss)
	TIMESTAMP	combination of date and time

Many other types are also possible e.g. LOGICAL, BIT, MONEY etc.

4.7.3 Constraints

What are constraints? When tables are created, it is common for one or more columns to have constraints associated with them. A constraint is basically a rule associated with a column that the data entered into that column must follow. For example, a "unique" constraint specifies that no two records can have the same value in a particular column. They must all be unique. The other two most popular constraints are "not null" which specifies that a column can't be left blank, and "primary key". A "primary key" constraint defines a unique identification of each record (or row) in a table. All of these and more will be covered in the future Advanced release of this Tutorial. Constraints can be entered in this SQL interpreter, however, they are not supported in this Intro to SQL tutorial & interpreter. They will be covered and supported in the future release of the Advanced SQL tutorial - that is, if "response" is good.

It is now time for you to design and create your own table. You will use this table throughout the rest of the tutorial. If you decide to change or redesign the table, you can either drop it and recreate it or you can create a completely different one. The SQL statement drop will be covered later.

Format of create table if you were to use optional constraints:

- **create table "tablename"**
- **("column1" "data type" [constraint],**
- **"column2" "data type" [constraint],**
- **"column3" "data type" [constraint]);**
- **[ ] = optional**

Note: You may have as many columns as you would like, and the constraints are optional.

Example:

- create table employee
- (first varchar(15),
- last varchar(20),
- age number(3),
- address varchar(30),
- city varchar(20),
- state varchar(20));

4.7.4 Deleting Databases and Dropping Tables and Views

It is also simple to delete tables and databases using GUI tools or SQL. In the case of a GUI, you will simply select a table or database from your main menu and delete it as you would a file in a file system manager.

In SQL, you would simply use the DELETE or DROP commands depending on if you were deleting a whole database or just a table in a single database.

In the case of deleting a whole database, you will use the DELETE command as follows:

```
DELETE DATABASE DATABASE_NAME;
```

The following example would delete the database MY_COMPANY:

```
DELETE DATABASE MY_COMPANY;
```

In the case of a table, you use the DROP command:

```
DROP TABLE TABLE_NAME;
```

such as:

```
DROP TABLE EMPLOYEES;
```

Essentially, when you use delete and drop, you are modifying the database management system's data dictionary. Hence, ...BE CAREFUL WHEN DELETING OR DROPPING!

4.7.5 Altering a Table

Finally, you should know that it is possible to "alter" a table after it has been created using either a standard GUI tool or by using the ALTER SQL command as follows:

```
ALTER TABLE TABLE_NAME
DROP COLUMN_NAME, COLUMN_NAME
ADD COLUMN_NAME DATA_TYPE, COLUMN_NAME DATA_TYPE
RENAME COLUMN_NAME NEW NAME
MODIFY COLUMN_NAME DATA_TYPE;
```

such as the following case in which we alter the table named EMPLOYEES by dropping the E_GENDER Column and adding an E_ZIP column which will accept INTEGERS and which must be filled in for every new employee added to the table, and the E_MIDDLE_INIT column which will accept a single character as a value.

```
ALTER TABLE EMPLOYEES
DROP E_GENDER
ADD E_ZIP INTEGER NOT NULL,
E_MIDDLE_INIT CHAR (1);
```

4.7.6 Creating Multicolumn Table

Now that you have the basic form under your belt, let us create a more complex multi-column table. On the previous page, there is a table called EMPLOYEES which has columns for last name, first name, date-of-hire, branch office, salary grade level, and annual salary. To define this table, enter the following SQL statement:

```
CREATE TABLE EMPLOYEES (
LAST_NAME CHARACTER ( 13 ) NOT NULL,
FIRST_NAME CHARACTER ( 10 ) NOT NULL,
HIRE_DATE DATE ,
BRANCH_OFFICE CHARACTER ( 15 ) ,
GRADE_LEVEL SMALLINT ,
SALARY DECIMAL ( 9 , 2 ) ) ;
```

As you can see, we've introduced several new elements.

First off, the NOT NULL clauses at the end of the column definitions for LAST_NAME and FIRST_NAME are examples of a constraint. A constraint sets requirements that have to be met. In this case, we're telling the system that LAST_NAME and FIRST_NAME must have values when data is entered into the table; these columns can't be left blank.

Also, our example shows three new data types: CHARACTER, SMALLINT, and DECIMAL. We haven't said much about data types up until now. Although SQL unfortunately doesn't support relational domains, it does support a set of basic data types. It uses them to allocate storage, constrain comparisons, and restrict data entry to some extent, but it doesn't have true type-checking the way other languages do.

SQL data types fall into six categories: character strings, exact numerics, approximate numerics, bit strings, datetimes, and intervals. We've listed all the categories here for completeness, even though we won't be discussing them all in this article. (Bit strings won't mean much to non-programmers.)

By the way, if you are wondering whether datetime is a typo, it isn't. It is the term the SQL standard uses to categorize most temporal data types. (Intervals are also temporal, but they've been given their own category.) You have already been introduced to two of the datetime data types, DATE and TIME, in our previous example.

Another data type you have already encountered, CHARACTER VARYING (or VARCHAR for short), is in the character string category. One of our new data types, CHARACTER or CHAR, is also a character string. But unlike VARCHAR, which is of varying length, CHAR is a fixed-length data type. LAST_NAME will always be 13 characters long, whether the employee's name is Poe or Pennworth-Chickering. (In Poe's case, the remaining 10 characters would be padded by blanks.)

From the user's perspective, VARCHAR and CHAR look the same, so why have two types? The reasons mainly have to do with storage requirements and performance. Generally, fixed-length character strings provide slightly faster access time, but unusually long ones waste space. In our APPOINTMENT2 example, it wouldn't make much sense to set aside 256 characters for every appointment's description; most entries wouldn't require that much space. A name,

on the other hand, also varies in length, but is usually 13 characters or fewer, so any lost space would be negligible. A good rule of thumb is, if you know that the length of a text field is likely to be uniform or relatively short, use CHAR; otherwise, use VARCHAR.

Our other new data types, SMALLINT and DECIMAL, are examples of exact numeric data types. SMALLINT is short for small integer. SQL also provides an INTEGER data type. Again, the reason for having two types has to do with storage. In our example, we knew that the GRADE_LEVEL column would never be larger than two digits, so SMALLINT was appropriate, but sometimes you don't know how large the values will be. When in doubt, use INTEGER. The actual precision and storage of SMALLINT and INTEGER will vary depending on the system you are using.

The DECIMAL data type, usually used for monetary values, lets you specify amounts with a fixed number of decimal places. Because it is an exact numeric data type, you get back exact results when using it in mathematical operations involving decimals. (The approximate numeric data types such as FLOAT [floating point number] have rounding errors in their decimal values, so they are not appropriate for financial applications.) To define a DECIMAL data type, use the form shown in our example:

```
DECIMAL(p,d)
```

The p stands for precision and the d stands for decimal places. Replace p with the total number of digits you will need and d with the number of digits you want placed to the right of the decimal point.

Now look at Listing A to see our updated CREATE TABLE syntax diagram, which reflects these new elements and shows the format for all the data types we have talked about so far. (There are other data types, but these are enough to get you going.)

SQL syntax diagrams can get complicated fast, but since you have looked at some examples, this one shouldn't be too hard to understand. We have added one more symbol to our notation—a vertical bar used to separate alternatives. In effect, we are saying that for each column definition you have a choice of data types. You have to choose one data type for each column. (As before, anything enclosed in square brackets is optional and anything enclosed in curly braces can be repeated any number of times. Except for parentheses, you would not include these special characters in actual SQL statements.) To save space, we have given the full data type names in the first set of alternatives and abbreviated names in the second set, but you can use them interchangeably.

4.8 Restructuring a Table with SQL

To add a new column to a table:

```
ALTER TABLE CUSTOMER
ADD CUSTTYPE CHAR(1)
```

To add a new column to a table with a default value:

```
ALTER TABLE CUSTOMER
ADD CUSTTYPE CHAR(1) INIT='R'
```

To delete a column from a table:

```
ALTER TABLE PART
DELETE WAREHOUSE_NUMBER
```

To change the data type of an existing column:

```
ALTER TABLE CUSTOMER
CHANGE COLUMN LAST TO CHAR(20)
```

4.9 Data Manipulation Language

The Data Manipulation Language (DML) is used to retrieve, insert and modify database information. These commands will be used by all database users during the routine operation of the database. Let us take a brief look at the basic DML commands:

- **INSERT**

The INSERT command in SQL is used to add records to an existing table. Returning to the personal_info example from the previous section, let us imagine that our HR department needs to add a new employee to their database. They could use a command similar to the one shown below:

```
INSERT INTO personal_info
values('bart','simpson',12345,$45000)
```

Note that there are four values specified for the record. These correspond to the table attributes in the order they were defined: first_name, last_name, employee_id, and salary.

- **SELECT**

The select statement is used to query the database and retrieve selected data that match the criteria that you specify. The SELECT command is the most commonly used command in SQL. It allows database users to retrieve the specific information they desire from an operational database. Let's take a look at a few examples, again using the personal_info table from our employees database.

The command shown below retrieves all of the information contained within the personal_info table. Note that the asterisk is used as a wildcard in SQL. This literally means "Select everything from the personal_info table."

```
SELECT *
FROM personal_info
```

Alternatively, users may want to limit the attributes that are retrieved from the database. For example, the Human Resources department may require a list of the last names of all employees in the company. The following SQL command would retrieve only that information:

```
SELECT last_name
FROM personal_info
```

Finally, the WHERE clause can be used to limit the records that are retrieved to those that meet specified criteria. The CEO might be interested in reviewing the personnel records of all highly paid employees. The following command retrieves all of the data contained within personal_info for records that have a salary value greater than $50,000:

```
SELECT *
FROM personal_info
WHERE salary > $50000
```

Here is the format of a simple select statement:

```
select "column1"[,"column2",etc] from "tablename"
[where "condition"];
[] = optional
```

- The column names that follow the select keyword determine which columns will be returned in the results. You can select as many column names that you would like, or you can use a "*" to select all columns.
- The table name that follows the keyword from specifies the table that will be queried to retrieve the desired results.
- The where clause (optional) specifies which data values or rows will be returned or displayed, based on the criteria described after the keyword where.

Conditional selections used in where clause:

=	Equal
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to
<>	Not equal to

- LIKE *See note below

*The LIKE pattern matching operator can also be used in the conditional selection of the where clause. Like is a very powerful operator that allows you to select only rows that are "like" what you specify. The percent sign "%" can be used as a wild card to match any possible character that might appear before or after the characters specified. For example:

- select first, last, city
- from empinfo
- where first LIKE 'Er%';

This SQL statement will match any first names that start with 'Er'. Strings must be in single quotes.

Or you can specify,

```
select first, last
from empinfo
where last LIKE '%s';
```

This statement will match any last names that end in a 's'.

```
select * from empinfo
where first = 'Eric';
```

This will only select rows where the first name equals 'Eric' exactly.

Sample table called "empinfo"

first id	last id	age	city	state
John	Jones 99980	45	Payson	Arizona
Mary	Jones 99982	25	Payson	Arizona
Eric	Edwards 88232	32	San Diego	California
Mary	Ann Edwards 88233	32	Phoenix	Arizona
Ginger	Howell 98002	42	Cottonwood	Arizona
Sebastian	Smith 92001	23	Gila Bend	Arizona
Gus	Gray 22322	35	Bagdad	Arizona
Mary	Ann May 32326	52	Tucson	Arizona
Erica	Williams 32327	60	Show Low	Arizona
Leroy	Brown 32380	22	Pinetop	Arizona
Elroy	Cleaver 32382	22	Globe	Arizona

Enter the following sample select statements in the SQL Interpreter Form at the bottom of this page. Before you press "submit", write down your expected results. Press "submit", and compare the results.

```
select first, last, city from empinfo;
select last, city, age from empinfo
where age > 30;
select first, last, city, state from empinfo
where first LIKE 'J%';
select * from empinfo;
select first, last, from empinfo
where last LIKE '%s';
select first, last, age from empinfo
where last LIKE '%illia%';
select * from empinfo where first = 'Eric';
UPDATE
```

The UPDATE command can be used to modify information contained within a table, either in bulk or individually. Each year, our company gives all employees a 3% cost-of-living increase in their salary. The following SQL command could be used to quickly apply this to all of the employees stored in the database:

```
UPDATE personal_info
SET salary = salary * 1.03
```

On the other hand, our new employee Bart Simpson has demonstrated performance above and beyond the call of duty. Management wishes to recognize his stellar accomplishments with a $5,000 raise. The WHERE clause could be used to single out Bart for this raise:

```
UPDATE personal_info
SET salary = salary + $5000
```

```
WHERE employee_id = 12345
DELETE
```

Finally, let us take a look at the DELETE command. You will find that the syntax of this command is similar to that of the other DML commands. Unfortunately, our latest corporate earnings report didn't quite meet expectations and poor Bart has been laid off. The DELETE command with a WHERE clause can be used to remove his record from the personal_info table:

```
DELETE FROM personal_info
WHERE employee_id = 12345
```

4.10 Relational Model Basics

In the relational model, a relation is represented as a table of information. It has one or more attributes, which correspond to the columns of the table, and zero or more instances of data with those attributes (called n-tuples or simply tuples), which correspond to the rows (see the table "The Relational Model," below).

For any given tuple, the actual values of its attributes must be taken from the attributes' domains. A domain is essentially a data type that defines the set of all permissible values.

For example, assume there's a domain called "Days-of-the-week" that consists of "Monday" through "Sunday." If a relation had a single attribute of that domain called WEEKDAY, every tuple in the relation would have to contain one of those values in its WEEKDAY column. A WEEKDAY value of "January" or "Cat" wouldn't be allowed.

Notice that we said an attribute had to contain one of those values. It couldn't contain more than one. In addition to being constrained to a domain, attribute values have to be atomic. That is, they have to be elementary; they can't be broken down into smaller parts without losing their meaning. An attribute's value containing both "Monday" and "Tuesday" can be broken down into two parts that still represent days of the week, so that value is not atomic. But if you broke "Monday" down into smaller parts— the letters M through y — they wouldn't mean anything by themselves, so "Monday" is an atomic value.

Relations have other properties as well. Most important, they have the mathematical property of closure. That is, any operation performed on a relation yields another relation. This allows you to perform mathematical operations on relations with predictable results. It also allows operations to be abstracted into variable expressions and nested.

In his original paper, Dr. Codd defined a collection of eight operators called relational algebra. Four of those operators, union, intersection, difference, and Cartesian product, were based on traditional set theory, but the rest were developed specifically for relations. Since then, Dr. Codd, Chris Date, and others have developed more operators. We will discuss three of these relational operators — project, select (or restrict), and join — later in this article.

4.11 What is a view?

Creating a view with SQL

To create a new view called "Housewares" based on the PART table:

```
CREATE VIEW HOUSEWARES AS
SELECT PART_NUMBER, PART_DESCRIPTION, UNITS_ON_HAND, UNIT_PRICE
FROM PART
WHERE ITEM_CLASS='HW"
```

To create a new view called "Housewares" with new column names:

```
CREATE VIEW HOUSEWARES (PNUM, DESC,OHAND, PRICE) AS
SELECT PART_NUMBER, PART_DESCRIPTION, UNITS_ON_HAND, UNIT_PRICE
FROM PART
WHERE ITEM_CLASS='HW"
```

4.11.1 Advantages of Using Views

- Views provide data independence;
- Data can be viewed in different ways by different users ;
- A view can contain only those columns required by a given user, making the database seem simpler than it is to the user; and
- A view can provide a measure of security to the database by eliminating sensitive columns from the views of unauthorized users.

4.11.2 SQL View

When you submit a query to an SQL database using SQL, the database will consult its data dictionary and access the tables you have requested data from. It will then put together a "view" based upon the criteria you have defined in your SQL query.

A "view" is essentially a dynamically generated "result" table that is put together based upon the parameters you have defined in your query. For example, you might instruct the database to give you a list of all the employees in the EMPLOYEES table with salaries greater than 50,000 USD per year. The database would check out the EMPLOYEES table and return the requested list as a "virtual table".

Similarly, a view could be composed of the results of a query on several tables all at once (sometimes called a "join"). Thus, you might create a view of all the employees with a salary of greater than 50K from several stores by accumulating the results from queries to the EMPLOYEES and STORES databases. The possibilities are limitless.

By the way, many databases allow you to store "views" in the data dictionary as if they were physical tables.

4.12 The Data Dictionary

How does the database know where all of these tables are located? Well, behind the scenes, the database maintains a "data dictionary" (a.k.a. catalog) which contains a list of all the tables in the database as well as pointers to their locations.

The system catalog, or data dictionary, is at the heart of any general-purpose DBMS. Essentially, the data dictionary is a table of tables containing a list of all the tables in the database, as well as the structure of the tables and often, special information about the database itself.

When you use SQL to talk to the database and provide a table name, the database looks up the table you referred to in the data dictionary. Of course, you needn't worry about the data dictionary; the database does all the searching itself. As we said before, you just need to know the name of the table you want to look at.

It is interesting to note that because the data dictionary is a table, in many databases, you can even query the data dictionary itself to get information about your environment. This can often be a very useful tool when exploring a new database.

Okay, so how do you actually grab table data using the data dictionary? Well, in an SQL database you create "views". Let's examine views a bit.

It is a "minidatabase" in itself whose function is to store the descriptions of the databases that the DBMS maintains. It stores data that describes each database, often called meta-data. This descriptor information is essential if the system is to do it's job properly e.g. it uses the catalog information about indexes to choose a specific access strategy, and it may use the catalog information about users and access privileges to grant or deny specific requests.

The catalog itself consists of a number of relations or tables, which hold all the data, but which will also include entries for the catalog tables themselves. These entries (entries regarding the catalog tables) are created automatically by the system as part of the system installation procedure as opposed to being created explicitly by the CREATE TABLE statement. However, methods do exist for updating and querying the catalog.

4.13 SQL Standardisation

At present, there are several standards for SQL but the goal it have DBMS products conform to one standard in order to increase competition among products and to promote inter-operability. The standards are as follows:

- SQL-86 was defined in 1986 as a bare bones standard and defined the union or common features of the most important DBMS products at thet time.
- SQL-89 is a higher version of SQL-86 and has a small number of new features.
- SQL-92 or SQL-2 was completed in 1992 and is significantly larger than SQL-89 including extra data types, outer joins, catalog specifications, domains and assertions, etc.
- SQL-86 and SQL-89 were just catching up with what was already on offer in products but SQL-2 has features not found yet in many products.

Conformance to SQL standards is becoming mandatory for many government agencies. In terms of work in progress, SQL-3 is scheduled for completion in 1995 or 1996 and includes several extensions on SQL-2 in the direction of type systems, stored procedures and object orientation.

4.14 Structured Query Language : MySQL

The SQL used in this document is "ANSI", or standard SQL i.e MySQL.

4.14.1 SQL Data Type

In SQL statement, data type is critical. Sometimes, it is even more important than the statement itself. Without using the correct data type, your SQL statement will generate "Type Mismatch" error message. There are 3 most commonly used data type:

- Number (In Access: AutoNumber, Number, Integer, Long, Double, Currency, etc.)
- String (In Access: Text, Memo)
- Date/Time (In Access: Date/Time)

Different data type has different syntax. This applies to WHERE, INSERT and UPDATE clause:

```
mySQL = "SELECT * FROM table_name WHERE fldID = 3"
mySQL = "SELECT * FROM table_name WHERE fldFirstName = 'Frank'"
mySQL = "SELECT * FROM table_name WHERE fldDate = #05/06/99#"
```

For number, nothing is needed; for string, a pair of single quote (') is required; for date/time, a pair of pound sign (#) is needed. Without these special characters, your SQL statement won't work!

4.14.2 Basics of the SELECT Statement

In a relational database, data is stored in tables. An example table would relate Social Security Number, Name, and Address:

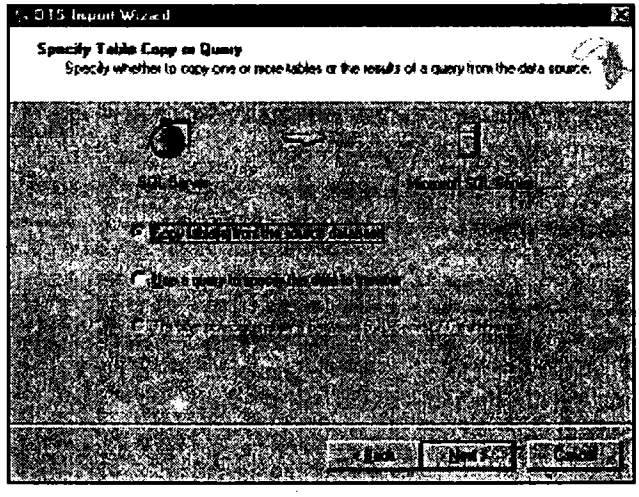


Fig. 4.4

Now, let us say you want to see the address of each employee. Use the SELECT statement, like so:

```
SELECT FirstName, LastName, Address, City, State
FROM EmployeeAddressTable;
```

The following is the results of your query of the database:

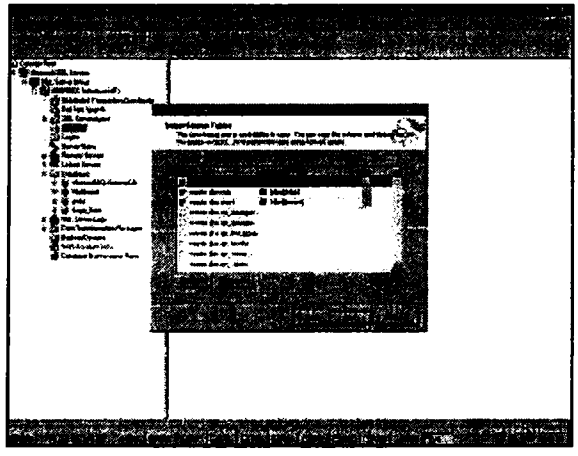


Fig. 4.5

To explain what you just did, you asked for the all of data in the EmployeeAddressTable, and specifically, you asked for the columns called FirstName, LastName, Address, City, and State. Note that column names and table names do not have spaces...they must be typed as one word; and that the statement ends with a semicolon (;). The general form for a SELECT statement, retrieving all of the rows in the table is:

```
SELECT ColumnName, ColumnName, ...
FROM TableName;
```

To get all columns of a table without typing all column names, use:

```
SELECT * FROM TableName;
```

Each database management system (DBMS) and database software has different methods for logging in to the database and entering SQL commands; see the local computer "guru" to help you get onto the system, so that you can use SQL.

```
mySQL = "SELECT * FROM table_name"
RESULT: Retrieve all records with all fields
mySQL = "SELECT fldFirstName, fldLastName FROM table_name"
RESULT: Retrieve all records with only First Name and Last Name
mySQL = "SELECT * FROM table_name ORDER BY fldDate"
RESULT: Retrieve all records sort by Date
mySQL = "SELECT * FROM table_name ORDER BY fldDate DESC"
RESULT: Retrieve all records sort by Date descending (Latest
  date goes first)
mySQL = "SELECT DISTINCT fldState FROM tblUSCity ORDER BY
  fldState"
```

RESULT: Retrieve all records from fldState where all duplicates have been eliminated sort by fldState

4.14.3 Conditional Selection

To further discuss the SELECT statement, let us look at a new example table (for hypothetical purposes only):

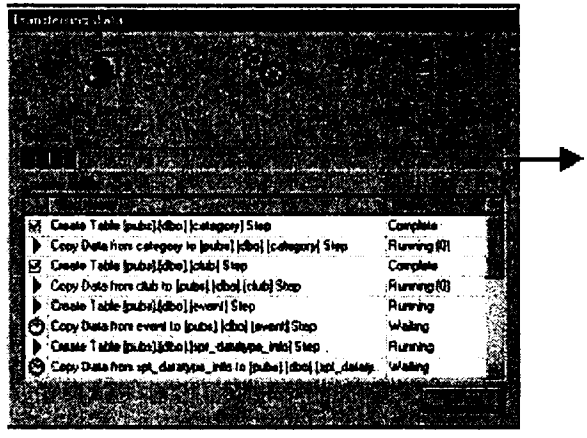


Fig. 4.6

```
mySQL = "SELECT * FROM table_name WHERE fldFirstName = 'Frank'"
RESULT: Retrieve only the record(s) where fldFirstName is
 "Frank"
mySQL = "SELECT * FROM table_name WHERE fldSalary >= 50000"
RESULT: Retrieve only the record(s) where fldSalary is greater
 than or equal to 50000
mySQL = "SELECT * FROM table_name WHERE fldDate >= #" & Date -
 7 & "#"
RESULT: Retrieve only the record(s) where fldDate is within the previous
7 days

mySQL = "SELECT * FROM table_name WHERE fldLastName LIKE 'f%'"
RESULT: Retrieve only the record(s) where fldLastName starts
 with letter "f"
mySQL = "SELECT * FROM table_name WHERE fldLastName LIKE '%er'"
RESULT: Retrieve only the record(s) where fldLastName ends with
 "er"
mySQL = "SELECT * FROM table_name WHERE fldLastName LIKE
 '%frank%'"
RESULT: Retrieve only the record(s) where fldLastName contains "frank"
anywhere (even partial of a word)
```

4.14.4 Relational Operators

There are six Relational Operators in SQL, and after introducing them, we will see how they are used:

```
= Equal
< or != (see manual) Not Equal
< Less Than
> Greater Than
<= Less Than or Equal To
>= Greater Than or Equal To
```

The WHERE clause is used to specify that only certain rows of the table are displayed, based on the criteria described in that WHERE clause. It is most easily understood by looking at a couple of examples.

If you wanted to see the EMPLOYEEIDNO's of those making at or over $50,000, use the following:

```
SELECT EMPLOYEEIDNO
FROM EMPLOYEESTATISTICSTABLE
WHERE SALARY >= 50000;
```

Notice that the >= (greater than or equal to) sign is used, as we wanted to see those who made greater than $50,000, or equal to $50,000, listed together. This displays:

```
EMPLOYEEIDNO
    010
    105
    152
    215
    244
```

The WHERE description, SALARY >= 50000, is known as a condition (an operation which evaluates to True or False). The same can be done for text columns:

```
SELECT EMPLOYEEIDNO
FROM EMPLOYEESTATISTICSTABLE
WHERE POSITION = 'Manager';
```

This displays the ID Numbers of all Managers. Generally, with text columns, stick to equal to or not equal to, and make sure that any text that appears in the statement is surrounded by single quotes ('). Note: Position is now an illegal identifier because it is now an unused, but reserved, keyword in the SQL-92 standard.

4.14.5 More Complex Conditions: Compound Conditions/Logical Operators

The AND operator joins two or more conditions, and displays a row only if that row's data satisfies ALL conditions listed (i.e. all conditions hold true). For example, to display all staff making over $40,000, use:

```
SELECT EMPLOYEEIDNO
FROM EMPLOYEESTATISTICSTABLE
WHERE SALARY > 40000 AND POSITION = 'Staff';
```

The OR operator joins two or more conditions, but returns a row if ANY of the conditions listed hold true. To see all those who make less than $40,000 or have less than $10,000 in benefits, listed together, use the following query:

```
SELECT EMPLOYEEIDNO
FROM EMPLOYEESTATISTICSTABLE
WHERE SALARY < 40000 OR BENEFITS < 10000;
AND & OR can be combined, for example:
SELECT EMPLOYEEIDNO
FROM EMPLOYEESTATISTICSTABLE
WHERE POSITION = 'Manager' AND SALARY > 60000 OR BENEFITS >
12000;
```

First, SQL finds the rows where the salary is greater than $60,000 and the position column is equal to Manager, then taking this new list of rows, SQL then sees if any of these rows satisfies the previous AND condition or the condition that the Benefits column is greater than $12,000. Subsequently, SQL only displays this second new list of rows, keeping in mind that anyone with Benefits over $12,000 will be included as the OR operator includes a row if either resulting condition is True. Also note that the AND operation is done first.

To generalize this process, SQL performs the AND operation(s) to determine the rows where the AND operation(s) hold true (remember: all of the conditions are true), then these results are used to compare with the OR conditions, and only display those remaining rows where any of the conditions joined by the OR operator hold true (where a condition or result from an AND is paired with another condition or AND result to use to evaluate the OR, which evaluates to true if either value is true). Mathematically, SQL evaluates all of the conditions, then evaluates the AND "pairs", and then evaluates the OR's (where both operators evaluate left to right).

To look at an example, for a given row for which the DBMS is evaluating the SQL statement Where clause to determine whether to include the row in the query result (the whole Where clause evaluates to True), the DBMS has evaluated all of the conditions, and is ready to do the logical comparisons on this result:

```
True AND False OR True AND True OR False AND False
```

First, simplify the AND pairs:

```
False OR True OR False
```

Now do the OR's, left to right:

```
True OR False
True
```

The result is True, and the row passes the query conditions. Be sure to see the next section on NOT's, and the order of logical operations.

To perform OR's before AND's, like if you wanted to see a list of employees making a large salary ($50,000) or have a large benefit package ($10,000), and that happen to be a manager, use parentheses:

```
SELECT EMPLOYEEIDNO
FROM EMPLOYEESTATISTICSTABLE
WHERE POSITION = 'Manager' AND (SALARY > 50000 OR BENEFITS >
10000);
```

Some Examples

```
mySQL = "SELECT * FROM table_name WHERE fldSex = 'Male' AND
fldAge >= 21"
```

Result: Retrieve only the record(s) where the following two conditions must all be satisfied: fldSex is Male and fldAge is greater than or equal to 21

```
mySQL = "SELECT * FROM table_name WHERE fldSex = 'Female' OR fldAge >=
21"
```

Result: Retrieve only the record(s) where any of the following two conditions is satisfied: fldSex is Female or fldAge is greater than or equal to 21

```
mySQL = "SELECT * FROM table_name WHERE fldPosition = 'Manager' AND _
(fldSalary > 50000 OR fldBenefit > 10000)"
```

Result: Retrieve only the record(s) where fldPostition must be Manager and fldSalary is greater than 50000 or fldBenefit is greater than 10000

```
mySQL = "SELECT * FROM table_name WHERE fldLastName IN
('Andrews','Dixon')"
```

Result: Retrieve only the record(s) where fldLastName is either Andrews or Dixon

```
mySQL = "SELECT * FROM table_name WHERE fldSalary BETWEEN 30000 AND
50000"
```

Result: Retrieve only the record(s) where fldSalary is between 30000 and 50000

```
mySQL = "SELECT * FROM table1, table2 WHERE table1.fldID =
table2.fldOwnerID"
```

Result: Retrieve only the record(s) from both table1 and table2 where fldID in table1 matches fldOwnerID in table2

4.14.6 Aggregate Operators

```
mySQL = "SELECT MAX(fldPrice) FROM tblBook"
```

Result: Find out the highest price from table Book

```
mySQL = "SELECT AVG(fldAge) FROM tblEmployee"
```

Result: Find out the average age of all employees.

```
mySQL = "SELECT COUNT(fldSalary) FROM tblEmployees WHERE
 fldSalary > 30000"
```

Result: Find out how many employees have salary higher than 30,000.

```
mySQL = "SELECT MIN(fldAge) FROM tblStudents"
```

Result: Find out the youngest student.

```
mySQL = "SELECT SUM(fldHits) FROM tblAllWebSites"
```

Result: Find out the total hits of all the websites.

4.14.7 IN & BETWEEN

An easier method of using compound conditions uses IN or BETWEEN. For example, if you wanted to list all managers and staff:

```
SELECT EMPLOYEEIDNO
FROM EMPLOYEESTATISTICSTABLE
WHERE POSITION IN ('Manager', 'Staff');
```

or to list those making greater than or equal to $30,000, but less than or equal to $50,000, use:

```
SELECT EMPLOYEEIDNO
FROM EMPLOYEESTATISTICSTABLE
WHERE SALARY BETWEEN 30000 AND 50000;
```

To list everyone not in this range, try:

```
SELECT EMPLOYEEIDNO
FROM EMPLOYEESTATISTICSTABLE
WHERE SALARY NOT BETWEEN 30000 AND 50000;
```

Similarly, NOT IN lists all rows excluded from the IN list.

Additionally, NOT's can be thrown in with AND's & OR's, except that NOT is a unary operator (evaluates one condition, reversing its value, whereas, AND's & OR's evaluate two conditions), and that all NOT's are performed before any AND's or OR's.

SQL Order of Logical Operations (each operates from left to right)
NOT
AND
OR

4.14.8 Using LIKE

Look at the EmployeeStatisticsTable, and say you wanted to see all people whose last names started with "S"; try:

```
SELECT EMPLOYEEIDNO
FROM EMPLOYEEADDRESSTABLE
WHERE LASTNAME LIKE 'S%';
```

The percent sign (%) is used to represent any possible character (number, letter, or punctuation) or set of characters that might appear after the "S". To find those people with LastName's ending in "S", use '%S', or if you wanted the "S" in the middle of the word, try '%S%'. The '%' can be used for any characters in the same position relative to the given characters. NOT LIKE displays rows not fitting the given description. Other possiblities of using LIKE, or any of these discussed conditionals, are available, though it depends on what DBMS you are using; as usual, consult a manual or your system manager or administrator for the available features on your system, or just to make sure that what you are trying to do is available and allowed. This disclaimer holds for the features of SQL that will be discussed below. This section is just to give you an idea of the possibilities of queries that can be written in SQL.

4.14.9 Joins

In this section, we will only discuss inner joins, and equijoins, as in general, they are the most useful. For more information, try the SQL links at the bottom of the page.

Good database design suggests that each table lists data only about a single entity, and detailed information can be obtained in a relational database, by using additional tables, and by using a join. First, take a look at these example tables:

AntiqueOwners

OwnerID	OwnerLastName	OwnerFirstName
01	Jones	Bill
02	Smith	Bob
15	Lawson	Patricia
21	Akins	Jane
50	Fowler	Sam

Fig. 4.7

Table *Store_Information*

store_name	Sales	Date
Los Angeles	$1500	Jan-05-1999
San Diego	$250	Jan-07-1999
Los Angeles	$300	Jan-08-1999
Boston	$700	Jan-08-1999

Fig. 4.8

Antiques

SellerID	BuyerID	Item
01	50	Bed
02	15	Table
15	02	Chair
21	50	Mirror
50	01	Desk
01	21	Cabinet
02	21	Coffee Table
15	50	Chair
01	15	Jewelry Box
02	21	Pottery
21	02	Bookcase
50	01	Plant Stand

Fig. 4.9

4.14.10 Keys

First, let us discuss the concept of keys. A primary key is a column or set of columns that uniquely identifies the rest of the data in any given row. For example, in the AntiqueOwners table, the OwnerID column uniquely identifies that row. This means two things: no two rows can have the same OwnerID, and, even if two owners have the same first and last names, the OwnerID column ensures that the two owners will not be confused with each other, because the unique OwnerID column will be used throughout the database to track the owners, rather than the names.

A foreign key is a column in a table where that column is a primary key of another table, which means that any data in a foreign key column must have corresponding data in the other table where that column is the primary key. In DBMS-speak, this correspondence is known as referential integrity. For example, in the Antiques table, both the BuyerID and SellerID are foreign keys to the primary key of the AntiqueOwners table (OwnerID; for purposes of argument, one has to be an Antique Owner before one can buy or sell any items), as, in both tables, the ID rows are used to identify the owners or buyers and sellers, and that the OwnerID is the primary key of the AntiqueOwners table. In other words, all of this "ID" data is used to refer to the owners, buyers, or sellers of antiques, themselves, without having to use the actual names.

4.14.11 Performing a Join

The purpose of these keys is so that data can be related across tables, without having to repeat data in every table—this is the power of relational databases. For example, you can find the names of those who bought a chair without having to list the full name of the buyer in the Antiques table...you can get the name by relating those who bought a chair with the names in the AntiqueOwners table through the use of the OwnerID, which relates the data in the two tables. To find the names of those who bought a chair, use the following query:

```
SELECT OWNERLASTNAME, OWNERFIRSTNAME
FROM ANTIQUEOWNERS, ANTIQUES
WHERE BUYERID = OWNERID AND ITEM = 'Chair';
```

Note the following about this query...notice that both tables involved in the relation are listed in the FROM clause of the statement. In the WHERE clause, first notice that the ITEM = 'Chair' part restricts the listing to those who have bought (and in this example, thereby owns) a chair. Secondly, notice how the ID columns are related from one table to the next by use of the BUYERID = OWNERID clause. Only where ID's match across tables and the item purchased is a chair (because of the AND), will the names from the AntiqueOwners table be listed. Because the joining condition used an equal sign, this join is called an equijoin. The result of this query is two names: Smith, Bob & Fowler, Sam.

Dot notation refers to prefixing the table names to column names, to avoid ambiguity, as follows:

```
SELECT ANTIQUEOWNERS. OWNERLASTNAME, ANTIQUEOWNERS. OWNERFIRSTNAME FROM
ANTIQUEOWNERS, ANTIQUES
WHERE ANTIQUES.BUYERID = ANTIQUEOWNERS.OWNERID AND ANTIQUES.ITEM   =
'Chair';
```

As the column names are different in each table, however, this wasn't necessary.

4.14.12 DISTINCT and Eliminating Duplicates

Let us say that you want to list the ID and names of only those people who have sold an antique. Obviously, you want a list where each seller is only listed once—you don't want to know how many antiques a person sold, just the fact that this person sold one (for counts, see the Aggregate Function section below). This means that you will need to tell SQL to eliminate duplicate sales rows, and just list each person only once. To do this, use the DISTINCT keyword.

First, we will need an equijoin to the AntiqueOwners table to get the detail data of the person's LastName and FirstName. However, keep in mind that since the SellerID column in the Antiques table is a foreign key to the AntiqueOwners table, a seller will only be listed if there is a row in the AntiqueOwners table listing the ID and names. We also want to eliminate multiple occurences of the SellerID in our listing, so we use DISTINCT on the column where the repeats may occur (however, it is generally not necessary to strictly put the Distinct in front of the column name).

To throw in one more twist, we will also want the list alphabetized by the LastName, then by FirstName (on a LastName tie). Thus, we will use the ORDER BY clause:

```
SELECT DISTINCT SELLERID, OWNERLASTNAME, OWNERFIRSTNAME
FROM ANTIQUES, ANTIQUEOWNERS
WHERE SELLERID = OWNERID
ORDER BY OWNERLASTNAME, OWNERFIRSTNAME;
```

In this example, since everyone has sold an item, we will get a listing of all of the owners, in alphabetical order by last name. For future reference (and in case anyone asks), this type of join is considered to be in the category of inner joins.

4.14.13 Aliases & In/Subqueries

In this section, we will talk about Aliases, In and the use of subqueries, and how these can be used in a 3-table example. First, look at this query which prints the last name of those owners who have placed an order and what the order is, only listing those orders which can be filled (that is, there is a buyer who owns that ordered item):

```
SELECT OWN.OWNERLASTNAME Last Name, ORD.ITEMDESIRED Item Ordered
FROM ORDERS ORD, ANTIQUEOWNERS OWN
WHERE ORD.OWNERID = OWN.OWNERID
AND ORD.ITEMDESIRED IN
(SELECT ITEM
FROM ANTIQUES);
```

This gives:

```
Last Name Item Ordered
Smith     Table
Smith     Desk
Akins     Chair
Lawson    Mirror
```

There are several things to note about this query:

- First, the "Last Name" and "Item Ordered" in the Select lines gives the headers on the report.
- The OWN & ORD are aliases; these are new names for the two tables listed in the FROM clause that are used as prefixes for all dot notations of column names in the query (see above). This eliminates ambiguity, especially in the equijoin WHERE clause where both tables have the column named OwnerID, and the dot notation tells SQL that we are talking about two different OwnerID's from the two different tables.

Note that the Orders table is listed first in the FROM clause; this makes sure listing is done-off of that table, and the AntiqueOwners table is only used for the detail information (Last Name).

Most importantly, the AND in the WHERE clause forces the In Subquery to be invoked ("= ANY" or "= SOME" are two equivalent uses of IN). What this does is, the subquery is performed, returning all of the Items owned from the Antiques table, as there is no WHERE clause. Then, for a row from the Orders table to be listed, the ItemDesired must be in that returned list of Items owned from the Antiques table, thus listing an item only if the order can be filled from another owner. You can think of it this way: the subquery returns a set of Items from which each ItemDesired in the Orders table is compared; the In condition is true only if the ItemDesired is in that returned set from the Antiques table.

Also notice, that in this case, that there happened to be an antique available for each one desired...obviously, that won't always be the case. In addition, notice that when the IN, "= ANY", or " = SOME" is used, that these keywords refer to any possible row matches, not column matches...that is, you cannot put multiple columns in the subquery Select clause, in an attempt to match the column in the outer Where clause to one of multiple possible column values in the subquery; only one column can be listed in the subquery, and the possible match comes from multiple row values in that one column, not vice-versa.

For now, that's enough on the topic of complex SELECT queries. Now let us dwell on to the other SQL statements.

4.14.14 Miscellaneous SQL Statements

Aggregate Functions

We will discuss five important aggregate functions: SUM, AVG, MAX, MIN, and COUNT They are called aggregate functions because they summarize the results of a query, rather than listing all of the rows.

SUM () gives the total of all the rows, satisfying any conditions, of the given column, where the given column is numeric.

- **AVG** () gives the average of the given column.
- **MAX** () gives the largest Fig.ure in the given column.
- **IN** () gives the smallest Fig.ure in the given column.
- **COUNT(*)** gives the number of rows satisfying the conditions.

Looking at the tables at the top of the document, let's look at three examples:

```
SELECT SUM(SALARY), AVG(SALARY)
FROM EMPLOYEESTATISTICSTABLE;
```

This query shows the total of all salaries in the table, and the average salary of all of the entries in the table.

```
SELECT MIN(BENEFITS)
FROM EMPLOYEESTATISTICSTABLE
WHERE POSITION = 'Manager';
```

This query gives the smallest Fig.ure of the Benefits column, of the employees who are Managers, which is 12500.

```
SELECT COUNT(*)
FROM EMPLOYEESTATISTICSTABLE
WHERE POSITION = 'Staff';
```

This query tells you how many employees have Staff status (3).

4.14.15 Inserting into Database

```
mySQL = "INSERT INTO table_name (fldUserID, fldProductName,_
fldApproval, fldDate) VALUES (" & _3 & ",'" & _
Request("fldProductName") & "'," & _
True & ",#" & _
Date() & "#)" s
```

RESULT: Insert data into table with user input. Note the different data type: fldUserID is Number, fldProductName is Text, fldApproval is Yes/No, and fldDate is Date/Time.

4.14.16 Deleting data from Database

```
mySQL = "DELETE FROM table_name WHERE fldID = 123"
```

Result: Delete record(s) where fldID is 123. Without WHERE clause, all records will be deleted!

4.14.17 Updating data in Database

```
mySQL = "UPDATE table_name SET fldProductName = 'NewProd', _
fldApproval = False, _fldDate = #" & Date() & "# WHERE fldID = 3"
```

Result: Update record where fldID is 3. Without WHERE clause, all records will be updated!

4.14.18 With Fields

```
mySQL = "ALTER TABLE table_name ADD fldAuto Counter"
```

Result: Add a new field called fldAuto with the property of AutoNumber

```
mySQL = "ALTER TABLE table_name ADD fldID Number NOT NULL"
```

Result: Add a new field called fldID with the property of Number. Empty entry is not allowed.

```
mySQL = "ALTER TABLE table_name ADD fldFirstName char(50) NULL"
```

Result: Add a new field called fldFirstName with the property of Text (Length 50). Empty is entry allowed.

```
mySQL = "ALTER TABLE table_name ADD fldRemarks Memo NULL"
```

Result: Add a new field called fldRemarks with the property of Memo. Empty entry allowed.

```
mySQL = "ALTER TABLE table_name ADD fldDate Date"
```

Result: Add a new field called fldDate with the property of Date/Time.

```
mySQL = "ALTER TABLE table_name ADD fldApproval YesNo"
```

Result: Add a new field called fldApproval with the property of Yes/No.

```
mySQL = "ALTER TABLE table_name DROP COLUMN fldFirstName"
```

Result: Delete column "fldFirstName" and all data in it from a table.

4.14.19 With Tables

```
mySQL = "CREATE TABLE tblHuiYang (fldAuto Counter)"
```

Result: Create a new table called tblHuiYang with a field called fldAuto (AutoNumber)

```
mySQL = "DROP TABLE tblHuiYang"
```

Result: Delete an existing table - tblHuiYang from database

This SQL Statement very briefly demonstrates Joins between multiple table. It's extremely useful and can significantly speed up processing time by join multiple SQL statements into one.

Find the customer's name who ordered OrderId "12345"

```
SQL = "SELECT C.Name FROM Customer C, Sale S WHERE C.CustomerID =
S.CustomerID AND S.OrderID = '12345'"
```

What's the Unit Price for OrderID "12345"

```
SQL = "SELECT P.UnitPrice FROM Product P, Sale S WHERE P.ProductID =
S.ProductID AND S.OrderID = '12345'"
```

How many orders for SONY from 4/1/00 to 4/7/00?

```
SQL = "SELECT Count(S.OrderID) AS Total FROM Product P, Sale S WHERE
P.ProductID = S.ProductID AND S.Date BETWEEN '4/1/00' AND '4/7/00' AND
P.Manufacturer = 'SONY'"
Set rs = con.Execute(SQL)
response.write rs("Total")
```

List all cutomers' name who live in CA and ordered SONY for the previous week

```
SQL = "SELECT C.Name FROM Product P, Customer C, Sale S WHERE
P.ProductID = S.ProductID AND C.CusomterID = S.CustomerID AND
P.Manufacturer = 'SONY' AND C.State = 'CA' AND S.Date >= '" & Date - 7
& "'"
Set rs = con.Execute(SQL)
While Not rs.EOF
response.write rs("Name") & vbCrLf
rs.MoveNext
Wend
```

4.15 SQL*Plus User Guide

4.15.1 Starting SQL*Plus

When starting SQL*Plus for the first time, you will have to perform the following steps:

Add the following lines to your .cshrc file that is in your home directory:

* set path = ($path /usr/local/ucc/ssru1/oracle)
* source /usr/local/ucc/ssru1/oracle/oracle/oraenv1
* Initialize your environment to run Oracle by typing:

```
source ~/.cshrc
```

From this point on your environment initialization will be handled automatically whenever a c shell is opened.

To start SQL*Plus, type **sqlplus** at the unix command prompt on the SGI server. On startup, SQL*Plus displays the following:

```
Enter user-name:
```

when requested for a user-name just press return. If you run into problems, use your user-id and user-password for the requested user-name and password.

Note: Because of the limited number of licences available, access to Oracle has been restricted only to those required to use it. If you are having difficulties starting up SQL*Plus it could be that you have not been granted access to it. In this case please bring it to the attention of your instructor.

Having logged on, you should now have the Oracle prompt:

SQL> which will be refered to as the prompt throughout this guide. From this point you can interactively enter SQL and SQL*Plus statements.

The EXIT and QUIT commands allow you to end your SQL*Plus session.

4.15.2 Entering Statements and Commands

Both SQL and SQL*Plus statements are entered by typing after the prompt. e.g. SQL> DESCRIBE SPJ.

This command will ask for the description of the columns of the SPJ (Supplier-Parts-Jobs) table in the database. e.g. SQL> SELECT * FROM SPJ;

The command will list all of the values in the SPJ table.

Note: all SQL commands must be terminated with a semi-colon.

4.15.3 Repeated Execution of Statements

The command buffer holds the most recently executed SQL statement. This statement can be re-executed by typing either RUN or /, the difference being RUN will first display and then execute the statement. Typing LIST will display the current SQL statement that is currently held in the buffer.

Note: SQL*Plus commands are not held in the buffer.

4.15.4 Correcting Mistakes

In multi-line statements, moving to a particular line (refered to as the current line) is done by entering the line number at the prompt. Oracle automatically displays the desired line.

Corrections can be made using the CHANGE, INPUT and DEL commands. The CHANGE command replaces one string with another string.

Example:

```
SQL> LIST
1* SELECT * SPJ
SQL> CHANGE /*/* FROM/
1* SELECT * FROM SPJ
```

Oracle will automatically display the effect of this change.

The INPUT command is used to add new lines to the statement after the current line. Multiple-lines can be entered with a <CR> after each line and a blank line to finish inputing. Conversely, the DEL command is used to delete the current line.

Example:

```
SQL> LIST
1 SELECT
2 *
3 FROM
4* SPJ
SQL> 2
2* *
SQL> DEL
1 SELECT
2 FROM
3* SPJ
SQL> 1
1* SELECT
SQL> INPUT S#
SQL> LIST
1 SELECT
2 S#
3 FROM
4* SPJ
```

Once corrections have been made, the statement can be re-executed using the RUN command.

4.15.5 Storing Statements

Any statement that is in the buffer can be saved into a file using the SAVE command.

```
e.g. SQL> SAVE SELS#.SQL
```

If the file already exists, you can overwrite the existing file by adding REPLACE to the end of the SAVE command.

4.15.6 Retrieving and Executing Stored Statements

The GET command will place the stored statement into the command buffer. The RUN command can then be used to execute the statement. You can also fetch and process the stored statement in one step using either the START or the @ commands.

```
e.g. SQL> START SELS#.SQL
or
SQL> @ SELS#.SQL
```

4.15.7 The SET command

SQL*Plus supports the SET command with which the user can define their own working environment. A number of important SET commands are given, a complete listing can be found by using the HELP SET command.

List of useful SET features:

PAGESIZE {14|n} : sets the page size where n = # of lines

PAUSE {OFF|ON|text} : controls scrolling on the terminal when running reports

SPACE {1|n} : sets the number of spaces between columns

LINESIZE {80|n} : sets the character width of a line

4.15.8 The LOGIN.SQL file

When SQL*Plus is started, this file is read and all statements and commands in it are executed. This is especially useful if the working environment that you prefer is different from that of the default working environnment. By placing all of the SET commands in the login.sql file, all of your evironment changes will be in effect when you start up SQL*Plus.

- The comparison operator
- Coupling conditions with AND, OR and NOT;
- The BETWEEN operator;
- The IN operator;
- The LIKE operator;
- The NULL operator;
- The IN operator with a subquery;
- The comparison operator with a subquery;
- The ANY and ALL operators; and
- The EXISTS operator.

The SELECT clause and functions

- Expressions in the SELECT clause;
- Removing duplicate rows with DISTINCT;
- Row numbers;
- Statistical functions; and
- General rules for using statistical functions.

The GROUP BY and HAVING clauses

- Grouping on one column;
- Grouping on two or more columns; and
- General rule for the HAVING clases.

The ORDER BY clause

- Sorting on one column;
- Sorting with sequence numbers;
- Sorting on more than one column; and
- Sorting of NULL values;

4.15.9 The SELECT Statement

The select statment is the most important and most often used statement in SQL. A SELECT statement is composed of a number of clauses, as shown in the following BNF definition.

```
<select statement> ::= <select clause> <from clause> [ <where clause> ]
[ <connect by clause> ] [ <group by clause> ] [ <having clause> ] [
<order by clause> ]
```

when formulating SELECT statements, it is important to follow the following rules:

- Each SELECT statement must have the SELECT and FROM clauses. All other clauses are optional.
- The order of the clauses is fixed and must be given in the order in which they are listed in the above definition.
- A HAVING clause can only be used if there is a GROUP BY clause.

4.15.10 Common Elements of SELECT

In this section, the most commonly used elements of the SELECT statement are outlined.

Literal (constant)

Oracle supports the following literals:

- integer literal
- decimal literal
- floating point literal
- alphanumeric literal

An integer literal is a whole number without a decimal point and may be preceeded by a plus or a minus.

A decimal literal is a number with or without a decimal point and may be preceeded by a plus or minus. By definition, every integer literal is also a decimal literal.

A floating point literal is a decimal literal that may be followed by an exponent.

An alphanumeric literal is a string of zero or more alphanumeric characters enclosed in single quotation marks. The following are characters are permitted in an alphanumeric literal:

- all lower case letters (a-z);
- all upper case letters (A-Z);
- all digits (0-9); and
- all remaining characters (such as ' # ~ _ and <).

Note: Single quotation marks within literals are represented by 2 single quotation marks to separate it from the quotation marks used to enclose the literal.

e.g. Alphanumeric literal value

```
'Jones'      Jones
'it''s'      it's
''
  ''''  '
```

4.15.11 Numeric Expressions

A numeric expression is an arithmetic expression with either an integer, decimal or floating point value.

Numeric literals and columns with numeric data types can be used with the four basic operators (+, -, *, /), if the required brackets can be used. The following points outline the use of numeric expressions:

- The value of an expression that contains a NULL value is by definition equal to NULL.
- Expression evaluation is performed in the following manner:
  - left to right,
  - brackets,
  - multiplication and division,
  - addition and subtraction.
- The data type of an expression is taken as the most precise data type occuring in the expression.
- Column specifications that occur in a numeric expression must have a numeric data type.
- An alphanumeric expression can be a numeric expression provided the value of the alphanumeric expression can be converted to a numeric value.

4.15.12 Alphanumeric Expressions

Alphanumeric expressions are similar to numeric expressions in that they have an alphanumeric value that is typically CHAR or VARCHAR. The || operator is the only operator defined for alphanumeric expressions and is used to concatinate two alphanumeric expressions.

4.15.13 Scalar Functions

Scalar functions are used to perform various calculations. A scalar function may have zero or more parameters and returns a value that is dependent upon the values of the parameters.

e.g.

ABS(-25.4) → 25.4 - the absolute value funtion.

4.15.14 Statistical Functions

Statistical functions are similar to scalar functions in that they both perform calculations, however there are two major diffences. Statistical function always have one parameter and the value of that parameter consists of a set of elements as opposed to one element for a scalar function parameter.

4.15.15 The FROM Clause

The FROM clause is used to specify which tables are to be queried. The syntax for the FROM clause is given below.

```
<from clause> ::=
FROM <table reference> [ {,<table reference> ]... ]
<table reference> ::=
<table specification> [ <pseudonym> ]
<table specification> ::= [ <user> . ] <table name>
```

4.15.16 Specifying Tables

If you wish to refer to a table created by someone else, you must specify the name of the owner before the name of the table. e.g. the user JONES wishes to use a table created by SMITH. The SELECT statement would be as follows:

```
SELECT *
FROM SMITH.ORDERS
```

4.15.17 Defining Views

(a) Specifying Columns

When you specify columns, you must identify from which table the column belongs. For example, to obtain all of the part numbers from the ORDERS table (assuming SMITH owns the table), we could use any of the following three SELECT statements:

```
SELECT P#
FROM ORDERS
SELECT ORDERS.P#
FROM ORDERS
SELECT SMITH.ORDERS.P#
FROM SMITH.ORDERS
```

If data from more than one table is required, all of the required tables must be named in the FROM clause.

(b) Specifying Column Headings

At times it is convenient to use an alternative name for a column. An alternative column heading may be specified behind any expression in the SELECT clause. SQL*Plus will place the column heading on top of the result instead of the expression itself.

Example:

```
SELECT SUPPNO id, NAME supplier
FROM SUPPLIERS
```

Where id and supplier are the column headings for SUPPNO and NAME respectively.

4.15.18 Pseudonyms

When multiple table specifications appear in the FROM clause, it is sometimes easier to use pseudonyms (aliases). Pseudonyms are specified in the FROM clause after each table specification eg.

```
SELECT S.S#
```

```
FROM SUPPLIER S, ORDERS O
WHERE S.J# = O.J#
```

Because oracle processes the FROM clause first, the use of pseudonyms before being declared is permitted.

Note: In instances where a table is re-ferred to more than once, the use of pseudonyms is mandatory.

4.15.19 The WHERE Clause

The WHERE clause is used to specify the rows required by a query through the use of a condition or series of conditions. Oracle individually processes all of the rows that are found in the intermediate table of the FROM clause. If a particular row satisfies the condition, the concerned row is passed to the intermediate result table for the WHERE clause.

All of the possible conditions that can be used in the WHERE clause are described below.

4.15.20 The Comparison Operator

The comparison of two expressions is the simplest condition. The value of a condition is either true, false or unknown depending on the operator and the value of the expression. The comparison operators provided by Oracle are:

Comparison operator	Meaning
=	equal to
<	less than
>	greater than
<=	ess than or equal to
>=	greater than or equal to
<>	or != or ^ = not equal to

In a comparison, it is possible to compare multiple values simultaneously. These combination of expressions are called row expressions. For example, the following SELECT statement returns all supplier numbers that supply part P3 to job J4. e.g.

```
SELECT S#
FROM SPJ
WHERE (P#, J#) = (P3, J4)
```

Note: Only the 'equal to' and 'not equal to' operators can be used when row expressions are compared.

4.15.21 Coupling conditions with AND, OR and NOT

A WHERE clause may contain multiple conditions if AND, OR and NOT operators are used. Oracle processes ANDs before ORs unless brackets are used to control the evaluation order. The truth table below contains all of the possible values with AND, OR and NOT for two conditions C1 and C2:

C1	C2	C1 AND C2	C1 OR C2	NOT C1
true	true	true	true	false
true	false	false	true	false
true	unknown	unknown	true	false
false	true	false	true	true
false	false	false	false	true
false	unknown	false	unknown	true
unknown	true	unknown	true	unknown
unknown	false	false	unknown	unknown
unknown	unknown	unknown	unknown	unknown

4.15.22 The BETWEEN Operator

The BETWEEN operator is a special operator which determines whether a value occurs within a given range of values.

Ex. Find supplier numbers that have status values between 20 and 30.

```
SELECT S#
FROM S
WHERE STATUS BETWEEN 20 AND 30
```

If, E1, E2 and E3 are expressions, then:

```
E1 BETWEEN E2 AND E3
```

is equivalent to:

```
(E1 >= E2) AND (E1 <= E3)
```

Clearly, E2 <= E3 for the BETWEEN operator to behave correctly.

4.15.23 The IN Operator

When you are required to determine whether a value appears in a given set of values, it can become rather combersome using OR operators. As a solution, Oracle provides the IN operator to simplify the problem.

Ex. Find all job numbers for jobs in London, Paris and New York.

```
SELECT J#
FROM J
WHERE CITY = 'London'
OR CITY = 'Paris'
OR CITY = 'New York'
```

While this statement is correct, clearly as the set of values inceases, the more long-winded the statement becomes. The IN operator can be used to simplify the statement as follows:

```
SELECT J#
FROM J
WHERE CITY IN ('London', 'Paris', 'New York')
```

In addition, the IN operator can be used where row expressions are compared with each other.

Ex. Get supplier numbers for suppliers who supply P2 to J4 or P3 to J2 or P5 to J1.

```
SELECT S#
FROM SPJ
WHERE (P#, J#) IN
(('P2','J4'),
('P3','J2'),
('P5','J1'))
```

4.15.24 The LIKE Operator

The LIKE operator is used to select alphanumeric values based on a particular pattern mask. In the pattern mask the percent sign (%) stands for zero, one or more characters while the underscore (_) stands for exactly one character.

Ex. Get the supplier number of suppliers whose name begins with the capital letter T.

```
SELECT S#
FROM S
WHERE NAME LIKE 'B%'
```

Ex. Get the supplier number for suppliers whose name begins with the captial letter S and has a small e as the penultimate letter.

```
SELECT S#
FROM S
WHERE NAME LIKE 'S%e_'
```

4.15.25 The NULL operator

The NULL operator is used to select rows that have no value is a particular column.

Ex. Find part numbers for parts for which the part colour is unknown.

```
SELECT P#
FROM P
WHERE COLOUR IS NULL
```

Note: when using the NULL operator the IS cannot be replaced by an equals sign.

4.15.26 The IN Operator with Subquery

While in the previous section the values for expressions for the IN operator were listed explicitly. However, the IN operator can also take on another form whereby Oracle determines the value of the literals at the point when the statement is processed.

Ex. We want to get the part numbers for red parts that are used in at one or more jobs.

As covered so far, to do this we would have to execute two queries however, we can use a subquery to generate the needed values:

```
SELECT P#
FROM SPJ
WHERE P# IN
(SELECT P#
FROM P
WHERE COLOUR = 'red')
```

The difference between the use of the IN operator with a set of expressions as opposed to with a subquery is that in the first case the set of values is fixed by the user, and in the second case the the values are not known until they are determined by Oracle during the processing of the subquery.

4.15.27 The Comparison Operator with a Subquery

In addition to being used after the IN operator, subqueries may also be used after any of the comparison operators. It is important to note that comparison operators are only valid if the subquery returns one value at that precise point in time. There are two reasons for using the comparison operators as opposed to the IN operator when a subquery returns one value.

By using a comparison operator you signal that the query always has one value. If the subquery returns more than one value, then either the contents of database are not correct or the database structure is not as you expected. In both cases the comparison operator functions as a means of control.

By using a comparison operator you give Oracle information about the expected number of values to be returned by the subquery, namely one. This information allows Oracle to decide upon the most appropriate processing strategy.

4.15.28 The ANY and ALL Operators

Another way of using the subquery is through the use of the ALL and ANY operators. The ALL and ANY operators are very similar to the IN operator in that they apply a comparison operator to a set of values. The ALL operator evaluates to TRUE if the result of the comparison is TRUE for each element of the set. The ANY operator on the other hand evaluates to TRUE if the there is at least one element in the set that evaluates to TRUE.

Ex. Give the supplier number and part number for parts that have the highest quantity on order.

```
SELECT SUPPNO, PARTNO
FROM QUOT
WHERE QONORDER >= ALL
(SELECT QONORDER
FROM QUOT)
```

Ex. Give the supplier number and part number for parts that do not have the longest delivery time.

```
SELECT SUPPNO, PARTNO
FROM QUOT
WHERE DELIVERY_TIME < ANY
(SELECT DELIVERY_TIME
FROM QUOT)
```

4.15.29 The EXISTS Operator

The EXISTS operator evaluates whether a subquery returns a row or not.

Ex. List the names of suppliers that have parts on order.

```
SELECT NAME
FROM SUPPLIERS
WHERE EXISTS
```

```
(SELECT *
FROM QUOT
WHERE SUPPNO = SUPPLIERS.SUPPNO
AND QONORDER > 0)
```

When using the EXISTS operator a correlated subquery is required to establish a relationship with the main query. This is accomplished through the use of column specification.

```
i.e. SUPPNO = SUPPLIERS.SUPPNO
```

But how does the statement acctually work? For every supplier, number in the SUPPLIERS table separately Oracle executes the subquery. If at least one row is returned the row condition is satisfied hence the supplier name for that particular row in the SUPPLIER table would be returned by the main query.

It is important to remember that when using the EXISTS operator, Oracle looks to see if the result of the subquery returns rows. Oracle does not look at the contents of the row. Hence, what is specified in the SELECT clause is totally irrelevant.

4.15.30 The SELECT Clause and Functions

When using the WHERE clause the intermediate result is a horizontal subset of the table. In contrast, the SELECT clause selects only columns, the result forming a vertical subset of a table. This section will examine the manipulation of columns using the SELECT clause.

4.15.31 Expressions in the SELECT Clause

Because Oracle evaluates the intermediate result row by row, each column expresssion may give rise to a value in the result row. So far only column names or an asterisk (*) - which returns all columns, have be used in the SELECT clause. In addition, the SELECT clause may also consist of expressions that may take the form of a literal, a calculation or a scalar function.

Ex. List the supplier number, part number and total cost of parts on order for each row of the QUOT table.

```
SELECT SUPPNO, PARTNO, PRICE * QONORDER
FROM QUOT
```

4.15.32 Removing Duplicate Rows with DISTINCT

When a SELECT clause is written with one or more column expressions preceeded by the word DISTINCT, Oracle removes all duplicate rows from the intermediate result.

Ex. List the cities that have suppliers.

```
SELECT DISTINCT CITY
FROM S
```

DISTINCT is concerned with the whole row and not only with the column expression that directly follows the word DISTINCT in the SELECT clause. In two cases, the use of DISTINCT has no effect on the result of a query.

When the SELECT clause includes at least one candidate key for each table specified in the FROM clause, the DISTINCT has no effect. Any table that has a candidate key is guaranteed to have no duplicates, hence the inclusion of a candidate key in the SELECT clause guarantees that there will be no duplicates in the final result.

When the SELECT clause results in one row with values, DISTINCT is clearly useless.

Row numbers

Row numbers can be displayed through the use of the system variable ROWNUM.

```
SELECT ROWNUM, *
FROM SPJ
```

4.15.33 Statistical Functions

Expressions in the SELECT clause may contain statistical functions. If a statistical function is used, the SELECT statement will yield only one row as a result (note: it is assumed that the SELECT statement has no GROUP BY clause.)

The following statistical functions are available in Oracle:

Function	Meaning
COUNT()	Counts the number of values in a column or the number of rows in a table
MIN()	Determines the smallest value in a column
MAX()	Determines the largest value in a column
SUM()	Determines the sum of values in a column
AVG()	Determines the weighted arithmetic mean of the values in a column
STDDEV()	Determines the standard deviation of the values in a column
VARIANCE()	Determines the variance of the values in a column

As opposed to COUNT, MIN and MAX, the functions SUM, AVG, VARIANCE and STDDEV are only applicable to columns and expressions with numeric data types. Because of the possibilities of duplicate values, the use of ALL and DISTINCT within these functions can lead to different results. Suppose we have the following table:

```
TABLE: VAL
10
10
20
30
30
SELECT SUM(VAL)
FROM TABLE
result = 100
SELECT SUM(DISTINCT VAL)
FROM TABLE
result = 60
```

From this example, it is clear that care must be taken when using these statistical functions.

4.15.34 General Rules for Using Statistical Functions

Oracle does not include NULL values in the calculation of functions. This may cause incorrect results to be returned. As a result, the user must make the appropriate modifications to the expression to ensure a correct result.

Two principle rules that must be followed when using statistical functions are:

- If a SELECT statement has no GROUP BY clause, and if the SELECT clause has one or more statistical functions, any column name specified in that SELECT clause must occur within a statistical function.
- If a SELECT statement does have a GROUP BY clause, any column name specified in the SELECT clause must occur within a statistical function or in the list of columns given in the GROUP BY clause or in both.

4.15.35 The GROUP BY and HAVING Clauses

The GROUP BY clause groups rows on the basis of similarities between the rows. The HAVING clause is similar to the WHERE clause in that it enables conditions to be applied to groups. Clearly, the HAVING clause can only be applied in conjunction with a GROUP BY clause.

Grouping on one column

The simplest form of the GROUP BY clause is where only one column is grouped.

Ex. Give all of the cities from the S table.

```
SELECT CITY
FROM S
GROUP BY CITY
```

When using the GROUP BY clause, it is important to understand the underlying workings of the GROUP BY clause. If we could view the intermediate result of the previous example, it would appear like this:

S#	SNAME	STATUS	CITY
S1, S4	Smith, Clark	20, 20	London
S2, S3	Jones, Blake	10, 30	Paris
S5	Adams	30	Athens

The columns are illustrated this way for illustrative purposes. When the GROUP BY clause is used all rows having the same city form a group. Each row in the intermediate result will have one value in the CITY column while the all other columns may contain multiple values. While it is not possible to actually view the intermediate result of a GROUP BY clause, knowing the underlying structure of the intermediate result is important when using statistical functions.

Ex. How many suppliers are there in each city.

```
SELECT TOWN, COUNT(*)
FROM S
GROUP BY TOWN
```

In this example the COUNT function is executed against each grouped row instead of against all rows.

Note: In principle, any statistical function can be used in a SELECT clause as long as the function operates on a column that is not grouped.

4.15.36 Grouping on Two or more Columns

A GROUP BY clause may also contain two or more columns.

Ex. Give the number of suppliers supply the same part to the same job.

```
SELECT COUNT(*), P#, J#
FROM SPJ
GROUP BY P#, J#
```

The HAVING clause is used in conjunction with the GROUP BY clause as a means of selecting groups on the basis of their particular group properties. While a condition in the having clause looks a lot like a 'normal' condition in a WHERE clause, conditions in the HAVING clause may contain statistical functions.

Ex. List all CITYS that have more than one supplier.

```
SELECT CITY
FROM S
GROUP BY CITY
HAVING COUNT(*) > 1
```

In the HAVING, condition we specified the selection of groups where the number of rows exceeds one. It is important to remember that the value of a statistical function in a HAVING clause is calculated for each group separately.

4.15.37 General Rule for the HAVING Clause

Just as there are a number of rules for the use of columns and statistical fuctions in SELECT clauses, the HAVING clause requires a similar type of rule, as follows:

- Each column specification specified in the HAVING clause must occur within a statistical function or must occur in the list of columns named in the GROUP BY clause.
- The ORDER BY clause.
- When a SELECT statement has no ORDER BY clause the sequence of the final result is unpredictable. Using an ORDER BY clause at the end of a SELECT statement is the only guarantee that the resulting rows will be sorted in some particular manner.

4.15.38 Sorting on one Column

The simplest form of sorting is on one column.

Ex. Find all supplier numbers and names; sort the result in order of supplier name.

```
SELECT S#, SNAME
FROM S
ORDER BY SNAME
```

Note: By default, Oracle will sort the result in ascending order. If the result is to be sorted in descending order, DESC must be specified. Similarily, the ascending order can be explicitly stated using

```
ASC.
Ex.
...
GROUP BY STATUS DESC
```

4.15.39 Sorting with Sequence Numbers

Oracle assigns a sequence number to each expression in the SELECT clause. In the GROUP BY clause expressions may be replaced by their respective sequence numbers. The following statement is equivilent to the previous example:

```
SELECT S#, SNAME
FROM S
ORDER BY 2
```

In this statement, the sequence number 2 represents the second expression in the SELECT clause namely, SNAME.

Note: It is essential to use sequence numbers when an expression consists of a function, a literal or a numeric expression as these expressions are not permitted explicitly within the GROUP BY clause.

4.15.40 Sorting on More than One Column

Multiple columns (or expressions) may also be specified in an ORDER BY clause.

Ex. list all supplier names, number and status; group the result by status and within that by name.

```
SELECT S#, NAME, STATUS
FROM S
GROUP BY STATUS, NAME
```

Sorting of NULL values

Oracle recognizes NULL values as the highest values in a column, hence they are always placed at the bottom of the result if the order is ascending and at the top if the order is descending.

4.15.41 Combining SELECT Statements

1. Introduction

A number of operators are provided to combine the results of individual SELECT statements. These operators are referred to as set operators and are an extension of the functionality of the SELECT statement. The set operators recognized in Oracle are: UNION, INTERSECT and MINUS.

2. Linking with UNION

When two select blocks are combined using the UNION operator, the result consists of the resulting rows of either or both of the select blocks. The UNION operator is equivalent to set union.

Ex. Give the supplier names for suppliers in London and Paris.

```
SELECT SNAME
FROM S
WHERE CITY = 'London'
UNION
SELECT SNAME
FROM S
WHERE CITY = 'Paris'
```

In this example, each of the select blocks return a table with one column and zero or more rows. The UNION operator essentially places the two tables under one another creating one table.

Note: When the UNION operator is used Oracle automatically removes all duplicate rows from the final result.

3. Rules for using UNION

The following rules for using the UNION operator must be observed:

- The SELECT clauses of all relevant select blocks must have the same number of expressions.
- Expressions which will be combined (or placed under one another) in the final result must be of comparable data types.
- An ORDER BY clause may only be specified after the last select block. The ordering is performed on the entire final result, only after all intermediate results have been combined.

4. Linking with INTERSECT

When two select blocks are combined with the INTERSECT operator, the final result consists of those rows which appear in the results of both of the select blocks. INTERSECT is equivalent to set intersection.

Ex. Give the supplier number and name for suppliers in London with status > 15.

```
SELECT S#, SNAME
FROM S
WHERE CITY = 'London'
INTERSECT
SELECT S#, SNAME
FROM S
WHERE STATUS > 15
```

In this example the INTERSECT operator is looking for the rows appearing in both tables, giving the final result as one table.

5. Linking with MINUS

When two select blocks are combined using the MINUS operator, the final result consists of those rows appearing in the result of the first block which do not appear in the result of the second block. The MINUS operator is equivalent to set difference.

Ex. Give the supplier numbers and names for suppliers in London that do not have status = 30.

```
SELECT S#, SNAME
FROM S
WHERE CITY = 'London'
MINUS
SELECT S#, SNAME
FROM S
WHERE STATUS = 30
```

6. Set operators and NULL values

While NULL values are not considered to be equivalent, this is not true when set operators are used. Duplicate rows that contain NULL values will be displayed as only one row, as Oracle considers NULL values to be equivalent when set operators are processed. This is in accordance with the relational model as defined by Codd (1990).

7. Combining multiple-set operators

No restriction is made on the number of set operations that can be used within a single SELECT statement. By default, set operators are applied in order from left to right. Explicity control of the execution may be handled through the use of round brackets around the select blocks.

4.16 Basics of an SQL Query

As we have already alluded to, a "query" is a structured request to the database for data. At its core, a query is something like,

"Hey, give me a list of all the clients in the CLIENTS table who live in the 213 area code!"

Or, in more specific terms, a query is a simple statement (like a sentence) which requests data from the database. Much as is the case with English, an SQL statement is made up of subjects, verbs, clauses, and predicates.

Let us take a look at the statement made above. In this case, the subject is "hey you database thing". The verb is "give me a list". The clause is "from the CLIENTS table". Finally, the predicate is "who live in the 213 area code."

We will explain the code later, but let me show you what the above statement might look like in SQL:

```
SELECT * FROM CLIENTS WHERE area_code = 213
SELECT = VERB = give me a list
FROM CLIENTS = CLAUSE = from the CLIENTS table
area_code = 213 = PREDICATE = who live in the 213 area code
```

4.16.1 Views

In SQL, you might (check your DBA) have access to create views for yourself. What a view does is to allow you to assign the results of a query to a new, personal table, that you can use in other queries, where this new table is given the view name in your FROM clause. When you access a view, the query that is defined in your view creation statement is performed (generally), and the results of that query look just like another table in the query that you wrote invoking the view. For example, to create a view:

```
CREATE VIEW ANTVIEW AS SELECT ITEM DESIRED FROM ORDERS;
```

Now, write a query using this view as a table, where the table is just a listing of all Items Desired from the Orders table:

```
SELECT SELLERID
FROM ANTIQUES, ANTVIEW
WHERE ITEMDESIRED = ITEM;
```

This query shows all SellerID's from the Antiques table where the Item in that table happens to appear in the Antview view, which is just all of the Items Desired in the Orders table. The listing is generated by going through the Antique Items one-by-one until there's a match with the Antview view. Views can be used to restrict database access, as well as, in this case, simplify a complex query.

4.16.2 Creating New Tables

All tables within a database must be created at some point in time. let us see how we would create the Orders table:

```
CREATE TABLE ORDERS
(OWNERID INTEGER NOT NULL,
ITEMDESIRED CHAR(40) NOT NULL);
```

This statement gives the table name and tells the DBMS about each column in the table. Please note that this statement uses generic data types, and that the data types might be different, depending on what DBMS you are using. As usual, check local listings. Some common generic data types are:

- **Char(x)** - A column of characters, where x is a number designating the maximum number of characters allowed (maximum length) in the column.
- **Integer** - A column of whole numbers, positive or negative.
- **Decimal(x, y)** - A column of decimal numbers, where x is the maximum length in digits of the decimal numbers in this column, and y is the maximum number of digits allowed after the decimal point. The maximum (4,2) number would be 99.99.
- **Date** - A date column in a DBMS-specific format.
- **Logical** - A column that can hold only two values: TRUE or FALSE.

One other note, the NOT NULL means that the column must have a value in each row. If NULL was used, that column may be left empty in a given row.

4.16.3 Inserting into Database

```
mySQL = "INSERT INTO table_name (fldUserID, fldProductName,_
fldApproval, fldDate) VALUES (" & _3 & ",'" & _
Request("fldProductName") & "'," & _
True & ",#" & _
Date() & "#)"
```

Result: Insert data into table with user input. Note the different data type: fldUserID is Number, fldProductName is Text, fldApproval is Yes/No, and fldDate is Date/Time.

4.16.4 Deleting data from Database

```
mySQL = "DELETE FROM table_name WHERE fldID = 123"
```

Result: Delete record(s) where fldID is 123. Without the WHERE clause, all records will be deleted!

4.16.5 Updating data in Database

```
mySQL = "UPDATE table_name SET fldProductName = 'NewProd',_
fldApproval = False, _
fldDate = #" & Date() & "# WHERE fldID = 3"
```

Result: Update record where fldID is 3. Without WHERE clause, all records will be updated!

4.16.6 With Fields

```
mySQL = "ALTER TABLE table_name ADD fldAuto Counter"
```

Result: Add a new field called fldAuto with the property of AutoNumber

```
mySQL = "ALTER TABLE table_name ADD fldID Number NOT NULL"
```

Result: Add a new field called fldID with the property of Number. Empty entry is not allowed.

```
mySQL = "ALTER TABLE table_name ADD fldFirstName char(50) NULL"
```

Result: Add a new field called fldFirstName with the property of Text (Length 50). Empty entry is allowed.

```
mySQL = "ALTER TABLE table_name ADD fldRemarks Memo NULL"
```

Result: Add a new field called fldRemarks with the property of Memo. Empty entry is allowed.

```
mySQL = "ALTER TABLE table_name ADD fldDate Date"
```

Result: Add a new field called fldDate.

4.16.7 Date/Time

```
mySQL = "ALTER TABLE table_name ADD fldApproval YesNo"
```

Result: Add a new field called fldApproval with the property of Yes/No

```
mySQL = "ALTER TABLE table_name DROP COLUMN fldFirstName"
```

Result: Delete column "fldFirstName" and all data in it from a table.

4.16.8 With Tables

```
mySQL = "CREATE TABLE tblHuiYang (fldAuto Counter)"
```

Result: Create a new table called tblHuiYang with a field called fldAuto (AutoNumber)

```
mySQL = "DROP TABLE tblHuiYang"
```

Result: Delete an existing table - tblHuiYang from the database

This SQL Statement Tutorial very briefly demonstrates the Joins between multiple table. It is extremely useful and can significantly speed up processing time by join multiple SQL statements into one. Please note that this demo is using the SQL Server.

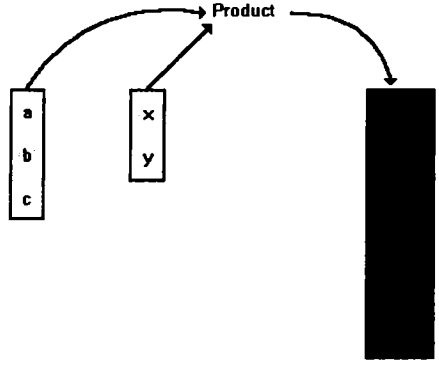


Fig. 4.10

Find the customer's name who ordered OrderId "12345"

```
SQL = "SELECT C.Name FROM Customer C, Sale S WHERE C.CustomerID =
S.CustomerID AND S.OrderID = '12345'"
```

What's the Unit Price for OrderID "12345"

```
SQL = "SELECT P.UnitPrice FROM Product P, Sale S WHERE P.ProductID =
S.ProductID AND S.OrderID = '12345'"
```

How many orders for SONY from 4/1/00 to 4/7/00?

```
SQL = "SELECT Count(S.OrderID) AS Total FROM Product P, Sale S WHERE
P.ProductID = S.ProductID AND S.Date BETWEEN '4/1/00' AND '4/7/00' AND
P.Manufacturer = 'SONY'"
Set rs = con.Execute(SQL)
response.write rs("Total")
```

List all cutomers' name who live in CA and ordered SONY for the previous week

```
SQL = "SELECT C.Name FROM Product P, Customer C, Sale S WHERE
P.ProductID = S.ProductID AND C.CusomterID = S.CustomerID AND
P.Manufacturer = 'SONY' AND C.State = 'CA' AND S.Date >= '" & Date - 7
& "'"
Set rs = con.Execute(SQL)
While Not rs.EOF
response.write rs("Name") & vbCrLf
rs.MoveNext
Wend
```

4.16.9 Altering Tables

Let us add a column to the Antiques table to allow the entry of the price of a given Item:

```
ALTER TABLE ANTIQUES ADD (PRICE DECIMAL(8,2) NULL);
```

The data for this new column can be updated or inserted. this is shown in a later portion of this book.

4.16.10 Adding Data

To insert rows into a table, undertake the following:

```
INSERT INTO ANTIQUES VALUES (21, 01, 'Ottoman', 200.00);
```

This inserts the data into the table, as a new row, column-by- column, in the pre-defined order. Instead, let us change the order and leave Price blank:

```
INSERT INTO ANTIQUES (BUYERID, SELLERID, ITEM)
VALUES (01, 21, 'Ottoman');
```

4.16.11 Deleting Data

Let's delete this new row back out of the database:

```
DELETE FROM ANTIQUES
WHERE ITEM = 'Ottoman';
```

But if there is another row that contains 'Ottoman', that row will be deleted also. Let us delete all rows (one, in this case) that contain the specific data we added before:

```
DELETE FROM ANTIQUES
WHERE ITEM = 'Ottoman' AND BUYERID = 01 AND SELLERID = 21;
```

4.16.12 Updating Data

Let's update a Price into a row that doesn't have a price listed yet:

```
UPDATE ANTIQUES SET PRICE = 500.00 WHERE ITEM = 'Chair';
```

This sets all Chair's Prices to 500.00. As shown above, more WHERE conditionals, using AND, must be used to limit the updating to more specific rows. Also, the additional columns may be set by separating equal statements with commas.

4.16.13 Indexes

Indexes allow a DBMS to access data quicker (please note: this feature is nonstandard/not available on all systems). The system creates this internal data structure (the index) which causes selection of rows, when the selection is based on indexed columns, to occur faster. This index tells the DBMS where a certain row is in the table given an indexed-column value, much like a book index tells you what page a given word appears. Let us create an index for the OwnerID in the AntiqueOwners column:

```
CREATE INDEX OID_IDX ON ANTIQUEOWNERS (OWNERID);
```

Now on the names:

```
CREATE INDEX NAME_IDX ON ANTIQUEOWNERS (OWNERLASTNAME,
OWNERFIRSTNAME);
```

To get rid of an index, drop it:

```
DROP INDEX OID_IDX;
```

By the way, you can also "drop" a table, as well (careful!—that means that your table is deleted). In the second example, the index is kept on the two columns, aggregated together—strange behavior might occur in this situation...check the manual before performing such an operation.

Some DBMS's do not enforce primary keys; in other words, the uniqueness of a column is not enforced automatically. What that means is, if, for example, we tried to insert another row into the AntiqueOwners table with an OwnerID of 02, some systems will allow me to do that, even though we do not, as that column is supposed to be unique to that table (every row value is supposed to be different). One way to get around that is to create a unique index on the column that we want to be a primary key, to force the system to enforce prohibition of duplicates:

```
CREATE UNIQUE INDEX OID_IDX ON ANTIQUEOWNERS (OWNERID);
```

4.16.14 GROUP BY & HAVING

One special use of GROUP BY is to associate an aggregate function (especially COUNT; counting the number of rows in each group) with groups of rows. First, assume that the Antiques table has the Price column, and each row has a value for that column. We want to see the price of the most expensive item bought by each owner. We have to tell SQL to group each owner's purchases, and tell us the maximum purchase price:

```
SELECT BUYERID, MAX(PRICE)
FROM ANTIQUES
GROUP BY BUYERID;
```

Now, say we only want to see the maximum purchase price if the purchase is over $1000, so we use the HAVING clause:

```
SELECT BUYERID, MAX(PRICE)
FROM ANTIQUES
GROUP BY BUYERID
HAVING PRICE 1000;
```

4.16.15 More Subqueries

Another common usage of subqueries involves the use of operators to allow a Where condition to include the Select output of a subquery. First, list the buyers who purchased an expensive item (the Price of the item is $100 greater than the average price of all items purchased):

```
SELECT BUYERID
FROM ANTIQUES
WHERE PRICE >
(SELECT AVG(PRICE) + 100 FROM ANTIQUES);
```

The subquery calculates the average Price, plus $100, and using that Fig.ure, an OwnerID is printed for every item costing over that Fig.ure. One could use DISTINCT BUYERID, to eliminate duplicates.

List the Last Names of those in the AntiqueOwners table, ONLY if they have bought an item:

```
SELECT OWNERLASTNAME
FROM ANTIQUEOWNERS
WHERE OWNERID IN
(SELECT DISTINCT BUYERID FROM ANTIQUES);
```

The subquery returns a list of buyers, and the Last Name is printed for an Antique Owner if and only if the Owner's ID appears in the subquery list (sometimes called a candidate list).

Note: on some DBMS's, equals can be used instead of IN, but for clarity's sake, since a set is returned from the subquery, IN is the better choice.

For an Update example, we know that the gentleman who bought the bookcase has the wrong First Name in the database...it should be John:

```
UPDATE ANTIQUEOWNERS
SET OWNERFIRSTNAME = 'John'
WHERE OWNERID =
(SELECT BUYERID FROM ANTIQUES WHERE ITEM = 'Bookcase');
```

First, the subquery finds the BuyerID for the person(s) who bought the Bookcase, then the outer query updates his First Name.

Remember this rule about subqueries: when you have a subquery as part of a WHERE condition, the Select clause in the subquery must have columns that match in number and type to those in the Where clause of the outer query. In other words, if you have "WHERE ColumnName = (SELECT...);", the Select must have only one column in it, to match the ColumnName in the outer Where clause, and they must match in type (both being integers, both being character strings, etc.).

4.16.16 EXISTS & ALL

EXISTS uses a subquery as a condition, where the condition is True if the subquery returns any rows, and False if the subquery does not return any rows; this is a nonintuitive feature with few unique uses. However, if a prospective customer wanted to see the list of Owners only if the shop dealt in Chairs, try:

```
SELECT OWNERFIRSTNAME, OWNERLASTNAME
FROM ANTIQUEOWNERS
WHERE EXISTS
(SELECT * FROM ANTIQUES WHERE ITEM = 'Chair');
```

If there are any Chairs in the Antiques column, the subquery would return a row or rows, making the EXISTS clause true, causing SQL to list the Antique Owners. If there had been no Chairs, no rows would have been returned by the outside query.

ALL is another unusual feature, as ALL queries can usually be done with different, and possibly simpler methods; let us take a look at an example query:

```
SELECT BUYERID, ITEM
FROM ANTIQUES
WHERE PRICE = ALL
(SELECT PRICE FROM ANTIQUES);
```

This will return the largest priced item (or more than one item if there is a tie), and its buyer. The subquery returns a list of all Prices in the Antiques table, and the outer query goes through each row of the Antiques table, and if its Price is greater than or equal to every (or ALL) Prices in the list, it is listed, giving the highest priced Item. The reason "=" must be used is that the highest priced item will be equal to the highest price on the list, because this Item is in the Price list.

4.16.17 UNION & Outer Joins (briefly explained)

There are occasions where you might want to see the results of multiple queries together, combining their output; use UNION. To merge the output of the following two queries, displaying the ID's of all Buyers, plus all those who have an Order placed you may undertake the following:

```
SELECT BUYERID
FROM ANTIQUES
UNION
SELECT OWNERID
FROM ORDERS;
```

Notice that SQL requires that the Select list (of columns) must match, column-by-column, in data type. In this case BuyerID and OwnerID are of the same data type (integer). Also notice that SQL does automatic duplicate elimination when using UNION (as if they were two "sets"); in single queries, you have to use DISTINCT.

The outer join is used when a join query is "united" with the rows not included in the join, and are especially useful if constant text "flags" are included. First, look at the query:

```
SELECT OWNERID, 'is in both Orders & Antiques'
FROM ORDERS, ANTIQUES
WHERE OWNERID = BUYERID
UNION
SELECT BUYERID, 'is in Antiques only'
FROM ANTIQUES
WHERE BUYERID NOT IN
(SELECT OWNERID
FROM ORDERS);
```

The first query does a join to list any owners who are in both tables, and putting a tag line after the ID repeating the quote. The UNION merges this list with the next list. The second list is generated by first listing those ID's not in the Orders table, thus generating a list of ID's excluded from the join query. Then, each row in the Antiques table is scanned, and if the BuyerID is not in this exclusion list, it is listed with its quoted tag. There might be an easier way to make this list, but it's difficult to generate the informational quoted strings of text.

This concept is useful in situations where a primary key is related to a foreign key, but the foreign key value for some primary keys is NULL. For example, in one table, the primary key is a salesperson, and in another table is customers, with their salesperson listed in the same row. However, if a salesperson has no customers, that person's name won't appear in the customer table. The outer join is used if the listing of all salespersons is to be printed, listed with their customers, whether the salesperson has a customer or not—that is, no customer is printed (a logical NULL value) if the salesperson has no customers, but is in the salespersons table. Otherwise, the salesperson will be listed with each customer.

Another important related point about Nulls having to do with joins: the order of tables listed in the From clause is very important. The rule states that SQL "adds" the second table to the first; the first table listed has any rows where there is a null on the join column displayed; if the second table has a row with a null on the join column, that row from the table listed second does not get joined, and thus included with the first table's row data. This is another occasion (should you wish that data included in the result) where an outer join is commonly used. The concept of nulls is important, and it may be worth your time to investigate them further.

4.16.18 Select ... From

What do we use SQL for? Well, we use it to select data from the tables located in a database. Immediately, we see two keywords: we need to SELECT information FROM a table. There you have it. The most basic SQL structure:

```
SELECT "column_name" FROM "table_name"
```

To illustrate the above example, assume that we have the following table:

Table *Store_Information*

store_name	Sales	Date
Los Angeles	$1500	Jan-05-1999
San Diego	$250	Jan-07-1999
Los Angeles	$300	Jan-08-1999
Boston	$700	Jan-08-1999

Fig. 4.11

We shall use this table as an example throughout the tutorial (This table will appear in all sections). To select all the stores in this table, we key in,

```
SELECT store_name FROM Store_Information
Result:
store_name
Los Angeles
San Diego
Los Angeles
Boston
```

4.16.19 Distinct

The SELECT keyword allows us to grab all information from a column (or columns) on a table. This, of course, necessarily means that there will be redundencies. What if we only want to select each DISTINCT element? This is easy to accomplish in SQL. All we need to do is to add DISTINCT after SELECT. The syntax is as follows:

```
SELECT DISTINCT "column_name"
FROM "table_name"
```

For example, to select all distinct stores in Table Store_Information,

Table *Store_Information*

store_name	Sales	Date
Los Angeles	$1500	Jan-05-1999
San Diego	$250	Jan-07-1999
Los Angeles	$300	Jan-08-1999
Boston	$700	Jan-08-1999

Fig. 4.12

we key in,

```
SELECT DISTINCT store_name FROM Store_Information
Result:
store_name
Los Angeles
San Diego
Boston
```

4.16.20 Where

Next, we might want to conditionally select the data from a table. For example, we may want to only retrieve stores with sales above $1,000. To do this, we use the WHERE keyword. The syntax is as follows:

```
SELECT "column_name"
FROM "table_name"
WHERE "condition"
```

For example, to select all stores with sales above $1,000 in Table Store_Information,

AntiqueOwners

OwnerID	OwnerLastName	OwnerFirstName
01	Jones	Bill
02	Smith	Bob
15	Lawson	Patricia
21	Akins	Jane
50	Fowler	Sam

Fig. 4.13

we key in,

```
SELECT store_name FROM Store_Information WHERE Sales > 1000
Result:
store_name
Los Angeles
```

4.16.21 Functions

Since we have started dealing with numbers, the next natural question to ask is whether it is possible to do math on those numbers such as summing them up or taking the average of them. The answer is yes! SQL has several arithematic functions, among them SUM and AVG. The syntax for this is,

```
SELECT "function type"("column_name")
FROM "table_name"
```

For example, if we want to get the sum of all sales from our example table,

AntiqueOwners

OwnerID	OwnerLastName	OwnerFirstName
01	Jones	Bill
02	Smith	Bob
15	Lawson	Patricia
21	Akins	Jane
50	Fowler	Sam

Fig. 4.14

we would type in

```
SELECT SUM(Sales) FROM Store_Information
```

Result:

```
SUM(Sales)
$2750
$2750 represents the sum of all Sales entries: $1500 + $250 + $300 +
$700.
```

4.16.22 Count

Another arithematic function is COUNT. This allows us to COUNT up the number of row in a certain table. The syntax is,

```
SELECT COUNT("column_name")
FROM "table_name"
```

For example, if we want to find the number of store entries in our table,

Table ***Store_Information***

store_name	Sales	Date
Los Angeles	$1500	Jan-05-1999
San Diego	$250	Jan-07-1999
Los Angeles	$300	Jan-08-1999
Boston	$700	Jan-08-1999

Fig. 4.15

we would key in

```
SELECT COUNT(store_name)
FROM Store_Information
```

Result:

```
Count(store_name)
4
```

COUNT and DISTINCT can be used together in a statement to fetch the number of distinct entries in a table. For example, if we want to find out the number of distinct stores, we would type,

```
SELECT COUNT(DISTINCT store_name)
FROM Store_Information
```

Result:

```
Count(DISTINCT store_name)
3
```

4.16.23 Groupby

Now we return to the aggregate functions. Remember, we used the SUM keyword to calculate the total sales for all stores? What if we want to calculate the total sales for each store? Well, we need to do two things: First, we need to make sure we select the store name as well as total sales. Second, we need to make sure that all the sales Fig.ures are grouped by stores. The corresponding SQL syntax is,

```
SELECT "column_name1", SUM("column_name2")
FROM "table_name"
GROUP BY "column_name1"
```

In our example, table Store_Information,

Table ***Store_Information***

store_name	Sales	Date
Los Angeles	$1500	Jan-05-1999
San Diego	$250	Jan-07-1999
Los Angeles	$300	Jan-08-1999
Boston	$700	Jan-08-1999

Fig. 4.16

we would key in,

```
SELECT store_name, SUM(Sales)
FROM Store_Information
GROUP BY store_name
```

Result:

```
store_name SUM(Sales)
Los Angeles $1800
San Diego $250
Boston $700
```

The GROUP BY keyword is used when we are selecting multiple columns from a table (or tables) and at least one arithematic operator appears in the SELECT statement. When that happens, we need to GROUP BY all the other selected columns, i.e., all columns except the one(s) operated on by the arithematic operator.

4.16.24 Having

Another thing that people might want to do is to limit the output based on the corresponding sum (or any other aggregate functions). For example, we might want to see only the stores with sales over $1,500. Instead of using the WHERE clause, though, we need to use the HAVING clause, which is reserved for aggregate functions. The HAVING clause is typically placed near the end of SQL, and SQL statements with the HAVING clause may or may not include the GROUP BY clause. The syntax is,

```
SELECT "column_name1", SUM("column_name2")
FROM "table_name"
GROUP BY "column_name1"
HAVING (arithematic function condition)
```

Note: the GROUP BY clause is optional.

In our example, table Store_Information,

AntiqueOwners

OwnerID	OwnerLastName	OwnerFirstName
01	Jones	Bill
02	Smith	Bob
15	Lawson	Patricia
21	Akins	Jane
50	Fowler	Sam

Fig. 4.17

we would type,

```
SELECT store_name, SUM(sales)
FROM Store_Information
GROUP BY store_name
HAVING SUM(sales) > 1500
```

Result:

```
store_name SUM(Sales)
Los Angeles $1800
```

4.16.25 Alias

We next focus on the use of aliases. There are two types of aliases that are used most frequently: column alias and table alias.

In short, column aliases exist to help organizing output. In the previous example, whenever we see total sales, it is listed as SUM(sales). While this is comprehensible, we can envision cases where the column heading can be complicated (especially if it involves several arithmetic operations). Using a column alias would greatly make the output much more readable.

The second type of alias is the table alias. This is accomplpished by putting an alias directly after the table name in the FROM clause. This is convenient when you want to obtain information from two separate tables (the technical term is 'perform joins'). The advantage of using a table alias when doing joins is readily apparent when we talk about joins.

Before we get into joins, though, let us look at the syntax for both the column and table aliases:

```
SELECT "table_alias"."column_name1" "column_alias"
FROM "table_name" "table_alias"
```

Briefly, both types of aliases are placed directly after the item they alias for, separate by a white space. We again use our table, Store_Information,

Table ***Store_Information***

store_name	Sales	Date
Los Angeles	$1500	Jan-05-1999
San Diego	$250	Jan-07-1999
Los Angeles	$300	Jan-08-1999
Boston	$700	Jan-08-1999

Fig. 4.18

We use the same example as that in Section 6, except that we have put in both the column alias and the table alias:

```
SELECT A1.store_name Store, SUM(Sales) "Total Sales"
FROM Store_Information A1
GROUP BY A1.store_name
```

Result:

```
Store Total Sales
Los Angeles $1800
San Diego $250
Boston $700
```

Notice that difference in the result: the column titles are now different. That is the result of using the column alias. Notice that instead of the somewhat cryptic "Sum(Sales)", we now have "Total Sales", which is much more understandable, as the column header. The advantage of using a table alias is not apparent in this example. However, they will become evident in the next section.

4.16.26 Joins

Now we want to look at joins. To do joins correctly in SQL requires many of the elements we have introduced so far.

A Join of Two Tables in SQL

This SQL statement retrieves the names of all employees who work in the department named "Computer Science"

```
SELECT Emp.Name
FROM Employees AS Emp, Departments AS Dept
WHERE Emp.DeptID = Dept.DeptID AND Dept.Name = "Computer
Science"
```

A clause such as Employees AS Emp identifies Emp as an alias for Employees and simplifies the SQL statement.

The semantics of this SQL statement are similar to the previous one, except for the FROM clause. This FROM clause specifies that the data retrieved will come from the cross product of the tables Employees and Departments; you can view this cross product here. From this cross product we select rows satisfying the condition

```
Emp.DeptID = Dept.DeptID AND Dept.Name = "Computer Science";
```

you can view this set of rows here. Finally, the SELECT clause specifies which data to return, namely

```
Emp.Name
Sergio
Dick
```

The previous paragraph describes only the semantics of the SQL statement. In general, the cross-product is huge and a DBMS should find an evaluation strategy which avoids the cross-product.

This SQL statement is called a Join since it joins together two tables.

The semantics of a SELECT/FROM/WHERE SQL query should be clear from this example: begin with the cross product of terms in the FROM clause, retrieve rows satisfying the WHERE condition, then return the columns specified in the SELECT clause.

Even data retrieval statements in SQL can be much more complex than illustrated by our examples, e.g., it is possible to nest queries and to use aggregate functions such as COUNT or MAX.

4.16.27 Writing SQL Statements

Our previous two examples have shown how to derive the output of SQL statements. As with any programming language, it is much harder to transform a high-level specification into SQL. We can provide this advice:

- Review the examples above and be sure you can derive the output yourself.
- Try some of the examples below.
- When writing an SQL statement, begin with the FROM clause.
- After writing your SQL statement, derive its output and check that it agrees with the specification.
- Before writing an SQL statement, you should know the data type of each column.

For example, if you know that the type of EmpID is numeric, you would write the condition in our first example as:

```
Employee.EmpID = 123
```

instead of

```
Employee.EmpID = "123"
```

Just knowing that all data in the table appears to be numeric is not sufficient to infer the type of the column. The DBMS enforces strong typing, so will reject queries that do not conform to its data types. Furthermore, your SQL statements should be valid if additional data, such as the row

321A Jim CS

is added to the table.

A few examples and solutions:

Retrieve the IDs of all employes with the name Sergio

```
SELECT Emp.EmpID
FROM Employees as Emp
WHERE Emp.Name = "Sergio"
```

Retrieve the IDs of all employees whose department is in the PCAT building

```
SELECT Emp.EmpID
FROM Employees as Emp, Departments as Dept
WHERE Emp.EmpID = Dept.DeptID and Dept.Bldg = PCAT
```

Epilogue

We have covered a very small part of what SQL can do. For further references, browse your friendly neighbourhood bookstore for a book with SQL or Database in the title.

Emp. EmpId	Emp. Name	Emp. DeptID	Dept. DeptID	Dept. Name	Dept. Bldg
123	Sergio	CS	Comm	Communications	Lincoln
234	Lara	Comm	Comm	Communications	Lincoln
135	Elayne	Comm	Comm	Communications	Lincoln
124	Dick	CS	Comm	Communications	Lincoln
123	Sergio	CS	CS	Computer Science	PCAT
234	Lara	Comm	CS	Computer Science	PCAT
135	Elayne	Comm	CS	Computer Science	PCAT
124	Dick	CS	CS	Computer Science	PCAT

rows satisfying the condition

```
Emp.DeptID = Dept.DeptID AND Dept.Name = "Computer Science"
```

Emp. EmpId	Emp. Name	Emp. DeptID	Dept. DeptID	Dept. Name	Dept. Bldg
123	Sergai	CS	CS	Computer Science	PCAT
124	Dick	CS	CS	Computer Science	PCAT

Let us assume that we have the following two tables,

Table ***Store_Information***

store_name	Sales	Date
Los Angeles	$1500	Jan-05-1999
San Diego	$250	Jan-07-1999
Los Angeles	$300	Jan-08-1999
Boston	$700	Jan-08-1999

Table ***Geography***

region_name	store_name
East	Boston
East	New York
West	Los Angeles
West	San Diego

Fig. 4.19

and we want to find out sales by region. We see that table Geography includes information on regions and stores, and table Store_Information contains sales information for each store. To get the sales information by region, we have to combine the information from the two tables. Examining the two tables, we find that they are linked via the common field, "store_name". We will first present the SQL statement and explain the use of each segment later:

```
SELECT A1.region_name REGION, SUM(A2.Sales) SALES
FROM Geography A1, Store_Information A2
WHERE A1.store_name = A2.store_name
GROUP BY A1.region_name
```

Result:

```
REGION SALES
East $700
West $2050
```

The first two lines tell SQL to select two fields, the first one is the field "region_name" from table Geography (aliased as REGION), and the second one is the sum of the field "Sales" from table Store_Information (aliased as SALES). Notice how the table aliases are used here: Geography is aliased as A1, and Store_Information is aliased as A2. Without the aliasing, the first line would become

```
SELECT Geography.region_name REGION, SUM(Store_Information.Sales)
SALES
```

which is much more cumbersome. In essence, table aliases make the entire SQL statement easier to understand, especially when multiple tables are included.

Next, we turn our attention to line 3, the WHERE statement. This is where the condition of the join is specified. In this case, we want to make sure that the content in "store_name" in table Geography matches that in table Store_Information, and the way to do it is to set them equal. This WHERE statement is essential in making sure you get the correct output. Without the correct WHERE statement, a Cartesian Join will result. Cartesian joins will result in the query returning every possible combination of the two (or whatever the number of tables in the FROM statement) tables. In this case, a Cartesian join would result in a total of 4 x 4 = 16 rows being returned.

Sample Examples

Example 4.1 Calculating the Number of Days in a Month

You can use DATEDIFF for part of it:

```
select datediff(dd,'Oct 1, 1998','Nov 1, 1998') → Result: 31
select datediff(dd,'Feb 1, 1998','Mar 1, 1998') → Result: 28
select datediff(dd,'Feb 1, 2000','Mar 1, 2000') → Result: 29
select datediff(dd,'Feb 1, 1900','Mar 1, 1900') → Result: 28
```

Code Sample

```
Set nocount on
declare @dateInQuestion datetime,
@howManyDaysThisMonth int
select @dateInQuestion = getdate()
select @dateInQuestion = dateadd(DAY, 1 + (-1 * datepart(d,
@dateInQuestion)), @dateInQuestion) select @howManyDaysThisMonth =
datediff(DAY,@dateInQuestion,(dateadd(MONTH,1,@dateInQuestion)))
select'thisMonth'=datepart(m,@dateInQuestion),'DaysInThisMonth' =
@howManyDaysThisMonth
```

Example 4.2 A Method to Sum a Count

In a query that returns a count using COUNT(*), is there a way to sum the counts? In SQL Server 7.0 Books Online, it is written that it "Cannot perform an aggregate function on an expression containing an aggregate or a subquery."

When we use a GROUP BY clause, we get a series of counts; we would like to find a way to total them. we don't necessarily need to show the counts, just the resultant SUM.

The SUM of the counts would be the count from the entire table without grouping.

If you are using other aggregates besides COUNT, aggregating the aggregates may give a different value than just a simple aggregate on the whole table. You can use a derived table:

- select sum(counter) from
- (select counter=count(*) from titles group by type) as

To group in order to show the counts by grouping as well as the grand total, you can use the rollup.

```
select type, number=count(*) from titles
group by type with rollup
```

Doing this will give you a NULL for the type in the row with the total, but you can change that as follows:

```
select case
when grouping(type)=1 then 'TOTAL'
else type
end,
number=count(*) from titles
group by type with rollup
```

Code Sample

```
Set nocount on
```

```
use dbTopisv
drop table foo
go
create table foo
(counter int,type int)
Insert foo values(1,1)
Insert foo values(1,1)
Insert foo values(1,1)
Insert foo values(10,2)
Insert foo values(20,2)
Insert foo values(100 ,3)
Insert foo values(200,3)
select * from foo
select counter=count(*) from foo
select counter=count(*) from foo group by type
select sum(counter) from
(select counter=count(*) from foo group by type) as t
select counter=avg(counter) from foo group by type
select sum(counter) from
(select counter=avg(counter) from foo group by type) as t
select type, number=count(*) from foo
group by type with rollup
select case
when grouping(type)=1 then 'TOTAL'
else convert(char(10),type)
end Type,
number=count(*) from foo
group by type with rollup
```

Example 4.3. Dropping all but the MAX Values in a GROUP BY Query

Here we would like to get a query that returns the last entries in a table. We have the following fields at our disposal:

- UniqueID (incremented by one each time)
- City Name
- Date
- Sales

Here we would like to return the last entry in the table for each of the cities that have existing sales.

My GROUP BY clause is CityName, Sales. I can't cause the MAX function that is used against the UniqueID to return just the last sales Fig.ures by city. There may be weeks or days where not all cities have sales, so they may be spread out in the table.

Answer

```
Try this:
SELECT ....
FROM T INNER JOIN
  (
```

```
SELECT MAX(UniqueID) as UniqueId
FROM T
GROUP BY CityName, Sales
) LastEntries
ON T.UniqueId = LastEntries.UniqueId
Code Sample
Set nocount on
Use dbTopisv
drop table foo
go
Create table foo (UniqueID int,daysale datetime,CityName char(10),Sales
int)
Insert into foo values (1,'1999-02-01','aaaaa',10)
Insert into foo values (2,'1999-02-02','aaaaa',10)
Insert into foo values (3,'1999-02-03','aaaaa',20)
Insert into foo values (4,'1999-02-01','bbbbb',30)
Insert into foo values (5,'1999-02-03','bbbbb',30)
Insert into foo values (6,'1999-02-04','bbbbb',40)
Insert into foo values (7,'1999-02-02','ccccc',50)
Insert into foo values (8,'1999-02-03','ccccc',50)
Insert into foo values (9,'1999-02-05','ccccc',50)
Insert into foo values (10,'1999-02-07','ccccc',60)
SELECT *
FROM foo INNER JOIN
  (
SELECT MAX(UniqueID) as UniqueId
FROM foo
GROUP BY CityName, Sales
) LastEntries
ON foo.UniqueId = LastEntries.UniqueId
```

Example 4.4. Setting the Current Date as the Default Value for a Datetime Field How can we cause the default value in a datetime table field to be set to the current date stamp?

Answer

```
Use the GETDATE function as your default:
* create table foo (datefield datetime default getdate(), bar
varchar(10))
```

Then,

```
insert foo (bar) select 'blah'
```

The datetime field will be set to the current time.

Example 4.5. Showing Unique Records

we need to be able to display the count of unique rows within a SQL Server 7.0 database. With a SELECT statement and a GROUP BY. clause, we will get a result set containing citynames and unique user names with duplicate citynames. Instead, here are want to display the count of unique users within a city. Using DISTINCT does not seem to help.

Answer

Use the following SELECT statement:

```
select count(distinct users), city
from ...
group by city
```

Code Sample

```
Set nocount on
Use dbtopisv
Drop table foo
go
Create table foo (users char(10), city char(10))
go
Insert foo values ('useraa','citya')
Insert foo values ('useraa','citya')
Insert foo values ('useraa','citya')
Insert foo values ('userab','citya')
Insert foo values ('userab','citya')
Insert foo values ('userab','citya')
Insert foo values ('userac','cityb')
Insert foo values ('userad','cityb')
Insert foo values ('useraa','cityb')
Insert foo values ('useraa','cityb')
select count(distinct users) users, city
from foo
group by city
Select * from foo
```

The Delete Statement

The delete statement is used to delete records or rows from the table.

```
delete from "tablename"
where "columnname" OPERATOR "value" [and|or "column" OPERATOR "value"];
[ ] = optional
```

Examples:

```
delete from employee;
```

Note: if you leave off the where clause, all records will be deleted!

delete from employee

where lastname = 'May';

delete from employee

where firstname = 'Mike' or firstname = 'Eric';

To delete an entire record/row from a table, enter "delete from" followed by the table name, followed by the where clause which contains the conditions to delete. If you leave off the where clause, all records will be deleted.

Exercises

1. Select Statement

Enter select statements to:

(i) Display the first name and age for everyone that's in the table.

```
select first, age from empinfo;
```

(ii) Display the first name, last name, and city for everyone that's not from Payson.

```
select first, last, city from empinfo
where city <> 'Payson';
```

(iii) Display all columns for everyone that is over 40 years old.

```
select * from empinfo
where age > 40;
```

(iv) Display the first and last names for everyone whose last name ends in an "ay".

```
select first, last from empinfo
where last LIKE '%ay';
```

(v) Display all columns for everyone whose first name equals "Mary".

```
select * from empinfo
where first = 'Mary';
```

(vi) Display all columns for everyone whose first name contains "Mary".

```
select * from empinfo
where first LIKE '%Mary%';
```

2. Insert statement

It is time to insert data into your new employee table. Your first three employees are the following:

Jonie Weber, Secretary, 28, 19500.00
Potsy Weber, Programmer, 32, 45300.00
Dirk Smith, Programmer II, 45, 75020.00

Enter these employees into your table first, and then insert at least 5 more of your own list of employees in the table.

After they are inserted into the table, enter select statements to:

1. Select all columns for everyone in your employee table.
2. Select all columns for everyone with a salary over 30000.
3. Select first and last names for everyone that's under 30 years old.
4. Select first name, last name, and salary for anyone with "Programmer" in their title.
5. Select all columns for everyone whose last name contains "ebe".
6. Select the first name for everyone whose first name equals "Potsy".
7. Select all columns for everyone over 80 years old.
8. Select all columns for everyone whose last name ends in "ith".

Create at least 5 of your own select statements based on specific information that you would like to retrieve.

Answers

Your Insert statements should be similar to: (note: use your own table name that you created)

```
insert into myemployees_ts0211
(firstname, lastname, title, age, salary)
values ('Jonie', 'Weber', 'Secretary', 28, 19500.00);
```

1.
```
select * from
myemployees_ts0211
```

2.
```
select * from
myemployees_ts0211
where salary > 30000
```

3.
```
select firstname, lastname
from myemployees_ts0211
where age < 30
```

4.
```
select firstname, lastname, salary
from myemployees_ts0211
where title LIKE '%Programmer%'
```

5.
```
select * from
myemployees_ts0211
where lastname LIKE '%ebe%'
```

6.
```
select firstname from
myemployees_ts0211
where firstname = 'Potsy'
```

7.
```
select * from
myemployees_ts0211
where age > 80
```

8.
```
select * from
myemployees_ts0211
where lastname LIKE '%ith'
```

3. Exercise on Create Table

You have just started a new company. It is time to hire some employees. You will need to create a table that will contain the following information about your new employees: firstname, lastname, title, age, and salary. After you create the table, you should receive a small form on the screen with the appropriate column names. If you are missing any columns, you need to double-check your SQL statement and recreate the table.

IMPORTANT: When selecting a table name, it is important to select a unique name that no one else will use or guess. Your table names should have an underscore followed by your initials and the digits of your birth day and month. For example, Tom Smith, who was born on November 2nd, would name his table myemployees_ts0211 Use this convention for all of the tables you create. Your tables will remain on a shared database until you drop them, or they will be cleaned up if they aren't accessed in 4-5 days. If "support" is good, we hope to eventually extend this to at least one week. When you are finished with your table, it is important to drop your table.

Your create statement should resemble:

```
create table myemployees_ts0211
(firstname varchar(30),
lastname varchar(30),
title varchar(30),
age number(2),
salary number(8,2));
```

4. Update statement

   (after each update, issue a select statement to verify your changes).

   (i) Jonie Weber just got married to Bob Williams. She has requested that her last name be updated to Weber-Williams.

   (ii) Dirk Smith's birthday is today, add 1 to his age.

   (iii) All secretaries are now called "Administrative Assistant". Update all titles accordingly.

   (iv) Everyone that's making under 30000 are to receive a 3500 a year raise.

   (v) Everyone that's making over 33500 are to receive a 4500 a year raise.

   (vi) All "Programmer II" titles are now promoted to "Programmer III".

   (vii) All "Programmer" titles are now promoted to "Programmer II".

Create at least 5 of your own update statements and submit them.

Answers to exercises

Ans:

```
    update myemployees_ts0211
    set lastname='Sushma - Agarmwal' where firstname='Sushma' and
lastname='Agarwal';
    update myemployees_ts0211
    set age=age+1
    where firstname='Dirk' and lastname='Smith';
          update myemployees_ts0211
          set title = 'Administrative Assistant'
          where title = 'secretary';
          update myemployees_ts0211
          set salary = salary + 3500
          where salary < 30000;
          update myemployees_ts0211
          set salary = salary + 4500
          where salary > 33500;
          update myemployees_ts0211
          set title = 'Programmer III'
          where title = 'Programmer II'
          update myemployees_ts0211
          set title = 'Programmer II'
          where title = 'Programmer'
```

5. Delete Statement Exercises

(use the select statement to verify your deletes):

1. Jonie Weber-Williams just quit, remove her record from the table;
2. It's time for budget cuts. Remove all employees who are making over 70000 dollars.

CHAPTER 5

EMBEDDED SQL AND APPLICATION PROGRAMMING INTERFACES

5.1 Overview of Embedded SQL

Embedded SQL refers to the use of standard SQL commands embedded within a procedural programming language. Embedded SQL is a collection of these commands:

- All SQL commands, such as SELECT and INSERT, available with SQL with interactive tools
- Flow control commands, such as PREPARE and OPEN, which integrate the
- Standard SQL commands within a procedural programming language
- Embedded SQL also includes extensions to some standard SQL commands.

Embedded SQL is supported by the ORACLE Precompilers. The ORACLE Precompilers interpret embedded SQL statements and translate them into statements that can be understood by procedural language compilers. Each of these ORACLE Precompilers translates embedded SQL programs into a different procedural language:

- the Pro*Ada Precompiler
- the Pro*C Precompiler
- the Pro COBOL Precompiler
- the Pro FORTRAN Precompiler
- the Pro Pascal Precompiler
- the Pro PL/I Precompiler

5.2 Pro*C

Compiler Embedded SQL is a method of combining the computing power of a high-level language like C/C++ and the database manipulation capabilities of SQL. It allows you to execute any SQL statement from an application program. Oracle's embedded SQL environment is called Pro*C. A Pro*C program is compiled in two steps. First, the Pro*C precompiler recognizes the SQL statements embedded in the program, and replaces them with appropriate calls to the functions in the SQL runtime library. The output is pure C/C++ code with all the pure C/C++ portions intact. Then, a regular C/C++ compiler is used to compile the code and produces the executable. For details, see the section on Demo Programs.

5.3 SQL

All SQL statements need to start with EXEC SQL and end with a semicolon ";". You can place the SQL statements anywhere within a C/C++ block, with the restriction that the declarative statements do not come after the executable statements. As an example:

```
{ int a; /* ... */
   EXEC SQL SELECT salary INTO :a FROM Employee WHERE SSN=876543210;
   /* ... */
   printf("The salary is %d\n", a); /* ... */ }
```

5.4 Preprocessor Directives

The C/C++ preprocessor directives that work with Pro*C are #include and #if. Pro*C does not recognize #define. For example, the following code is invalid:

```
   #define THE_SSN 876543210 /* ... */
   EXEC SQL SELECT salary INTO :a FROM Employee WHERE SSN = THE_SSN;
   /* INVALID */
```

- Statement Labels

You can connect C/C++ labels with SQL as in:

```
EXEC SQL WHENEVER SQLERROR GOTO error_in_SQL; /* ... */
error_in_SQL: /* do error handling */
```

We will come to what WHENEVER means later in the section on Error Handling.

5.5 Host Variables

Basics Host variables are the key to the communication between the host program and the database. A host variable expression must resolve to an lvalue (i.e., it can be assigned). You can declare host variables according to C syntax, as you declare regular C variables. The host variable declarations can be placed wherever C variable declarations can be placed. (C++ users need to use a declare section; see the section on C++ Users.)

The C datatypes that can be used with Oracle include:

```
   * char * char[n] * int * short * long * float * double
```

- **VARCHAR[n]**—This is a psuedo-type recognized by the Pro*C precompiler. It is used to represent blank-padded, variable-length strings. Pro*C precompiler will convert it into a

structure with a 2-byte length field and a n-byte character array. You cannot use register storage-class specifier for the host variables. A host variable reference must be prefixed with a colon ":" in SQL statements, but should not be prefixed with a colon in C statements. When specifying a string literal via a host variable, the single quotes must be omitted; Pro*C understands that you are specifying a string based on the declared type of the host variable. C function calls and most of the pointer arithmetic expressions cannot be used as host variable references even though they may indeed resolve to lvalues.

The following code illustrates both legal and illegal host variable references:

```
int deptnos[3] = { 000, 111, 222 }; int get_deptno() { return
deptnos[2];
} int *get_deptnoptr() { return &(deptnos[2]);
} int main() { int x; char *y; int z; /* ... */
EXEC SQL INSERT INTO emp(empno, ename, deptno) VALUES(:x, :y, :z);/* EGAL
*/
EXEC SQL INSERT INTO emp(empno, ename, deptno) VALUES(:x + 1,
/* LEGAL: the reference is to x */ 'Big Shot',
/* LEGAL: but not really a host var */ : deptnos[2]);
/* LEGAL: array element is fine */ EXEC SQL INSERT INTO emp(empno,
ename, deptno) VALUES(:x, :y, :(*(deptnos+2)));
/* ILLEGAL: although it has an lvalue */
EXEC SQL INSERT INTO emp(empno, ename, deptno) VALUES(:x, :y,
:get_deptno()); /* ILLEGAL: no function calls */
EXEC SQL INSERT INTO emp(empno, ename, deptno) VALUES(:x, :y,
(*get_depnoptr()));
/* ILLEGAL: although it has an lvalue */ /* ... */ }
```

5.6 Pointers

You can define pointers using the regular C syntax, and use them in embedded SQL statements. As usual, prefix them with a colon:

```
int *x; /* ... */
EXEC SQL SELECT xyz INTO :x FROM ...;
```

The result of this SELECT statement will be written into *x, not x.

5.7 Structures

Structures can be used as host variables, as illustrated in the following example:

```
typedef struct { char name[21];
/* one greater than column length; for '\0' */
int SSN; } Emp; /* ... */ Emp bigshot; /* ... */
EXEC SQL INSERT INTO emp (ename, eSSN) VALUES (:bigshot);
```

5.8 Arrays

Host arrays can be used in the following way:

```
int emp_number[50];
char name[50][11]; /* ... */
EXEC SQL INSERT INTO emp(emp_number, name) VALUES (:emp_number,
:emp_name);
```

which will insert all the 50 tuples in one go. Arrays can only be single dimensional. The example char name[50][11] would seem to contradict that rule. However, Pro*C actually considers name a one-dimensional array of strings rather than a two-dimensional array of characters. You can also have arrays of structures. When using arrays to store the results of a query, if the size of the host array (say n) is smaller than the actual number of tuples returned by the query, then only the first n result tuples will be entered into the host array.

5.9 Indicator Variables

Indicator variables are essentially "NULL flags" attached to host variables. You can associate every host variable with an optional indicator variable. An indicator variable must be defined as a 2-byte integer (using the type short) and, in SQL statements, must be prefixed by a colon and immediately follow its host variable. Or, you may use the keyword INDICATOR in between the host variable and indicator variable. Here is an example:

```
short indicator_var;
EXEC SQL SELECT xyz INTO :host_var:indicator_var FROM ...; /* ... */
EXEC SQL INSERT INTO R VALUES(:host_var INDICATOR :indicator_var, ...);
```

You can use indicator variables in the INTO clause of a SELECT statement to detect NULL's or truncated values in the output host variables. The values Oracle can assign to an indicator variable have the following meanings:

1. The column value is NULL, so the value of the host variable is indeterminate. 0 Oracle assigned an intact column value to the host variable. >0 Oracle assigned a truncated column value to the host variable. The integer returned by the indicator variable is the original length of the column value.
2. Oracle assigned a truncated column variable to the host variable, but the original column value could not be determined. You can also use indicator variables in the VALUES and SET clause of an INSERT or UPDATE statement to assign NULL's to input host variables. The values your program can assign to an indicator variable have the following meanings: -1 Oracle will assign a NULL to the column, ignoring the value of the host variable. >=0 Oracle will assign the value of the host variable to the column.

5.10 Datatype Equivalencing

Oracle recognizes two kinds of datatypes: internal and external. Internal datatypes specify how Oracle stores column values in database tables. External datatypes specify the formats used to store values in input and output host variables. At precompile time, a default Oracle external datatype is assigned to each host variable. Datatype equivalencing allows you to override this default equivalencing and lets you control the way Oracle interprets the input data and formats the output data. The equivalencing can be done on a variable-by-variable basis using the VAR statement. The syntax is:

```
EXEC SQL VAR <host_var> IS <type_name> [ (<length>) ];
```

For example, suppose you want to select employee names from the emp table, and then pass them to a routine that expects C-style '\0'-terminated strings. You need not explicitly '\0'-terminate the names yourself. Simply equivalence a host variable to the STRING external datatype, as follows:

```
char emp_name[21]; EXEC SQL VAR emp_name IS STRING(21);
```

The length of the ename column in the emp table is 20 characters, so you allot emp_name 21 characters to accommodate the '\0'-terminator. STRING is an Oracle external datatype specifically designed to interface with C-style strings. When you select a value from the ename column into emp_name, Oracle will automatically '\0'-terminate the value for you. You can also set the equivalence of user-defined datatypes to Oracle external datatypes using the TYPE statement. The syntax is:

```
EXEC SQL TYPE <user_type> IS <type_name> [ (<length>) ][REFERENCE];
```

You can declare a user-defined type to be a pointer, either explicitly, as a pointer to a scalar or structure, or implicitly as an array, and then use this type in a TYPE statement. In these cases, you need to use the REFERENCE clause at the end of the statement, as shown below:

```
typedef unsigned char *my_raw; EXEC SQL TYPE my_raw IS VARRAW(4000)
REFERENCE; my_raw buffer; /* ... */ buffer = malloc(4004);
```

Here we allocated more memory than the type length (4000) because the precompiler also returns the length, and may add padding after the length in order to meet the alignment requirement on your system.

5.11 Dynamic SQL

While embedded SQL is fine for fixed applications, sometimes it is important for a program to dynamically create entire SQL statements. With dynamic SQL, a statement stored in a string variable can be issued. PREPARE turns a character string into a SQL statement, and EXECUTE executes that statement. Consider the following example.

```
char *s = "INSERT INTO emp VALUES(1234, 'jon', 3)";
EXEC SQL PREPARE q FROM :s;
EXEC SQL EXECUTE q;
Alternatively, PREPARE and EXECUTE may be combined into one statement:
char *s = "INSERT INTO emp VALUES(1234, 'jon', 3)";
EXEC SQL EXECUTE IMMEDIATE :s;
```

5.12 Error Handling

After each executable SQL statement, your program can find the status of execution either by explicit checking of SQLCA, or by implicit checking using the WHENEVER statement. These two ways are covered in details below.

5.12.1 SQLCA

SQLCA (SQL Communications Area) is used to detect errors and status changes in your program. This structure contains components that are filled in by Oracle at runtime after every executable SQL statement. To use SQLCA you need to include the header file sqlca.h using the #include directive. In case you need to include sqlca.h at many places, you need to first undefine the macro SQLCA with #undef SQLCA. The relevant chunk of sqlca.h follows:

```
* #ifndef SQLCA *
#define SQLCA 1 struct sqlca { /* ub1 */ char sqlcaid[8];
/* b4 */
long sqlabc; /* b4 */
long sqlcode; struct { /* ub2 */ unsigned short sqlerrml; /* ub1 */
char sqlerrmc[70]; } sqlerrm; /* ub1 */
char sqlerrp[8]; /* b4 */
long sqlerrd[6]; /* ub1 */
char sqlwarn[8]; /* ub1 */
char sqlext[8]; }; /* ... */
```

The fields in sqlca have the following meaning:

sqlcaid This string component is initialized to "SQLCA" to identify the SQL Communications Area.

sqlcabc This integer component holds the length, in bytes, of the SQLCA structure.

sqlcode This integer component holds the status code of the most recently executed SQL statement: 0 No error.

> 0 Statement executed but exception detected. This occurs when Oracle cannot find a row that meets your WHERE condition or when a SELECT INTO or FETCH returns no rows. <0 Oracle did not execute the statement because of an error. When such errors occur, the current transaction should, in most cases, be rolled back.

sqlerrm This embedded structure contains the following two components: sqlerrml - Length of the message text stored in sqlerrmc.

sqlerrmc - Up to 70 characters of the message text corresponding to the error code stored in sqlcode.

sqlerrp Reserved for future use.

sqlerrd This array of binary integers has six elements: sqlerrd[0] - Future use. sqlerrd[1] - Future use. sqlerrd[2] - Numbers of rows processed by the most recent SQL statement.

sqlerrd[3] - Future use.

sqlerrd[4] - Offset that specifies the character position at which a parse error begins in the most recent SQL statement.

sqlerrd[5] - Future use.

sqlwarn This array of single characters has eight elements used as warning flags. Oracle sets a flag by assigning to it the character 'W'.

sqlwarn[0] Set if any other flag is set.

sqlwarn[1] Set if a truncated column value was assigned to an output host variable.

sqlwarn[2] Set if a NULL column value is not used in computing a SQL aggregate such as AVG or SUM.

sqlwarn[3] Set if the number of columns in SELECT does not equal the number of host variables specified in INTO.

sqlwarn[4] Set if every row in a table was processed by an UPDATE or DELETE statement without a WHERE clause.

sqlwarn[5] Set if a procedure/function/package/package body creation command fails because of a PL/SQL compilation error.

sqlwarn[6] No longer in use.

sqlwarn[7] No longer in use. sqlext Reserved for future use. SQLCA can only accommodate error messages up to 70 characters long in its sqlerrm component. To get the full text of longer (or nested) error messages, you need the sqlglm() function: void sqlglm(char *msg_buf, size_t *buf_size, size_t *msg_length); where msg_buf is the character buffer in which you want Oracle to store the error message; buf_size specifies the size of msg_buf in bytes; Oracle stores the actual length of the error message in *msg_length. The maximum length of an Oracle error message is 512 bytes.

5.13 WHENEVER

Statement This statement allows you to do automatic error checking and handling.

The syntax is:

```
EXEC SQL WHENEVER <condition> <action>;
```

Oracle automatically checks SQLCA for <condition>, and if such condition is detected, your program will automatically perform <action>. <condition> can be any of the following: SQLWARNING - sqlwarn[0] is set because Oracle returned a warning

.SQLERROR - sqlcode is negative because Oracle returned an error

.NOT FOUND - sqlcode is positive because Oracle could not find a row that meets your WHERE condition, or a SELECT INTO or FETCH returned no rows <action> can be any of the following:

.CONTINUE - Program will try to continue to run with the next statement if possible

.DO - Program transfers control to an error handling function

.GOTO <label> - Program branches to a labeled statement

.STOP - Program exits with an exit() call, and uncommitted work is rolled back.

Some examples of the WHENEVER statement:

```
EXEC SQL WHENEVER SQLWARNING DO print_warning_msg(); EXEC SQL WHENEVER
NOT FOUND GOTO handle_empty;
```

Here is a more concrete example:

```
/* code to find student name given id */ /* ... */
for (;;)
{ printf("Give student id number : ");
scanf("%d", &id);
EXEC SQL WHENEVER NOT FOUND GOTO notfound;
EXEC SQL SELECT studentname INTO :st_name FROM student WHERE studentid =
:id;
printf("Name of student is %s.\n", st_name);
continue;
notfound: printf("No record exists for id %d!\n", id); }
/* ... */
```

5.14 C++ Users

To get the precompiler to generate appropriate C++ code, you need to be aware of the following issues: * Code emission by precompiler. To get C++ code, you need to set the option CODE=CPP while executing proc. C users need not worry about this option; the default caters to their needs. Parsing capability. The PARSE option of proc may take the following values:

*** PARSE=NONE.**

C preprocessor directives are understood only inside a declare section, and all host variables need to be declared inside a declare section.

*** PARSE=PARTIAL.**

C preprocessor directives are understood; however, all host variables need to be declared inside a declare section. * PARSE=FULL.

C preprocessor directives are understood and host variables can be declared anywhere. This is the default when CODE is anything other than CPP; it is an error to specify PARSE=FULL with CODE=CPP. So, C++ users must specify PARSE=NONE or PARSE=PARTIAL. They therefore lose the freedom to declare host variables anywhere in the code. Rather, the host variables must be encapsulated in declare sections as follows:

EXEC SQL BEGIN DECLARE SECTION; // declarations...

EXEC SQL END DECLARE SECTION; You need to follow this routine for declaring the host and indicator variables at all the places you do so.

File extension. You need to specify the option CPP_SUFFIX=cc or CPP_SUFFIX=C.

Location of header files

By default, proc searches for header files like stdio.h in standard locations. However, C++ has its own header files, such as iostream.h, located elsewhere. So you need to use the SYS_INCLUDE option to specify the paths that proc should search for header files.

List of Embedded SQL Statements Supported by Pro*C

Declarative Statements

EXEC SQL ARRAYLEN

To use host arrays with PL/SQL EXEC SQL BEGIN DECLARE

SECTION EXEC SQL END DECLARE SECTION

To declare host variables

EXEC SQL DECLARE: To name Oracle objects

EXEC SQL **INCLUDE**

To copy in files

EXEC SQL TYPE To equivalence datatypes **EXEC SQL VAR**

To equivalence variables **EXEC SQL WHENEVER**

To handle runtime errors Executable Statements

EXEC SQL ALLOCATE

To define and control Oracle data

EXEC SQL ALTER
EXEC SQL ANALYZE EXEC
SQL AUDIT
EXEC SQL COMMENT
EXEC SQL CONNECT
EXEC SQL CREATE
EXEC SQL DROP
EXEC SQL GRANT
EXEC SQL NOAUDIT EXEC
SQL RENAME EXEC
SQL REVOKE EXEC
SQL TRUNCATE EXEC
SQL CLOSE
EXEC SQL DELETE
To query and manipulate Oracle data EXEC SQL EXPLAIN PLAN EXEC SQL FETCH EXEC SQL INSERT EXEC SQL LOCK TABLE EXEC SQL OPEN EXEC SQL SELECT EXEC SQL UPDATE EXEC SQL COMMIT
To process transactions
EXEC SQL ROLLBACK
EXEC SQL SAVEPOINT
EXEC SQL SET TRANSACTION
EXEC SQL DESCRIBE
To use dynamic SQL EXEC
SQL EXECUTE
EXEC SQL PREPARE EXEC
SQL ALTER SESSION
To control sessions
EXEC SQL SET ROLE
EXEC SQL EXECUTE END-EXEC

5.15 Microsoft Embedded SQL

Server Microsoft threw its hat into the ring for embedded databases, demonstrating a lightweightWindows CE version of SQL Server at its Windows CE developer conference. The lightweight SQL Server is in internal testing at Microsoft now. There is no official launch date or name, said Microsoft officials. Microsoft does not know what the final size of the database will be, but it will be larger than the 50-Kbyte databases out on the market from Sybase and Oracle and smaller than the 1-Mbyte-plus laptop databases, said Barry Goffe, SQL Server

product manager. In particular, Microsoft will offer a storage engine, a query processor, and a replication engine to ISVs. Set-top boxes and small handhelds that need a sales-force-automation application are the likeliest places the embedded version of SQL will show up, he added. Oracle, Sybase, and IBM have all debuted or revealed plans for iny embedded databases.

There are three ways in which SQL can be used:

1. Executing SQL statement interactively (Direct or Interactive SQL),
2. Write a set of SQL statements in a module and then executing that module (Module language or Procedures), and
3. Writing SQL statements from an application programming interface such as PowerBuilder (Embedded SQL).

Although we have been using Direct Invocation of SQL to study SQL as a programming language, SQL is mostly used in the context of some other host language. Module language or Procedure is SQL code that resides separately from the host language. The host languages then use native mechanisms to directly invoke those procedures. In contrast, embedded SQL is SQL statement written directly in an application interface and executed along with other code in the application. Consider the following example of code written in the PowerBuilder environment:

```
String ls_nam Long ll_cost SELECT Item_Name,
Item_cost INTO :ls_name, :ll_cost FROM ITEM WHERE Item_id = '050';
IF SQLCA.SQLCode = -1 then MessageBox("HOSPITAL APPLICATION",
SQLCA.SQLErrText,Information!)
HALT END IF
IF ll_cost > 10 THEN MessageBox("HOSPITAL APPLICATION", "This Item costs
more than ten dollars")
```

When using Embedded SQL, we must pay attention to the following:

- **USING Matching Data Types.** The variable declaration of string and long are powerBuilder data types. In this example, PowerBuilder data type String matches with the SQL data type Char. The SELECT statement uses column item_name, which is of Char data type in SQL and holds the value in the PowerBuilder variable ls_name which is a string. The only expectation from SQL is that if the host language uses a data type, it should simply match with the characteristics of a SQL data type.
- **CHECKING FOR return codes.** When executing a SQL statements interactively, you may see an error message at times, even when your SQL statement in syntactically correct. This may happen in situations such as a subquery returning more than one row, or the requested record locked by some other user. You see the response visually and take appropriate actions. In Embedded SQL, since it is your application program that is sending the query, the programmer must take responsibility to check for the status of your query after executing a SQL statement. An attribute SQLCODE of the SQLCA transaction object contain the message returned by the database. This must be checked after every embedded SQL statement. In embedded SQL, we use the host variables to store the values return from the SQL query.

These host variables are application variables preceded by a colon (:). We must note that if a SQL query returns a result set (as opposed to a single value), then we obviously cannot store that in a single variable. In situations where we have to process multiple rows of table we must resort to CURSORS.

5.16 CURSORS

A cursor is something like a pointer that traverses a collection of rows and acts as intermediary to resolve our host language's inability to handle an arbitrary collection of rows. There are four Statements:

```
DECLARE findavg CURSOR FOR SELECT salary FROM Employee;
LONG ll_sal, ll_tot, ll_count OPEN findavg; emp: FETCH NEXT findavg INTO
:ll_sal;
IF SQLCA.SQLCODE <> 0 THEN GOTO finished;
ELSE ll_count++ ll_tot = ll_sal + ll_tot;
END IF GOTO emp finished: CLOSE findavg;
sle_1.text = string (ll_tot/ ll_count)
```

There are four steps to using a CURSOR:

- DECLARE a cursor.
- DECLARE findavg CURSOR FOR SELECT salary FROM Employee is an example of a declaration of cursor. Findavg is the name of the cursor that is later used in the OPEN and CLOSE statements as well. Any value that must be supplied by the host program can be specified in the DECLARE statement. As an example, if we are interested in finding the average of only those employees whose salary is above a certain value specifed by the host program, we could have written.

```
DECLARE findavg CURSOR FOR SELECT salary FROM Employee WHERE salary >
:ll_mysal OPEN the cursor.
```

There is nothing complex about opening the cursor. OPEN findavg will do it! Note that there are no host variables associated with the OPEN statement. All values to the cursor are specified only in the DECLARE statement, but it is actually obtained by the OPEN statement.

.FETCH statement allows the data to be into applications' area one row at a time. It simply fetches the row after the row on which the cursor is positioned. Many DBMS also support formats of FETCH other than the customary (and default) FETCH NEXT such as FETCH FIRST, FETCH PRIOR, or FETCH LAST After fetching, you can also do a positioned delete or Update.

.DELETE FROM TableName WHERE CURRENT OF CursorName; will delete the current row where the cursor is positioned. CLOSE is used to close the cursor. Only when you CLOSE the cursor the system resources are freed up. It also protects you from inadvertently using the cursor in later data operations.

CHAPTER 6

OBJECT MODELLING AND DATABASE DESIGN

6.1 Modelling

Modelling is the process of attempting to accurately characterize a physical phenomenon by mathematical or logical relationships which can be programmed to run on a computer. Then, by generating the appropriate representative input data, we can use this model to simulate the real system and observe the responses to various input conditions.

Models are important aids to thinking, creativity and communication and, as these are major preoccupations of the analyst, their relevance should be clear. The selection of an appropriate model requires flexibility and imagination, adopting a model relevant to the circumstances.

6.2 Models

A model is not just a smaller copy of something. A model is a formal representation which:

- Hides uninteresting detail
- Substitutes symbols for bulkier components
- Highlights important facts
- Promotes understanding of the whole.

The modeling phase of database design is crucial to assure an accurate representation of the user's perspective of the business. Some reasons for modeling data include the following:

- Comprehending thoroughly the user's perspective of the data.
- Studying data apart from physical considerations.
- Understanding who uses the data and when.
- Preventing duplication of data in the database.

- Providing complete, accurate, and essential data input for physical design.
- A sound database design costs less!
- Better match to requirements = happier users, less maintenance
- More understandable solutions = more utilization, less maintenance
- Fewer construction errors = less debugging, less maintenance
- More system flexibility = less maintenance
- Greater reusability of ideas = less maintenance

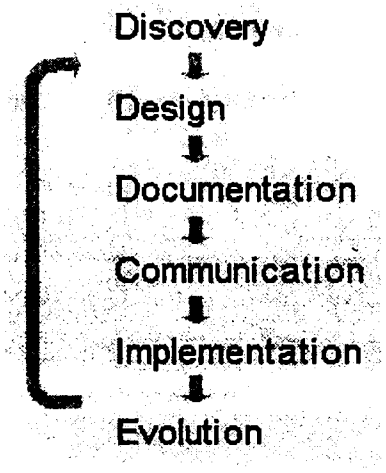


Fig. 6.2

One particular development, the use of models, is proving to be of especial importance because their use extends intuition and experience by analyzing the effects of uncertainty and by exploring the likely consequences of different planning assumptions.

A model is any simplified abstract of reality. It may be a physical object such as an architectural scale model or it may be what is termed a 'symbolic model'. These are representations of reality in numeric, algebraic, symbolic or graphical form.

Business models are symbolic models which represents the operations of the organisation by set of logically linked arithmetic and algebraic statements. Such models enable operations to be explored at low cost and with nil risk. Models are invariably computer-based and; use the processing power of the computer to enhance a manager's analytical ability.

6.2.1 Introduction to Data Modeling

Most people involved in application development follow some kind of methodology. A methodology is a prescribed set of processes through which the developer analyses the client's requirements and develops an application. Major database vendors and computer gurus all practice and promote their own methodology. Some database vendors even make their analysis, design, and development tools conform to a particular methodology. If you are using the tools of a particular vendor, it may be easier to follow their methodology as well. For example, when CNS develops and supports Oracle database applications, it uses the Oracle toolset. Accordingly, CNS follows

Oracle's CASE*Method application development methodology (or a reasonable facsimile thereof).

One technique commonly used in analyzing the client's requirements is data modeling. The purpose of data modelling is to develop an accurate model, or graphical representation, of the client's information needs and business processes. The data model acts as a framework for the development of the new or enhanced application. There are almost as many methods of data modeling as there are application development methodologies. CNS uses the Oracle CASE*Method for its data modeling.

As time goes by, applications tend to accrue new layers, just like an onion. We develop more paper pushing and report printing, adding new layers of functions and features. Soon it gets to the point where we can only see with difficulty the core of the application where its essence lies. Around the core of the application we see layer upon layer, protecting, nurturing, but ultimately obscuring the core. Our systems and applications often fall victim to these protective or hiding processes. The essence of an application is lost in the shuffle of paper and the accretion of day-to-day changes. Data modeling encourages both the developer and the client to tear off these excess layers, to explore and revisit the essence or purpose of the application once more. The new analysis determines what needs to feed into and what feeds from the core purpose.

Creating an information system is a process that ranges from planning to implementation. The systems development life cycle (SDLC) is a tool used to manage the activities of such a systems development project. Professionals use various strategies, or methodologies, to complete each phase of the SDLC. The modeling phase ensures that user requirements are met, issues are addressed, and proper information is provided to developers. Two methodologies commonly used to execute the modeling phase of a systems development project are:

- Process-oriented

Systems are modeled based on the flow of business processes.

- Data-oriented

Systems are modelled and based on the business application's data, regardless of data usage or data access needs. The phases of process modeling and data modeling do not coincide. As a result, it is difficult to produce deliverables from both methods in a parallel timeframe. Although process modeling is still used, in recent years database designers have placed more emphasis on the use of data modeling. The process of data modeling generally involves a phase concept similar to that of the SDLC. This course will focus on the phases, techniques, and aspects of data modeling implementing object model and physical model techniques.

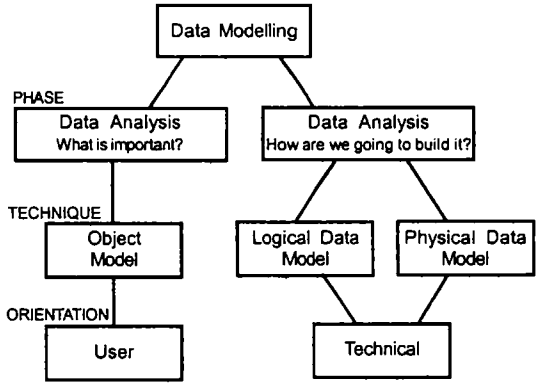


Fig. 6.3

In the creation of a database, data modeling is also described as a step-by-step process. Each step is designed to solve certain problems, meet specific goals, and naturally lead to the next step. This process is usually easier and more productive if each step is done independently and in order.

6.2.2 Models in a System Context

We can think of modeling an information system as viewing a cube from (at least) three dimensions.

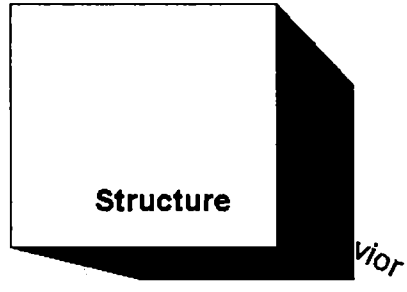


Fig. 6.4

6.2.3 The Relational Roots of Data Models

Modeling data is not natural or intuitive. We use abstractions to translate from business problems to RDBMS solutions. Relational theory lies at the root.

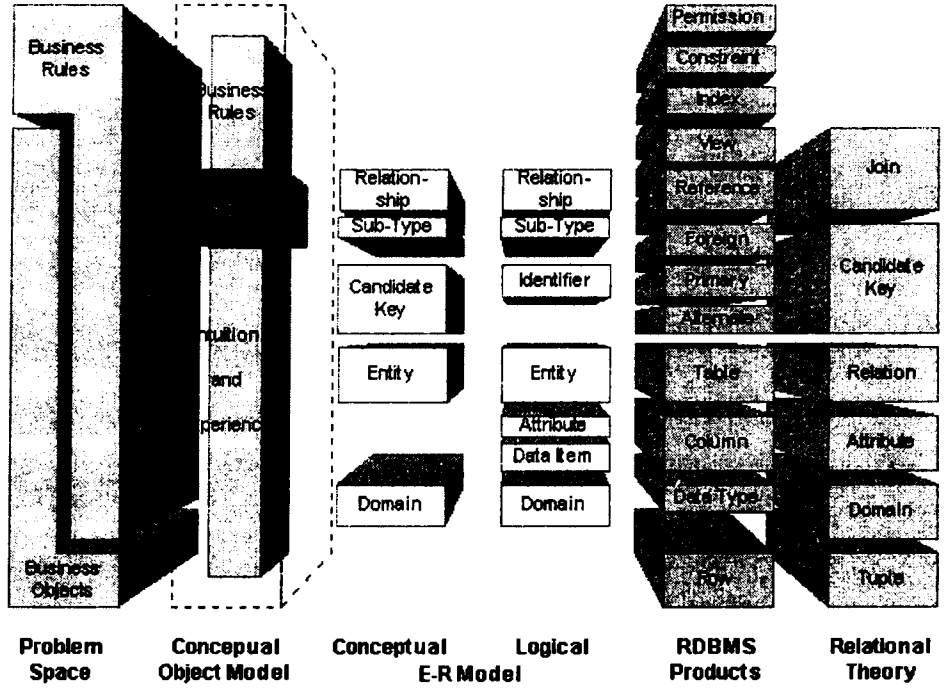


Fig. 6.5

6.2.4 Data Model: Reality to Relational

Today, most databases are relational. But the real world is not. Data models are built to map business concepts into relational objects.

6.3 Types of Data Models

Data models are usually categorized by levels of abstraction:

- Conceptual
- Logical
- Physical

These have no agreed formal definitions. Professional data modelers understand the approximate scope of each. These layers may appear in different ways. Some approaches deal only with the physical or logical model. Others offer elements of all three but not necessarily in three separate views.

6.3.1 Conceptual Data Model

A conceptual data model shows data through business eyes. It suppresses technical details to emphasize:

- All entities which have business meaning
- Important relationships (including many-to-many)
- A few significant attributes in the entities
- A few identifiers or candidate keys

6.3.2 Conceptual Data Model – An Example

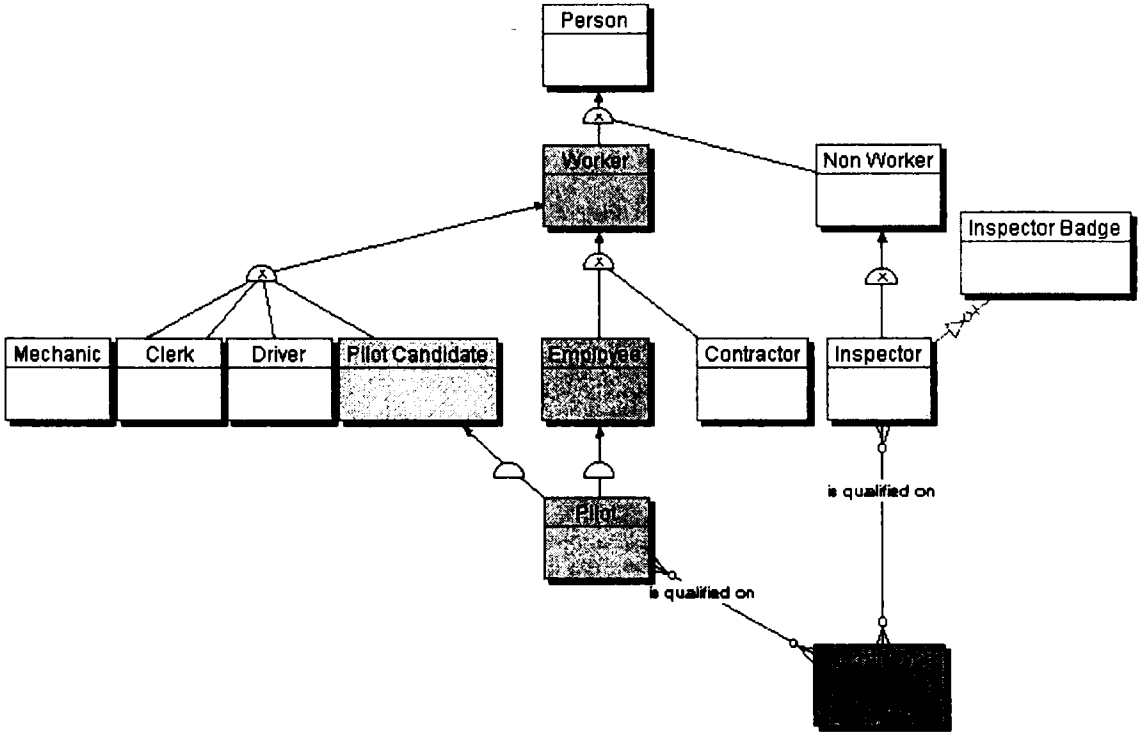


Fig. 6.6

Logical Data Model - Same Example

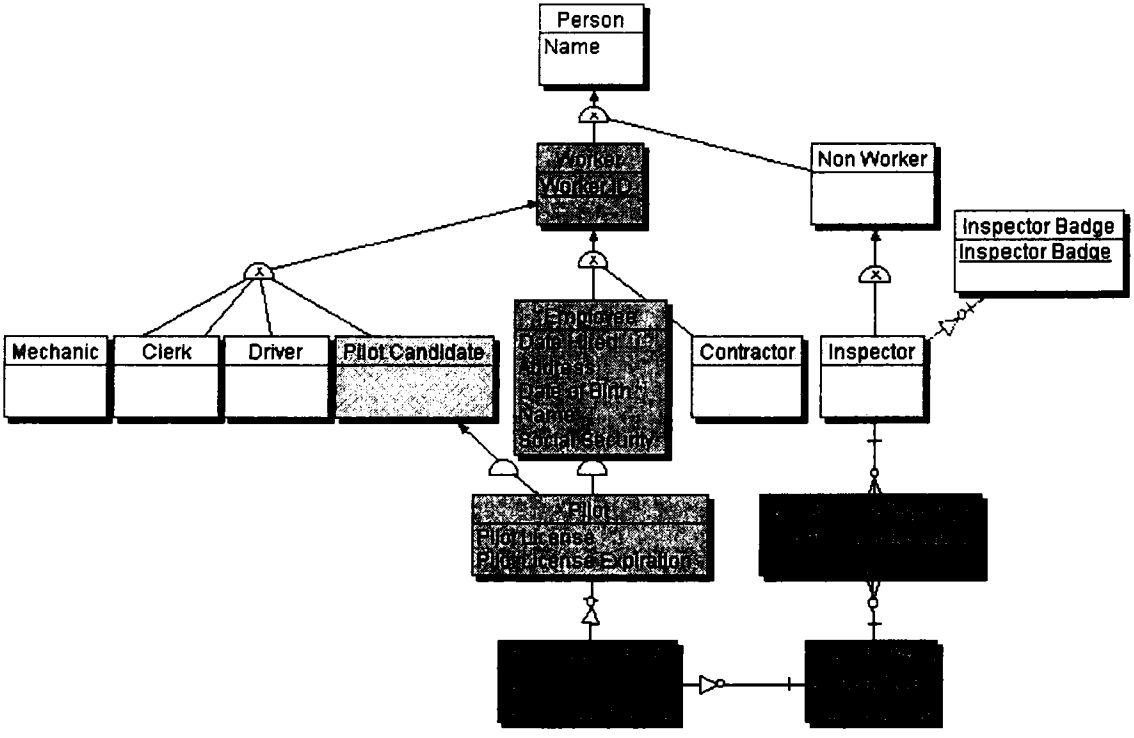


Fig. 6.7

Physical Data Model - Same Example

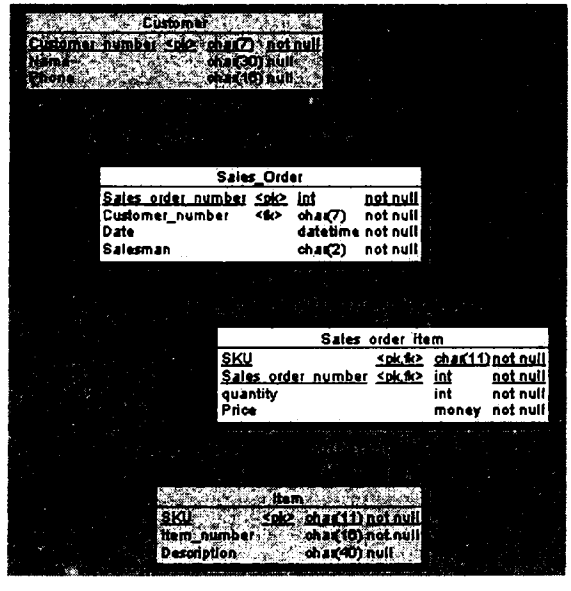


Fig. 6.8

6.3.3 The Logical Data Model

The Logical Data Model is a generic relational schema (in at least 1NF) which

- Replaces many-to-many relationships with associative entities
- Defines a full population of entity attributes

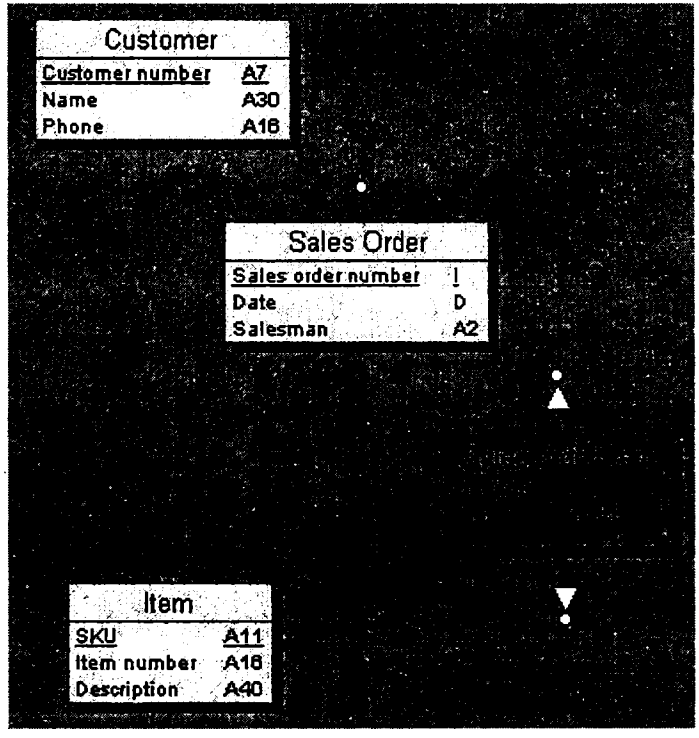


Fig. 6.9

Propagation of foreign keys may be explicit or implied in a logical data model. As long as the resulting physical schema includes the necessary foreign key columns and joins, the representation of foreign keys in the logical model is a matter of convenience and taste.

Replacing many-to-many relationships with associative entities is necessary to model 1st normal form, support internal attributes and secondary relationships, and enable alternate identifiers.

There are other forms of conceptual data models, such as Object Role Models.

A Logical Data Model

- May Use non-physical entities for domains and sub-types
- Establishes entity identifiers
- Has no specifics for any RDBMS or conFig.uration

6.3.4 Physical Data Model

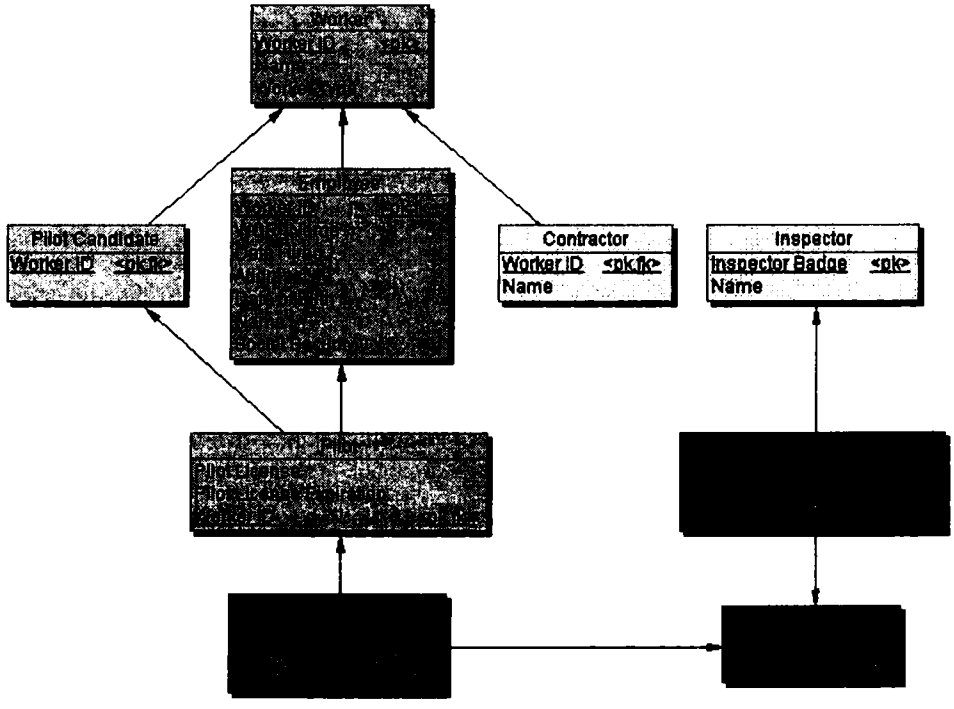


Fig. 6.10

A physical data model is a database design for:

- One DBMS product;
- One site conFig.uration; and
- Physical Data Model.

A physical data model may include:

- Referential integrity;
- Indexes;
- Views;
- Alternate keys and other constraints; and
- Tablespaces and physical storage objects.

6.4 Model Development

To develop a model which is realistic and has adequate predictive qualities is a collaborative effort between management and information specialists.

The key points are:

- The model should have a purpose and be objective orientated.

- Model building is an iterative, creative process with the aim of identifying those variable and relationship which must be included in the model so that it is capable of predicting overall system performance. It is not essential or indeed possible, to including all variables in a model. The variables in a model of greatest importance are those which govern, to a greater or less extent, the achievement of the specified objectives. These are the critical variables.
- The best model is the simplest one with the fewest variables that has adequate predictive qualities. To obtain this ideal there must be a thorough understanding of the system. The management who operate the system have this understanding and must be involved in the model building, otherwise over elaborate and overly mathematical models may result if the model building exercise is left to systems professionals.

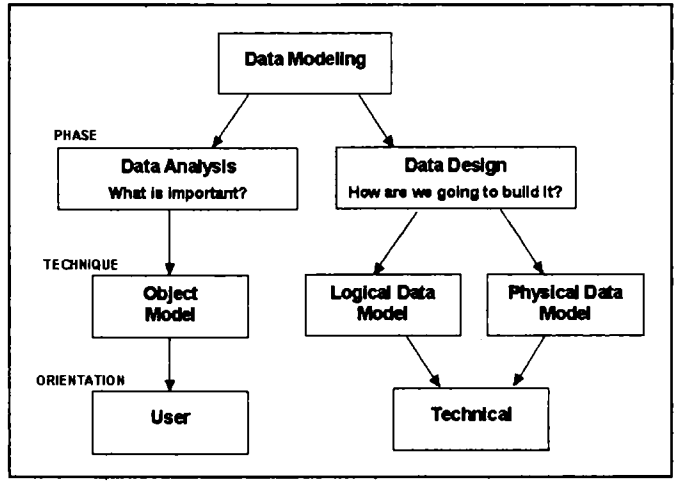


Fig. 6.11: Data Modelling Stage

6.5 Attributes of Modeling

Modeling demands the following conventions and attributes.

- **Simplicity**

Models are used for simplifying the complicated business world. In reality systems are usually large, complex and confusing. If the Analyst is to be able to gain understanding then unnecessary complexity must be stripped way. Recalling all the detail about real systems will also be difficult and the analyst will wish to keep track of a large amount of information so that it can later be examined and understood. For this propose pictorial or graphical models are more suitable than narrative ones. Imagine the difficulty of reading a map which is presented entirely in words! A good model represents the system's structures and relationships in a clear and concise way.

- **Consistency**

Consistency is highly desirable. Symbols may be used to represent various aspects of information flow, organisational structure, decision making processes, etc. The same symbol needs to be

used consistently through all models. Envisage the difficulties of following a map which changes its scale and symbolic presentation from page to page.

- **Completeness**

Models need to be reasonably complete. Omissions and vague areas of understanding in a model lead to similar difficulties with the actual system. Organisations develop standard models in an attempt to ensure that models used are both consistent and comprehensive. The National Computing Centre Documentation Standards (NCC,1979, 1987) provide a standard set of consistent and interrelated models which have to be constructed and completed. It includes a cross-reference system which the Analyst can use to check for consistency and completeness in his models. However it is unlikely that standards alone can provide the variety of models needed in system development. Any attempt to do so can quickly lead to bureaucratic and time-consuming form- filling which threatens the very essence of models.

- **Precision**

Appropriate accuracy is necessary if a model is to communicate information effectively. This demands an understanding of the propose of the model. A program specification is different in detail from an overview of a proposed system presented to user management, and so different models will be necessary. Many user manuals fail to grasp this point. They fail to communicate because all facts are presented at the same level of accuracy and detail.

- **Hierarchy**

Hierarchical models enable the Analyst to maintain several levels of detail in the constructed models. A high-level model will have little detail on it since its primary purpose is to highlight the most important features of the system. A motorway map contained in a diary will show only the motorway network or most of the main roads linking the towns and cities. It will be sufficient to plan a journey from New Delhi to Noida (by motorway), but it will not help the motorist find his way to a particular district of Noida. For this purpose a lower level model is required showing the main trunk roads, and this should permit the general location of the desired area. At this point a still lower lever model is necessary if the motorist is to find the actual street. He will then consult a street plan, and possibly a location map of the particular premises he wishes to visit to complete his journey. A similar model is required in information systems design. Each tier of the model will provide vital information systems design. Each tier of the model will provide vital information, but the different levels will mean that detail can be progressively absorbed. Imagine the condition of the motorist trying to drive from London to Manchester using only street plan!

6.6 Types of Model

- The E-R Model
- The Object-Oriented Model
- Record-based Logical Models
- The Relational Model
- The Network Model

- The Hierrarchical Model
- Physical Data Models

6.6.1 The E-R Model

The entity-relationship model is based on a perception of the world as consisting of a collection of basic objects (entities) and relationships among these objects.

An entity is a distinguishable object that exists. Each entity has associated with it a set of attributes describing it. For example number and balance for an account entity. A relationship is an association among several entities. e.g. A cust_acct relationship associates a customer with each account he or she has. The set of all entities or relationships of the same type is called the entity set or relationship set. Another essential element of the E-R diagram is the mapping cardinalities, which express the number of entities to which another entity can be associated via a relationship set. We will see later how well this model works to describe real-world situations.

The overall logical structure of a database can be expressed graphically by an E-R diagram:

- **Rectangles:** represent entity sets.
- **Ellipses:** represent attributes.
- **Diamonds:** represent relationships among entity sets.
- **Lines:** link attributes to entity sets and entity sets to relationships.

See Fig. 6.12 for an example.

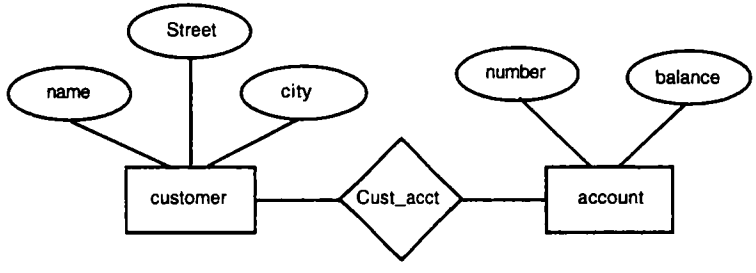


Fig. 6.12

6.6.2 The Object-Oriented Model

The object-oriented model is based on a collection of objects, like the E-R model.

An object contains values stored in instance variables within the object. Unlike the record-oriented models, these values are themselves objects.

Thus objects contain objects to an arbitrarily deep level of nesting. An object also contains bodies of code that operate on the the object.

These bodies of code are called methods. Objects that contain the same types of values and the same methods are grouped into classes.

A class may be viewed as a type definition for objects. Analogy: the programming language concept of an abstract data type. The only way in which one object can access the data of another object is by invoking the method of that other object. This is called sending a message to the object. Internal parts of the object, the instance variables and method code, are not visible externally. Result is two levels of data abstraction. For example, consider an object representing a bank account. The object contains instance variables number and balance. The object contains a method pay-interest which adds interest to the balance.

Under most data models, changing the interest rate entails changing code in application programs. In the object-oriented model, this only entails a change within the pay-interest method. Unlike entities in the E-R model, each object has its own unique identity, independent of the values it contains:

- Two objects containing the same values are distinct.
- Distinction is created and maintained in physical level by assigning distinct object identifiers.

Object-based Logical Models

- Describe data at the conceptual and view levels.
- Provide fairly flexible structuring capabilities.

Allow one to specify data constraints explicitly.

Over 30 such models, including:

- Entity-relationship model.
- Object-oriented model.
- Binary model.
- Semantic data model.
- Infological model.
- Functional data model.

6.6.3 Record Based Models

Also describe data at the conceptual and view levels. Unlike object-oriented models, are used to

- Specify overall logical structure of the database, and
- Provide a higher-level description of the implementation.
- Named so because the database is structured in fixed-format records of several types.
- Each record type defines a fixed number of fields, or attributes.
- Each field is usually of a fixed length (this simplifies the implementation).
- Record-based models do not include a mechanism for direct representation of code in the database.
- Separate languages associated with the model are used to express database queries and updates.
- The three most widely-accepted models are the relational, network, and hierarchical.

- This course will concentrate on the relational model.
- The network and hierarchical models are covered in appendices in the text.

6.6.4 Physical Data Models

- Are used to describe data at the lowest level.
- Very few models, e.g.
  - Unifying model.
  - Frame memory.

Instances and Schemes

- Databases change over time.
- The information in a database at a particular point in time is called an instance of the database.
- The overall design of the database is called the database scheme.
- Analogy with programming languages:
- Data type definition - scheme
- Value of a variable - instance

There are several schemes, corresponding to levels of abstraction:

- Physical scheme
- Conceptual scheme
- Subscheme (can be many)
- Logical data models.

Data flow diagrams, entity relationship models and entity life histories are the three major models used to present a logical view of the information systems of the enterprise. These three models are pictorial, and interrelated through the details maintained in a common data dictionary.

- Structure chart
- Data Flow Diagram
- Decision Tables.
- Decision Trees.
- Tight or Structured English.

6.7 Introduction to Object Modeling

The first step of data modeling involves identifying objects that are important to the user. In order to determine these objects accurately, designers and users must communicate frequently and in-depth. The goal of this joint effort is to produce an object model that ultimately results in an efficiently-structured base of data and information for the user. In this lesson, the object model will be illustrated with the entity-relationship technique.

Sometimes called an entity-relationship (E-R) model, this technique is actually a graphic diagram of the user's objects. Since the E-R diagram is graphic, a data dictionary is used to provide additional information, such as business rules and constraints, that cannot be drawn on a diagram. When used together, these tools provide users and designers with effective documentation for this phase of database design.

Object modeling interprets and represents the meaning of user's objects from the user's point of view, without regard for processes, structures, or implementation. To place objects in the proper perspective, developers usually conduct interviews and/or group discussions such as JAD sessions with users. Since information is sometimes more readily understood in graphic form, the E-R diagram, with its few simple geometric elements, has proved to be a successful tool for user sessions.

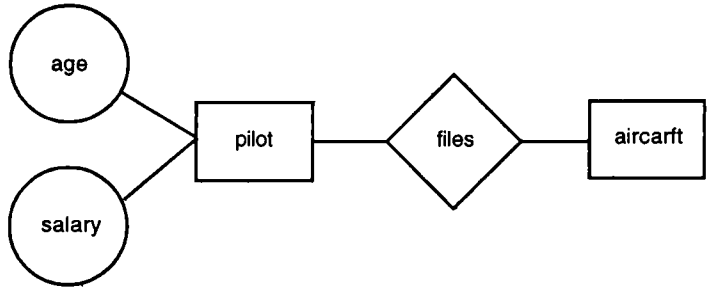


Fig. 6.13: Object Modelling

The E-R diagram provides three main concepts for data analysis:

Entities:

- Objects of the business.
- Relationships
- Associations between objects.

Attributes:

- Descriptive properties of objects and relationships.
- In addition to the basic E-R elements, this text will cover several variations of entities and relationships, including:

Entity hierarchies:

- Weak entities
- Characteristic entities
- Associative entities
- Multiple relationships
- Multi-member relationships
- Relationship roles
- Recursive relationships

Remember that in creating an E-R diagram, different modelers may use slightly different structures; there is no single correct design. Still, since it was introduced, the basic theory and design concepts of this technique have changed little. As a result, E-R diagramming continues to be a popular technique of object modeling.

6.8 Perspectives of Data Modeling

Since data modeling is not an exact science, it is important to realize that data modeling methodologies, techniques, and terms often overlap, depending on the perspective and training of the developer. This sometimes results in uncertainty about names and functions. For example, a model of user data might be called a business model, an object model, or a logical data model. Some professionals view these models as different techniques, while others use the names interchangeably.

- Object Model

A model of facts and objects that describe the business compiled by users and data modelers.

- Logical Data Model

Transformation of the object model into logical tables that may or may not appear as physical data tables in the DBMS. Policies, security functions, and audit features are added during this phase.

- Physical Data Model

Implementation of the logical data model into physical data tables. Structures and access methods are specified during this stage.

6.9 Types of Reality

Developers use facts about the business to build data models. Users provide and verify these facts during the modeling process. However, verification is not enough since different types of facts are used in various modeling phases. These facts are often classified into types of reality to distinguish between them for different models. Three types of reality used are:

- Facts of life,
- Facts of policy
- Facts of implementation.

There are as detailed below:

Facts of Life	Student buys books. Professor teaches class. Customer buys product.
Facts of Policy	Student can have no more than two majors. Course has a required textbook. Salesman has one sales region.
Facts of Implementation	Primary keys can contain no null values. Each cell can contain only one value. Filenames can have

Facts of life are static, the things not likely to change. They provide the "essence," or basic ingredients, of all models. These facts are simple statements depicted in the object model during the analysis phase. When accurately defined, these facts of life represent the user's view of reality and relevancy as defined by focus.

Facts of policy are the rules or conventions that define and govern everyday business. They are sometimes difficult to define and are subject to change. These facts usually appear in the logical data model during analysis and/or design.

Facts of implementation are determined by the system's construction and can include system limitations and structures. These are not facts about the business and are not included on object or logical data models. Facts of implementation are considered in the physical model during the implementation phase.

6.10 Fundamental Analysis Concepts

One of the early stages of database design is the analysis stage. In this stage, designers and analysts communicate with users to determine important business information. Two methods commonly used at this stage are process analysis and data analysis. Originally, developers designed applications from the process perspective, and some still do. More recently, however, the data analysis approach has become popular. Analysis from this perspective eliminates some of the problems of the process approach.

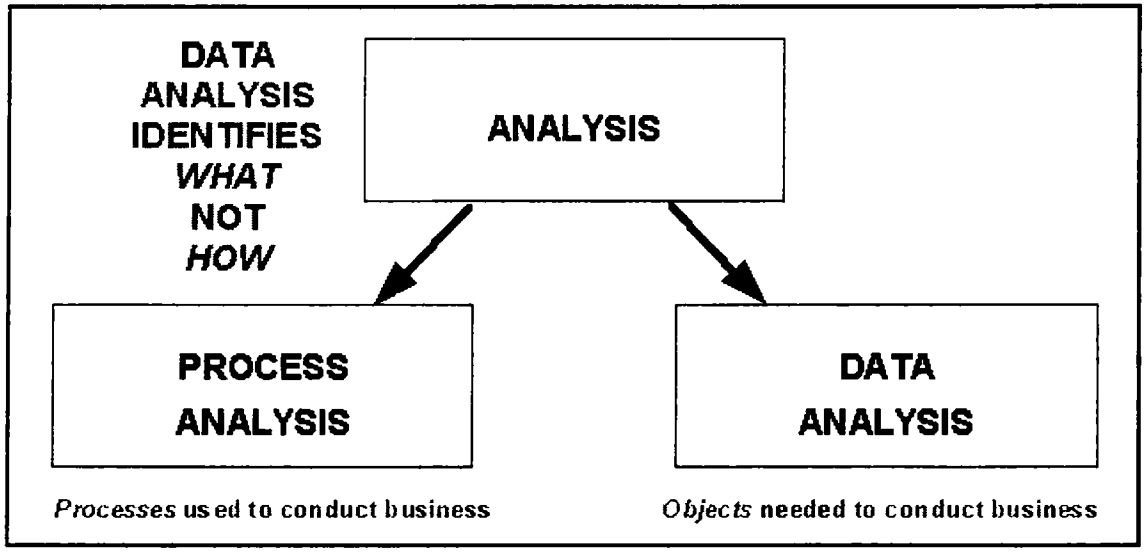


Fig. 6.14

6.10.1 The Traditional Process-Driven Approach

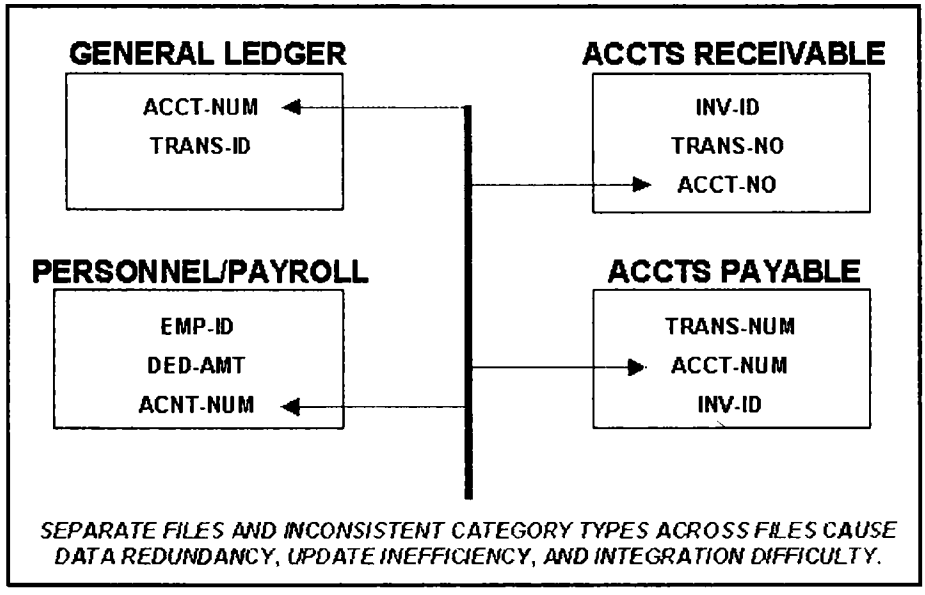


Fig. 6.15

In process analysis, designers examine the user's business processes and procedures in order to develop applications and files for an information system. The focus in this approach is on how business is carried out. Typically, this type of structured analysis is done along these guidelines:

- Identify relevant and required business processes.
- Analyse each process in complete detail.
- Convert processes to computer programs and manual procedures.
- Design storage methods for the data used by these processes.

Basically, process analysis is dynamic, showing the movement of information through the business organisation. For example, the student registration process could be described as follows:

- A student visits an advisor.
- The advisor fills out and approves the student registration form.
- The advisor sends the student registration form to the registration office.
- The registration office checks availability of the requested classes.
- The registration office sends the student a letter confirming classes.

Process modelling is not without its problems. For instance, since business processes are subject to change, so are process-oriented information systems. These changes can result in time-consuming rework for programmers. Also, in viewing business processes, developers may actually be viewing the same data from different perspectives. Therefore, the data could be replicated in various files, causing data redundancy and update inconsistency. In addition, duplicate data can be stored in conflicting formats, making file integration difficult.

Because this, analysis technique models business processes, it generally starts out representing processes and procedures that are already in place. However, this approach can lead to an incremental process of improvements to the existing system and prevent new and more creative ways of viewing problems.

6.10.2 The Data Analysis Approach

DATA MODELING STEPS				
Step	Data Analysis (what)		Database Design (how)	Database Construction
Deliverable	Object Model	Logical Data Model (Data Requirements)	Physical Data Model	Database
Constraints	Business world facts (based on focus)	plus: Data Characteristics	plus: Technology	——
Main Contributor	Users	Technicians		

Fig. 6.16: Data Modelling Steps

Using data analysis techniques, developers design programs and files based on objects of the user's business and their associations. This approach views the objects, or data, independently from processes, physical structure, or access methods. Thus, the focus is on what is important, not how it is done. Analysis of data overcomes some of the problems of process analysis regarding :

- Redundancy of Data. Objects are considered individually, the goal being that each data element is placed in only one location.
- Inconsistency of field types across files.
- Data elements are only stored once, so field type inconsistency is eliminated. *Inaccessibility issues, including:
  - Data buried within data structures.
  - Designs based on the data analysis approach are independent of structural considerations.
  - Data unavailable across applications.
  - Designs based on the data analysis approach are also independent of access issues. If the data is stored in the database, it can be accessed.
  - No data.

One reason for user input into data analysis is to ensure that all data is captured.

The data analysis approach provides a more stable structure for an information system. Since data is static, it seldom changes. Also, the emphasis on data rather than processes can provide users with new perspectives. As a result, an opportunity is created to change ineffective procedures or methods.

Data analysis techniques are designed to create an environment in which the developer and the user can effectively communicate in order to decide what information should be kept in the database.

6.11 Stages of Data Modelling

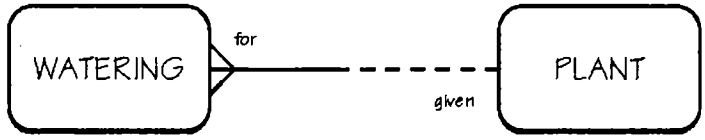


Fig. 6.17

The following stages of data modelling were adapted from the article "Making Models" by Richard Branton (see Sources of Information). They offer an organized strategy for developing a database program and fit well into the systems development life cycle.

6.11.1 Data Analysis

Determination of the data requirements, or what information is to be kept in the database.

- Identify what not how.
- Develop two data models:
- Object model

A business model that contains objects (and their relationships) that are important to the user. Input is provided by business experts and users. Storage and access methods are not considered in this phase. Some techniques used for this model are the entity-relationship diagram and the conceptual data model.

- Logical data model

The object model is transformed into logical data tables. As this model is developed, business policies, security, and audit control functions are added. The model is called logical because the data may or may not be used in a physical database table.

6.11.2 Database Design

Definition of the database structure, or how the data is stored.

- Create a physical data model from a logical data model.
- Add DBMS constraints.
- Determine the operating environment and performance considerations.
- Describe the data tables that will be used and their characteristics.

6.11.3 Database Construction

Implementation of the data to the DBMS program.

- Build a database from the physical data model.
- Describe the database information to the DBMS program.

6.12 Fundamentals of Object Modeling

The object model is a model of the business world of things and their associations. The objects and relationships become the essential model, defining what is important to the business.

Essential models are sometimes called business models, object models, logical models, or conceptual models. The terms object model and business model will be used throughout this course in describing the analysis phase.

The business model is not a computer model, but a business exercise. With the use of facilitation techniques, business users, called subject matter experts (SME's), provide facts about the objects, come to understand the model, and verify its accuracy. One facilitation technique used to accelerate the analysis process and to increase interaction between users and analysts is the Joint Application Design (JAD) session. This is a group session usually consisting of SME's, a sponsor, a facilitator, scribes, modeling experts, and observers. In the JAD session, participants determine user requirements and identify issues and opportunities. The environment is user-oriented with users creating the diagrams. The facilitator mediates the session and encourages the users, but does not correct or change the diagrams. Tools typically used in JAD sessions include:

- Focus statements and lists to determine the objects and relationships that should or should not appear in the model and the terms used.
- Entity-relationship (E-R) diagrams and E-R text, which are techniques used to diagram objects.
- Information requirements lists to identify user needs.
- Issues lists to identify important problems or issues.

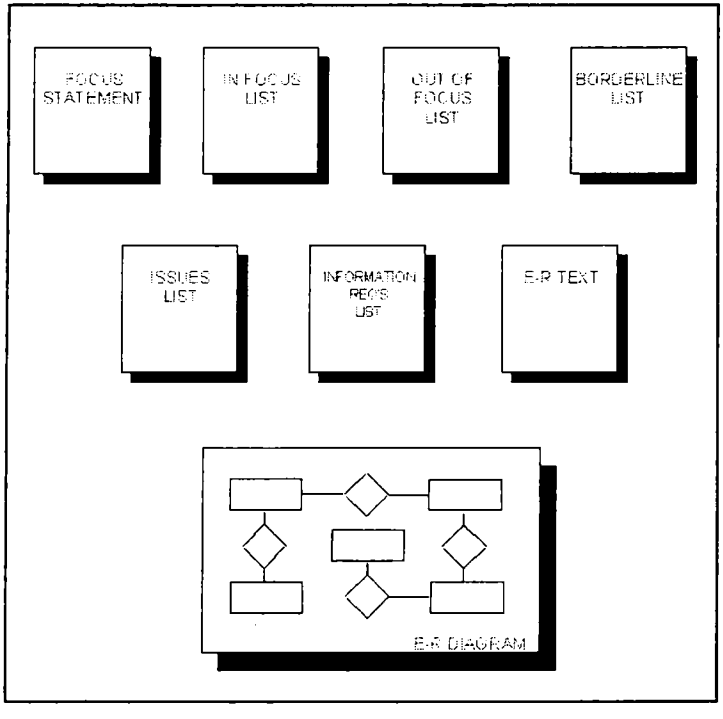


Fig. 6.18

In building the data model, the developer communicates directly with users. Sometimes the user can provide explicit answers, but often the developer must draw conclusions from the user's statements. From this communication, the developer makes decisions about what is to be stored in the database. The result of this joint effort between users and developers is a business model that accurately describes the business, and, consequently, becomes a corporate asset.

It is important to remember that object models are implementation-independent. They depict the essence of the system and what the system must do apart from how the system will or could be physically implemented.

6.12.1 Purpose of an Object Model

- Represents the relatively stable world of objects.
- Provides relatively stable data structures.
- Defines the world of objects and their relationships.
- Generates communication with the user.
- Provides an opportunity for new perspectives.

6.12.2 Benefits of Data Modeling

The benefits of data modelling in relation to the database as follows:

- Clear, factual data without ambiguity.
- Accuracy of data and facts as verified by the user.
- Ability to respond to ad hoc requests.
- Careful analysis of data results in less rework later.
- Allows evolution as the business scope expands.
- Technology independence allows easier implementation of models into changing technology.
- Application independence provides integration of databases with different applications.

In relation to development of the database, they are as follows:

- Step-by-step processes ensure that each issue is addressed appropriately.
- Less rework results in better productivity.
- The user participates while communicating with the developer in everyday language.
- Explicit questions asked by the developer can provide in-depth communication with the user.

6.13 Sources for a Data Model

- Business Rules
- Data dictionary
- Expert knowledge
- Use cases
- Patterns

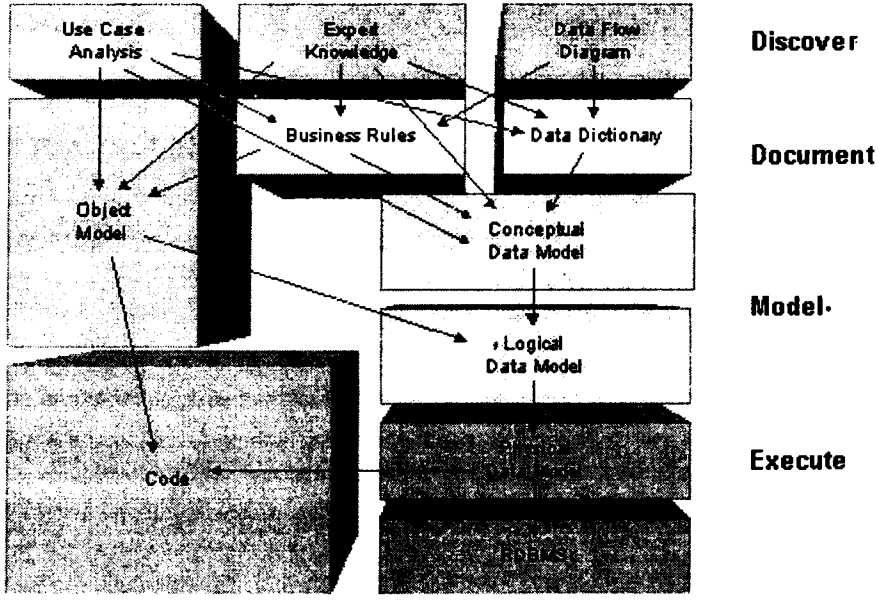


Fig. 6.19

6.13.1 Business Rules7

There is no formal science of business rules. They attempt to map business knowledge and policies into executable systems. "Business rule" is a misnomer. They include both things and their behaviors - "objects" and "rules".

A number of business and system modeling tools hold problems, requirements, notes, or other text in a catalog of business issues. See for example, **CASEwise, LBMS Systems Engineer, Silverrun, System Architect, Visible Workbench.**

A few data modeling tools (e.g., PowerDesigner) provide features to capture business rules and connect them to modeled objects.

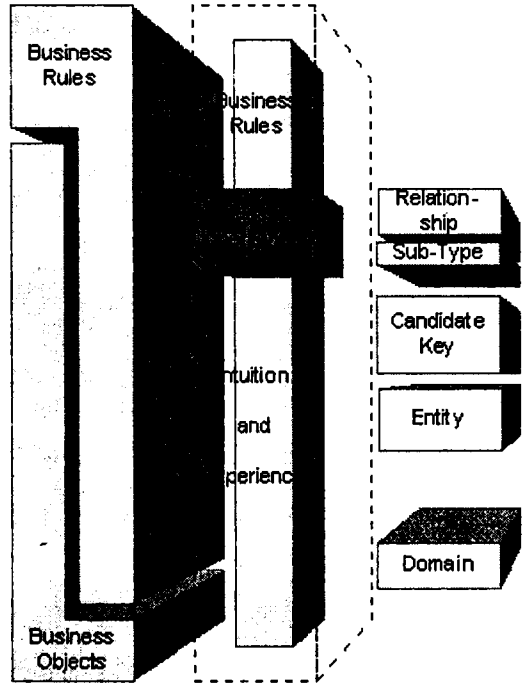


Fig. 6.20

Several types of business rules have been proposed and are useful for categorizing.

Business rules reveal entities, attributes, relationships and procedural constraints.

- **Types of Business Rules**

Several sources suggest these types:

- Definition – Justifying the existence of an entity or attibute and describing its meaning
- Fact – A connection or relationship between two object types
- Derivation – How instances of one object type can be derived from others
- Constraint – Limiting the population of entities or values of attributes

Recent writings by Ross and von Halle, as well as Object Role Modeling, establish these four categories of rules.

Since there is no formal science to business rules, these types are useful only for classification. Even those tools which support them (e.g., PowerDesigner) make little or no attempt to operate differently on the different types.

A fact type rule can express a complex relationship between what might become attributes and/or entities in an entity-relationship diagram. For example:

- An employee (entity) may have an assigned parking space (entity) or monthly parking allowance amount (attribute within employee) but not both.

While this statement is a rather simple valid business fact, it cannot be modeled on an entity-relationship diagram. Such rules are sometimes captured as text as pseudo-code for procedural solutions.

6.14 MODELING : Three Schema Architecture

In general, modelling is based on the three-schema architecture. A Schema is a abstract definition of reality. The levels are:

- Level 0: Real World
- Level 1: Conceptual Model - E-R Model or Object-Oriented Model
- Level 2: Implementation Model - Relational Model
- Level 3: Physical Model - Physical Data Structures.

In designing a database, we begin with the development of a conceptual model. A number of different conceptual modeling approaches are used including:

- Hierarchical (legacy)
- Network (legacy)
- Entity-Relationship (linked to relational model)
- Object-Oriented (generally converted to a relational model)

In this chapter, we shall concentrate on Entity - Relationship Model only.

6.15 Entity-relationship Model

An entity-relationship diagram is a snapshot of data structures. Entity-Relationship diagrams (ERD) emerged in the 1970's from work by Dr. Peter Chen and others. They were looking for means to simplify the representation of large and complex data storage concepts. It has many variations. CASE tools such as Oracle's Designer/2000; Powersoft's S-Designer tool; and the Information Engineering Facility (IEF) tool implement the E-R modeling approach in a variety of similar ways. There is no single standard diagramming approach, but there are a set of common constructs that can be learned that will enable you to use any E-R modeling CASE tool or understand any E-R modeling diagram.

Fig. 6.21

Since ERD relationships are between entities, the ERD is not capable of expressing intra-entity, inter- attribute relationships - i.e., within one entity. This can be done by the following:

- By certain extensions to the ERD in some methods
- With unmodeled text rules and constrains
- Using Object Role Modeling for the conceptual model

Model, which is most often used as a tool for communications between database designers and end users during analysis phase of database development process.

E-R model is a detailed, logical representation of the data for an organisation or for a business area, is expressed in terms of entities in the business environment, the relationships (or associations) among those entities, and the attributes.(or properties) of both entities and their relationships.

An E-R model is normally expressed as an entity-relationship diagram, which is a graphical representation of an E-R model.

6.15.1 Entity

Entity (sometimes called an entity class): - A collection of entities that share common properties or characteristics. Normally is represented by rectangles. For example:

Fig. 6.22

The next step in modeling a service or process is to identify the entities involved in that process. An entity is a thing or object of significance to the business, whether real or imagined, about which the business must collect and maintain data, or about which information needs to be known or held. An entity may be a tangible or real object like a person or a building; it may be an activity like an appointment or an operation; it may be conceptual as in a cost center or an organisational unit.

Whatever is chosen as an entity must be described in real terms. It must be uniquely identifiable. That is, each instance or occurrence of an entity must be separate and distinctly identifiable from all other instances of that type of entity.

For example, if we were designing a computerized application for the care of plants in a greenhouse, one of its processes might be tracking plant waterings. Within that process, there are two entities: the Plant entity and the Watering entity. A Plant has significance as a living flora of beauty. Each Plant is uniquely identified by its biological name, or some other unique reference to it. A Watering has significance as an application of water to a plant. Each Watering is uniquely identified by the date and time of its application.

6.15.2 Diagramming Entities

The entity symbol is a rectangle with the name at the top. The entity name must be descriptive and meaningful. An unambiguous text definition is important. The entity graphic symbol may differ slightly by CASE tool or author but the shape is unimportant.

The entity name is how people will refer to the entity. In the conceptual model an entity name should not be limited by RDBMS product limits - this is a generic name for the business. When physical DBMS object names are assigned, be sure to take into account the naming limits and reserved words of your targe platforms.

For example, what is a customer?

- Someone who has purchased?
- Any organisation or person who may purchase?
- Will this entity definition be clear two years later to a new team?

An Entity represents:

- A tangible thing, a real-world event, or any intangible concept:
  "Product", "Sales visit", "Customer class discount"
- A class of things, not any one instance.
  "Person" has instances of "Tom" and "Simone".

Entities are not:

- Independent or Dependent. Those terms apply only to the identification choice you make.
- Fundamental, Attributive, or Associative. Classifications have meaning only in a model context of entities and relationships.

An entity is an object that can be identified in the user's world and is important to the user's view of the business. An entity class is a collection of entities of the same type; this term is often used interchangeably with entity. Objects being classified as entities are independent and can be physical or conceptual.

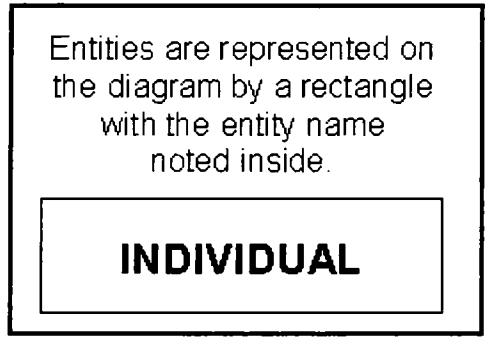


Fig. 6.23

*Entities are represented on the E-R diagram by rectangles with the entity name noted inside. *An Entity is something about which we store data for use by managers and workers. An entity can be either an item found in reality or an abstract concept. *Consider the example entities shown in Fig. 6.24 here: CUSTOMER_ORDER, CUSTOMER, PRODUCT.

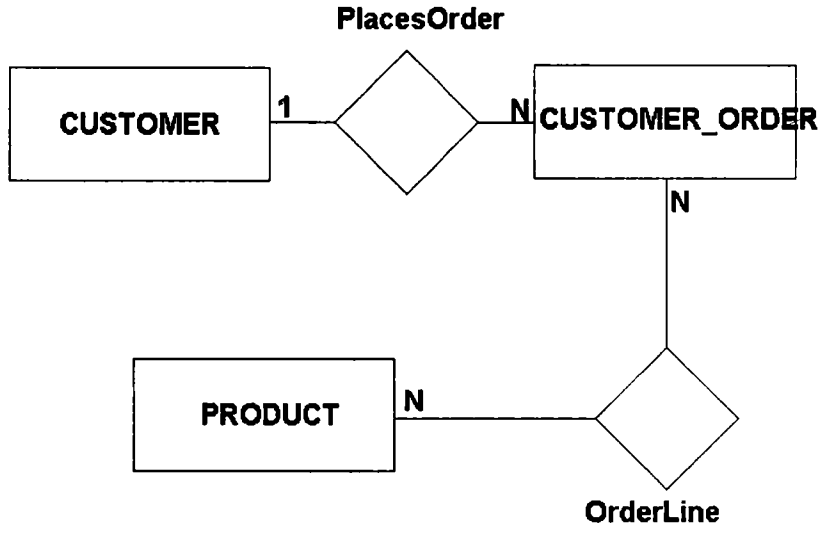


Fig. 6.24

- Distinguish between an entity and instance (or occurrence) of an entity for each of the entities shown in this example.
- Naming Conventions should follow a standard established by the work organisation. Here we capitalize the names of entities, but this may vary from one organisation to another.

Below are some of the entities depicted in the Student Registration E-R Diagram.

Individual Mercer University Advisor Macon Campus Student Enrolled Atlanta Campus Student Not Enrolled Douglas Center Campus Registration Procedure Class Registrar Office Classroom Confirmation Letter Course Fee Class Roster Enrollment Qualification Individual Record Financial Status

- An entity occurrence (or entity instance) is a particular instance of an entity (usually more than one instance). These occurrences should not be confused with attributes.

ENTITY	ENTITY OCCURRENCE
ADVISOR	Dr. John Brown
STUDENT ENROLLED	Mary Jones
COURSE	INSY312

Fig. 6.25

- The entity name should be a singular noun or noun phrase and should convey the meaning of the object appropriately.

Examples of entity names . . .

Singular noun or noun phrase
INDIVIDUAL not INDIVIDUALS
CLASS not CLASSES

Proper level of specificity

ENROLLMENT QUALIFICATION may be more appropriate than QUALIFICATION
INDIVIDUAL'S RECORD may be more appropriate than RECORD

Entities and Entity Sets

An entity is an object that exists and is distinguishable from other objects. For instance, John Harris with S.I.N. 890-12-3456 is an entity, as he can be uniquely identified as one particular person in the universe.

An entity may be concrete (a person or a book, for example) or abstract (like a holiday or a concept). An entity set is a set of entities of the same type (e.g., all persons having an account at a bank). Entity sets need not be disjoint. For example, the entity set employee (all employees of a bank) and the entity set customer (all customers of the bank) may have members in common.

An entity is represented by a set of attributes.

e.g. name, S.I.N., street, city for "customer" entity. The domain of the attribute is the set of permitted values (e.g. the telephone number must be seven positive integers).

Formally, an attribute is a function which maps an entity set into a domain. Every entity is described by a set of (attribute, data value) pairs. There is one pair for each attribute of the entity set. e.g. a particular customer entity is described by the set {(name, Harris), (S.I.N., 890-123-456), (street, North), (city, Georgetown)}. An analogy can be made with the programming language notion of type definition. The concept of an entity set corresponds to the programming language type definition. A variable of a given type has a particular value at a point in time. Thus, a programming language variable corresponds to an entity in the E-R model.

Fig. 6.26 shows two entity sets.

RELATIONSHIP	RELATIONSHIP OCCURRENCE
ADVISOR determines CLASS	Dr. John Brown determines INSY312
CLASS located in CLASSROOM	INSY430 located in Room 8
INDIVIDUAL registers for CLASS	Mary Jones registers for BUSN202

Fig. 6.26

We will be dealing with five entity sets in this section:

- Branch, the set of all branches of a particular bank. Each branch is described by the attributes branch-name, branch-city and assets.

- Customer, the set of all people having an account at the bank. Attributes are customer-name, S.I.N., street and customer- city.
- Employee, with attributes employee-name and phone-number.
- Account, the set of all accounts created and maintained in the bank. Attributes are account-number and balance.
- Transaction, the set of all account transactions executed in the bank. Attributes are transaction-number, date and amount.

6.15.3 Attributes

After you identify an entity, then you describe it in real terms, or through its attributes. An attribute is any detail that serves to identify, qualify, classify, quantify, or otherwise express the state of an entity occurrence or a relationship. Attributes are specific pieces of information which need to be known or held.

An attribute is either required or optional. When it is required, we must have a value for it, a value must be known for each entity occurrence. When it is optional, we could have a value for it, a value may be known for each entity occurrence. For example, some attributes for Plant are: description, date of acquisition, flowering or non-flowering, and pot size. The description is required for every Plant. The pot size is optional since some plants do not come in pots. Again, some of Watering's attributes are: date and time of application, amount of water, and water temperature. The date and time are required for every Watering. The water temperature is optional since we do not always check it before watering some plants.

The attributes reflect the need for the information they provide. In the analysis meeting, the participants should list as many attributes as possible. Later, they can weed out those that are not applicable to the application, or those the client is not prepared to spend the resources on to collect and maintain. The participants come to an agreement on which attributes belong with an entity, as well as which attributes are required or optional.

The attributes which uniquely define an occurrence of an entity are called primary keys. If such an attribute doesn't exist naturally, a new attribute is defined for that purpose, for example an ID number or code.

What is an Attribute?

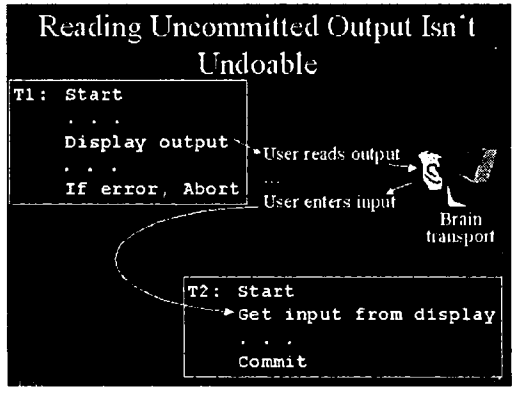


Fig. 6.27

The basic unit of information about any entity occurance. Eventally those attributes which migrate into the physical design become columns in database tables. Attributes may be local to the entity ("Name", "Address") or inherited by relationship from another entity. Synonomous and/or related terms are data element (as in a data flow diagram), column (of a database table), and data item.

Data item sometimes denotes a central, reusable definition versus an attribute which is an instance of that definition being used in some entity.

An attribute is an identifying or descriptive property of either an entity or a relationship. It has no meaning apart from the element that it describes. An attribute must appear in only one place in the model. If an attribute could be represented in several entities, it should be drawn as a separate entity. An attribute usually has many values and is described by a domain that clarifies its value type. Date formats and ranges, number ranges, and decimal places are examples of attribute domains.

Examples of Attributes

ENTITY	ATTRIBUTES
INDIVIDUAL	StudentID StudentName StudentAddress StudentPhone
INDIVIDUAL'S RECORD	StudentID Classes Completed Class Grade Major

Fig. 6.28

Attributes v/s Attribute Values

Attributes for the entity INDIVIDUAL:

ATTRIBUTES	ATTRIBUTE VALUES
StudentID	100 150 200
StudentName	Bob Jones Gail Smith Bill Wright
StudentPhone	449-6712 384-2386 280-4555

Fig. 6.29

An attribute is represented on the E-R diagram in one of two ways:

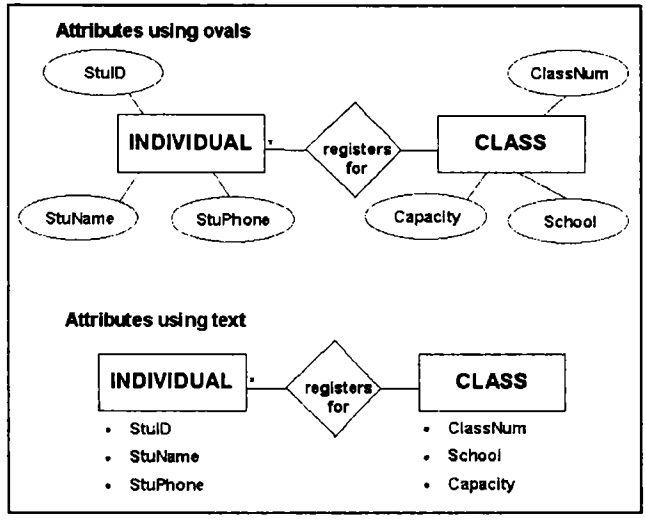


Fig. 6.30

- By ellipses (ovals) attached to the element it describes.
- By listing the attributes as text.

Relationship Between E-R Diagramming Elements

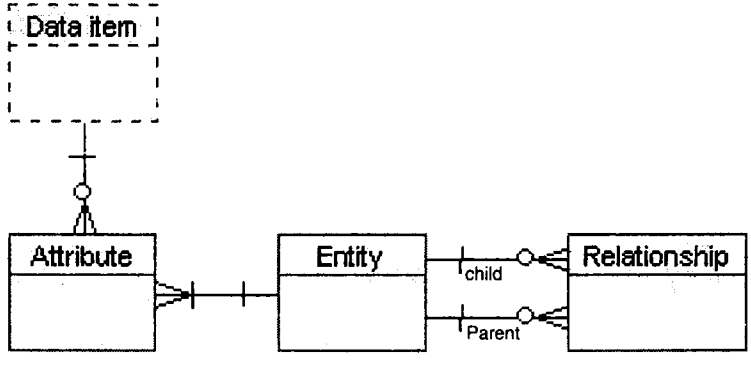


Fig. 6.31

6.15.4 Relationships

After two or more entities are identified and defined with attributes, the participants determine if a relationship exists between the entities. A relationship is any association, linkage, or connection between the entities of interest to the business; it is a two-directional, significant association between two entities, or between an entity and itself. Each relationship has a name, an optionality (optional or mandatory), and a degree (how many). A relationship is described in real terms.

Rarely will there be a relationship between every entity and every other entity in an application. If there are only two or three entities, then perhaps there will be relationships between them all. In a larger application, there are not always relationships between one entity and all of the others.

Assigning a name, an optionality, and a degree to a relationship helps confirm the validity of that relationship. If you cannot give a relationship all these things, then perhaps there really is no relationship at all. For example, there is a relationship between Plant and Watering. Each Plant may be given one or more Waterings. Each Watering must be for one and only one specific Plant.

Relationships is an association between the instances of one or more entity types that is of interest to the organisation. Relationships are Represented by diamond shape. For example:

Fig. 6.32

Owns is a M:N relationship between Customer and Product.

There are others basic symbols:

Primary Key

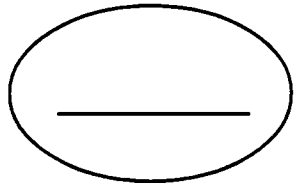


Fig. 6.33

Multi-valued attribute

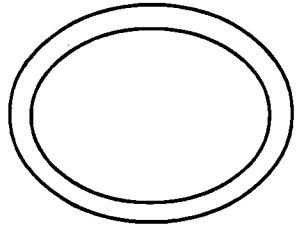


Fig. 6.34

E-R Diagram for SCT&S Database

What is a Relationship?

A connection, association, or rule among entities:

- "Customer places Sales Order"
- "Item occurs on Sales Order"

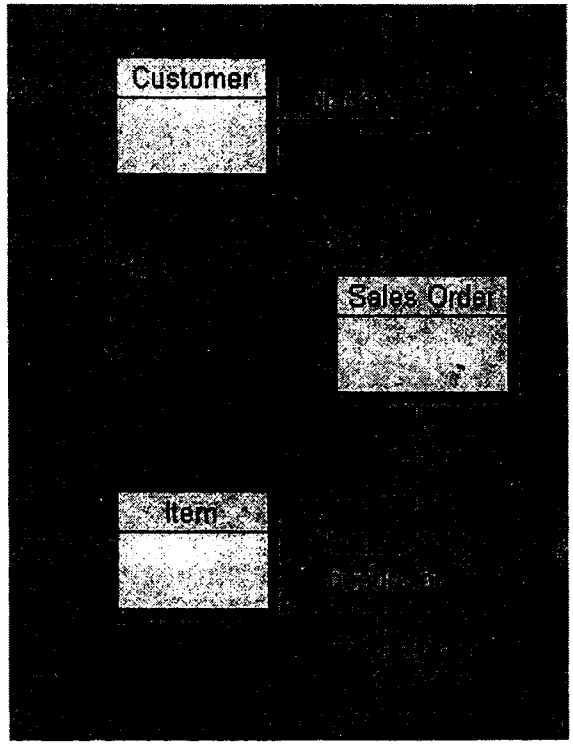


Fig. 6.35

In a conceptual model, it is sufficient to state or draw the relationship.

A logical model defines specific means of joining two entities via implied or expressed foreign keys.

Relationships can be classified into a few relationship types. Relationship Types

Relationships are grouped by their cardinality:

- One-to-Many is the only relational form.
- >99% of logical model

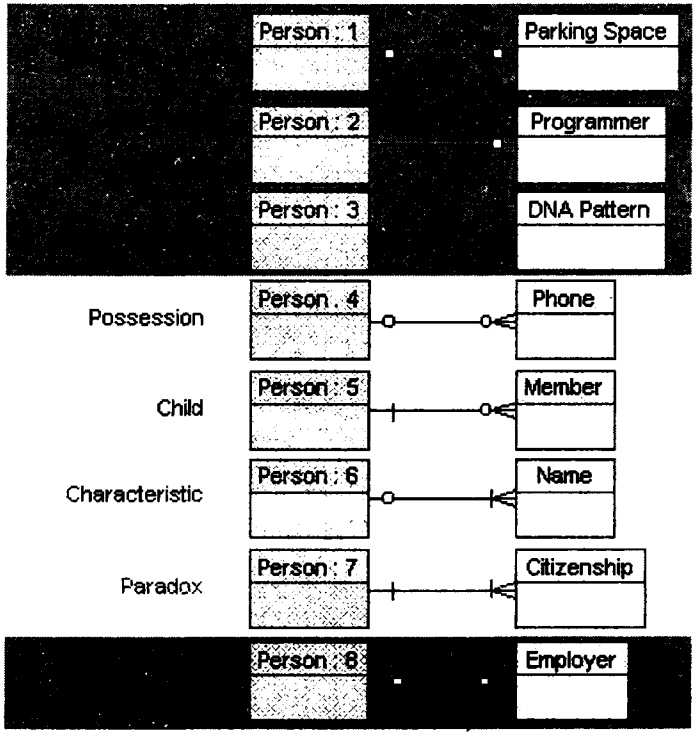


Fig. 6.36

6.15.5 Relationship Types

- One-to-One is a special case of One-to-Many;
- <1% of a logical model
- Many-to-Many becomes an associative entity

Diagramming Relationships

The relationship symbol is a line between two entites.

Define a relationship with:

- Predicate statements in one or both directions
- Unambiguous text description
- (A name is not important)
- Cardiality symbols at each end

Will the relationship be clear two years later to a new team?

Relationships are :

- Unambiguous, immutable expressions of business rules.
- Binary or unary in IE, SSADM, IDEF1X and OO methods.
- Logical objects. Relationships can be reattached, with their properties intact, to different entities.

Relationships are not:

- Identifying or Non-Identifying. Those apply only to entity identification.
- Information containers. If you sense a need for information about a "relationship" then it is an entity!
- DBMS objects. Relationships only define joins between entities.

6.15.6 Association in Relationship

Relationships show associations between entities. They also clarify entities. Relationship classes are associations among entity classes; this term is often used interchangeably with relationship. A relationship may or may not have attributes that need to be defined. Relationships can be binary (between two entities) or complex (involving more than two entities). Other terms used to describe complex relationships include ternary and unary.

RELATIONSHIP	RELATIONSHIP OCCURRENCE
ADVISOR determines CLASS	Dr. John Brown determines INSY312
CLASS located in CLASSROOM	INSY430 located in Room 8
INDIVIDUAL registers for CLASS	Mary Jones registers for BUSN202

- A relationship is represented on the E-R diagram by a diamond drawn between associated entities.

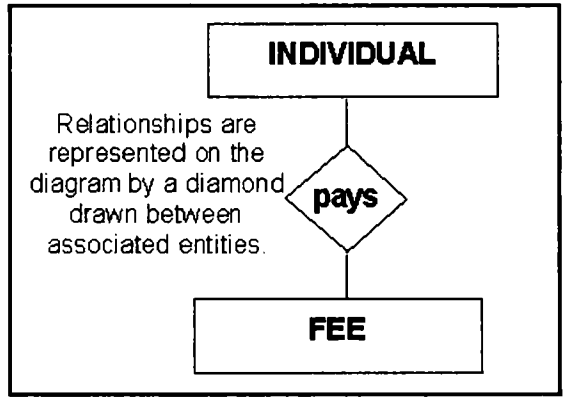


Fig. 6.37

- A relationship occurrence (or relationship instance) is actually an association between entity occurrences.

RELATIONSHIP	RELATIONSHIP OCCURRENCE
ADVISOR determines CLASS	Dr. John Brown determines INSY312
CLASS located in CLASSROOM	INSY430 located in Room 8
INDIVIDUAL registers for CLASS	Mary Jones registers for BUSN202

Fig. 6.38

- A relationship name should:
  - Be a verb or verb phrase.
  - The majority of cases can be covered with four basic verbs or phrases: known by, has, owned by, refers to.
  - Use correct noun-verb agreement.
  - sInclude all participating entities.

The relationship nickname should not include the entities. Establish one entity in the relationship as an anchor in order to properly name the relationship. Specify an anchor only affects how the relationship is read, not the nature of the relationship. Therefore, the anchor choice should be based on which form sounds best.

- **Examples of relationship names.**

Examples of proper noun-verb agreement . . . Yes . . .

INDIVIDUAL registers for CLASS REGISTRAR OFFICE processes ENROLLMENT QUALIFICATION

No . . .

INDIVIDUAL registration CLASS REGISTRAR OFFICE procedure ENROLLMENT QUALIFICATION

Examples of relationship names including all participating entities . . .

MACON OFFICE stores INDIVIDUAL'S RECORD ADVISOR approves REGISTRATION REGISTRAR OFFICE generates CLASS ROSTER

Relationship nickname does not include entities.

- stores
- approves
- generates

Examples of naming anchor entities.

Consider:

MACON OFFICE stores INDIVIDUAL'S RECORD

- or -

INDIVIDUAL'S RECORD is stored by MACON OFFICE

Note that the first example sounds best; therefore, it should probably be designated as the anchor entity. Anchor designation can be depicted by placing an asterisk (*) on the diagram next to the anchor entity.

6.15.7 Relationship Notations

There are many notation styles for relationships. There are no standards for relationship style. Different styles are read in different directions. But they all express the same information.

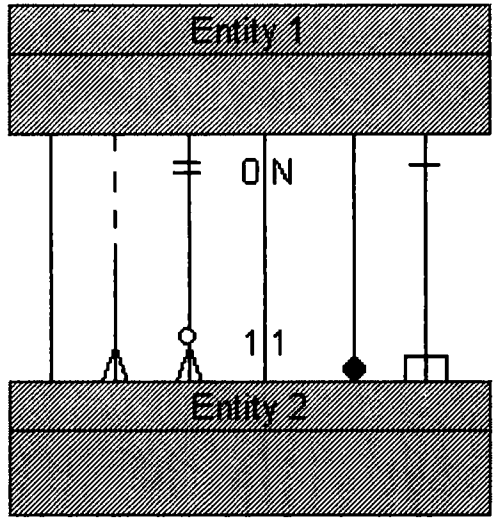


Fig. 6.39

6.15.8 Relationship Cardinality

Cardinality specifies the number of instances which may be involved in each entity of a relationship. Most methods show the Boolean abstract, not the absolute number, becasue this determines the relationship type.

Fig. 6.40

6.15.9 Relationships & Relationship Sets

A relationship is an association between several entities. A relationship set is a set of relationships of the same type. Formally it is a mathematical relation on $n \geq 2$ (possibly non- distinct) sets. If $E_1, E_2, \ldots, E_n$ are entity sets, then a relationship set R is a subset of

$$\{(e_1, e_2, \ldots, e_n) \mid e_1 \in E_1, e_2 \in E_2, \ldots, e_n \in En\}$$

Fig. 6.41

where $\{(e_1, e_2, \ldots, e_n) \mid e_1 \in E_1, e_2 \in E_2, \ldots, e_n \in En\}$ is a relationship.

For example, consider the two entity sets customer and account. We define the relationship CustAcct to denote the association between customers and their accounts. This is a binary relationship set. Going back to our formal definition, the relationship set CustAcct is a subset of all the possible customer and account pairings. This is a binary relationship. Occasionally there are relationships involving more than two entity sets. The role of an entity is the function it plays in a relationship. For example, the relationship works for ordered pairs of employee entities. The first employee takes the role of manager, and the second one will take the role of worker.

A relationship may also have descriptive attributes. For example, date (last date of account access) could be an attribute of the CustAcct relationship set.

6.15.10 Populating Attributes

For each entity, ask "What properties does this thing have - even if nothing else exists around it?"

- A person has age – even the last person on earth.
- A building has height – even if it is abandoned.
- A song has a key, even if it is unsung.

6.15.11 Domains

A domain (in relational usage) is a set of values and operations which may be used to populate and operate on one or more columns. The values may be specified by list or formula. While is not yet any theoretical or practical way to limit the operations applied to a domain, this example shows the need. Sometimes you can reveal entities by looking in the data dictionary for homeless attributes:

- "To whom does atomic weight belong?
- And what about the year in which an element was discovered?"
- Aggregate atomic weight and year into anew entity
- Called Atomic Element.

Cross-check data elements captured in the data dictionary from data flow diagrams, use cases, or other analysis.

- All data structures and data elements
- Discovered in analysis must be

- Accounted for in the logical model.

Entities often occur in hierarchies - family trees related by inheritance.
This is sub-typing or specialisation and generalisation - the same as building OO class structures.
Each child entity inherits all attributes and relationships from its parent.

6.15.12 Looking for Hierarchies

Entities often occur in hierarchies - family trees are related by inheritance.
This is sub-typing or specialisation and generalisation - the same as building OO class structures.
Each child entity inherits all attributes and relationships from its parent.

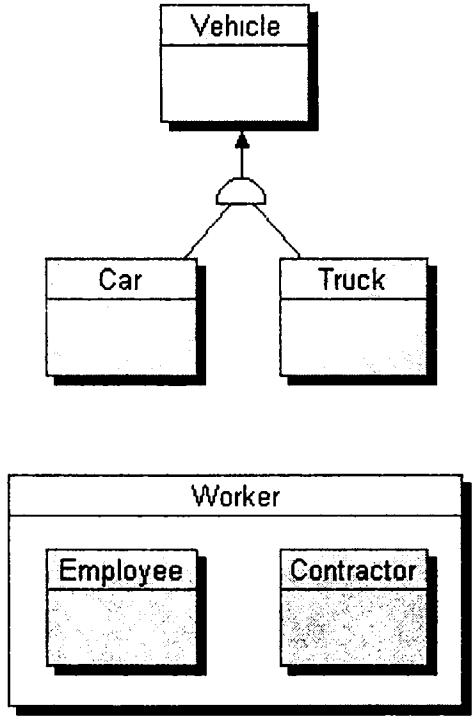


Fig. 6.43

We define properties at their highest level in the hierarchy to avoid redundancy. In a conceptual model, we ignore how inheritance operates. Later we want to specify how super-and sub-types map from the logical model to physical structures.

6.15.13 Generalisation

Hierarchies let us locate attributes and relationships at the appropriate level.
All vehicles have:

- VIN and Registration
- Owner

- The attributes and relationship are generalized to all vehicles.

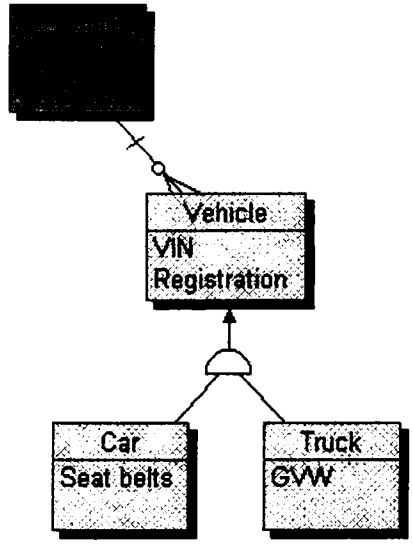


Fig. 6.44

6.15.14 Specialisation

- Only cars have primary drivers and seat belts.
- Only trucks have gross vehicle weight.

These attributes and relationships are specialized to the child level.

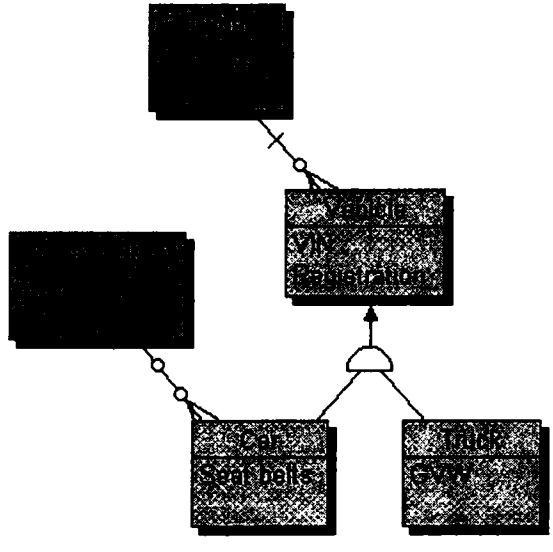


Fig. 6.45

6.15.15 Specialisation of Relationships

Optional parent relationships usually hide a need for specialisation. Ask your self:

- Is the relationship sometimes true for each instance? Then it is correctly modeled as optional.
- Is the relationship always true for some instances? Then it requires a specialisation.

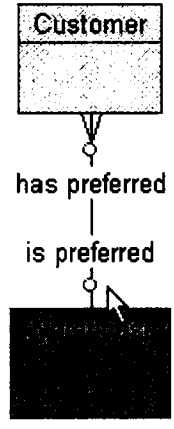


Fig. 6.46

Some customers must have a default warehouse.

For others it does not exist (in this model).

By splitting the Customer entity into two sub-types, we can model the relationship to Warehouse precisely.

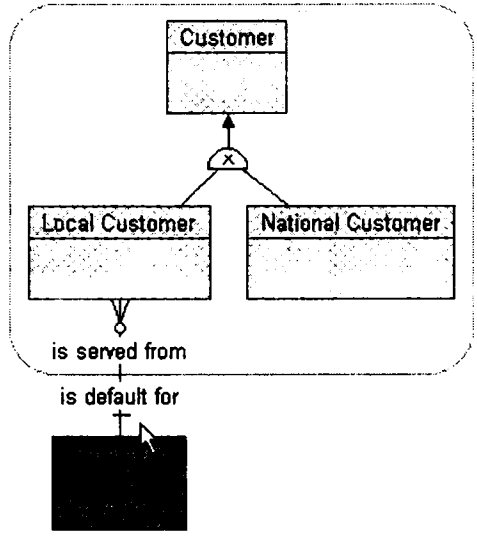


Fig. 6.47

6.15.16 Attributes

It is possible to define a set of entities and the relationships among them in a number of different ways. The main difference is in how we deal with attributes.

Consider the entity set employee with attributes employee-name and phone-number. We could argue that the phone be treated as an entity itself, with attributes phone-number and location. Then we have two entity sets, and the relationship set EmpPhn defining the association between employees and their phones. This new definition allows employees to have several (or zero) phones. New definition may more accurately reflect the real world. We cannot extend this argument easily to making employee-name an entity. The question of what constitutes an entity and what constitutes an attribute depends mainly on the structure of the real world situation being modeled, and the semantics associated with the attribute in question.

6.15.17 Mapping Constraints

An E-R scheme may define certain constraints to which the contents of a database must conform.

Mapping Cardinalities: express the number of entities to which another entity can be associated via a relationship. For binary relationship sets between entity sets A and B, the mapping cardinality must be one of:

- **One-to-one:** An entity in A is associated with at most one entity in B, and an entity in B is associated with at most one entity in A.
- **One-to-many:** An entity in A is associated with any number in B. An entity in B is associated with at most one entity in A.
- **Many-to-one:** An entity in A is associated with at most one entity in B. An entity in B is associated with any number in A.
- **Many-to-many:** Entities in A and B are associated with any number from each other.

The appropriate mapping cardinality for a particular relationship set depends on the real world being modeled. (Think about the CustAcct relationship.)

Existence Dependencies: if the existence of entity X depends on the existence of entity Y, then X is said to be existence dependent on Y. (Or we say that Y is the dominant entity and X is the subordinate entity.)

For example, Consider account and transaction entity sets, and a relationship log between them. This is one-to-many from account to transaction. If an account entity is deleted, its associated transaction entities must also be deleted. Thus account is dominant and transaction is subordinate.

6.15.18 Keys

Differences between entities must be expressed in terms of attributes. A superkey is a set of one or more attributes which, taken collectively, allow us to identify uniquely an entity in the entity set. For example, in the entity set customer, customer-name and S.I.N. is a superkey. Note that customer-name alone is not, as two customers could have the same name.

A **superkey** may contain extraneous attributes, and we are often interested in the smallest superkey. A superkey for which no subset is a superkey is called a candidate key.

In the example above, S.I.N. is a candidate key, as it is minimal, and uniquely identifies a customer entity.

A **primary key** is a candidate key (there may be more than one) chosen by the DB designer to identify entities in an entity set. An entity set that does not possess sufficient attributes to form a primary key is called a weak entity set. One that does have a primary key is called a strong entity set. For example,

The entity set transaction has attributes transaction-number, date and amount. Different transactions on different accounts could share the same number. These are not sufficient to form a primary key (uniquely identify a transaction). Thus transaction is a weak entity set. For a weak entity set to be meaningful, it must be part of a one- to-many relationship set. This relationship set should have no descriptive attributes.

The idea of strong and weak entity sets is related to the existence dependencies seen earlier.

Member of a strong entity set is a dominant entity. Member of a weak entity set is a subordinate entity. A weak entity set does not have a primary key, but we need a means of distinguishing among the entities. The discriminator of a weak entity set is a set of attributes that allows this distinction to be made. The primary key of a weak entity set is formed by taking the primary key of the strong entity set on which its existence depends (see Mapping Constraints) plus its discriminator.

To illustrate:

- Transaction is a weak entity. It is existence-dependent on account.
- The primary key of account is account-number.
- Transaction-number distinguishes transaction entities within the same account (and is thus the discriminator). * So the primary key for transaction would be (account-number, transaction-number).

Just Remember: The primary key of a weak entity is found by taking the primary key of the strong entity on which it is existence-dependent, plus the discriminator of the weak entity set.

6.15.19 Primary Keys for Relationship Sets

The attributes of a relationship set are the attributes that comprise the primary keys of the entity sets involved in the relationship set.

For example:

- S.I.N. is the primary key of customer, and
- Account-number is the primary key of account.
- The attributes of the relationship set custacct are then (account-number, S.I.N.).
- This is enough information to enable us to relate an account to a person.

If the relationship has descriptive attributes, those are also included in its attribute set. For example, we might add the attribute date to the above relationship set, signifying the date of last access to an account by a particular customer.

Note that this attribute cannot instead be placed in either entity set as it relates to both a customer and an account, and the relationship is many-to-many.

The primary key of a relationship set R depends on the mapping cardinality and the presence of descriptive attributes.

With no descriptive attributes:

- **Many-to-many:** all attributes in R.
- **One-to-many:** primary key for the "many" entity.
- Descriptive attributes may be added, depending on the mapping cardinality and the semantics involved.

6.15.20 Relationship Cardinality

The cardinality of a relationship shows the number of entity occurrences possible between entities in a relationship. The purposes of cardinality are to determine reality, identify different user perceptions, and to define meaning. Cardinality can also be used to enforce rules. In order to determine relationship cardinality, the modeler should question the user.

Cardinality shows the maximum number of occurrences that can participate in a relationship (as represented by "one" or "many") and can be graphically expressed in several ways. However, in this text it is shown as one (1) or many (M or N) outside the entity rectangle. The types of cardinality are:

- 1:1 (one-to-one) relationships: A single occurrence of one entity relates to a single occurrence of another entity.
- 1:N (one-to-many) relationships: A single occurrence of one entity relates to many occurrences of another entity.
- N:M (many-to-many) relationships: Many occurrences of one entity relate to many occurrences of another entity.

6.15.21 Determining relationship cardinality

Traditionally, textbooks describe cardinality in a one-dimension format. These formats are:

- 1:1 (one-to-one) relationships
- 1:N (one-to-many) relationships
- N:M (many-to-many) relationships

A 1:N relationship can be graphically represented as:

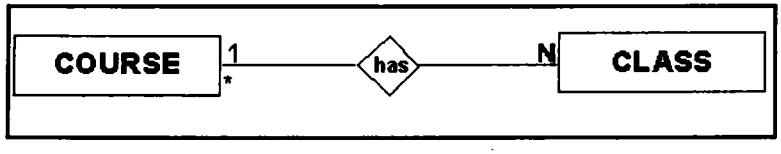


Fig. 6.48

This diagram can be read as "one course has many classes." However, the relationship is actually two-dimensional; it must be clear when read from either direction. For example, when read from the opposite direction, one class has one course. Consider the following example of an alternative diagram that may illustrate the relationship more clearly.

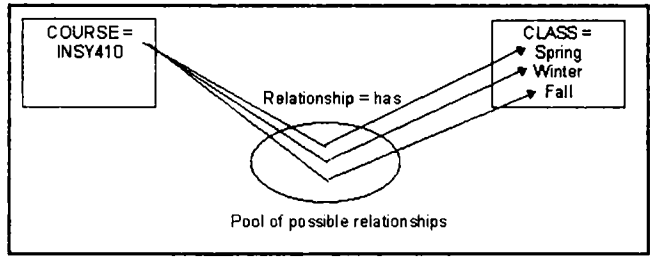


Fig. 6.49

Note that the entity occurrence INSY410 has three relationship occurrences with the entity CLASS. Furthermore, one relationship occurrence of "has" is defined by one entity occurrence from COURSE and one entity occurence from CLASS. Finally, one entity occurrence from CLASS (for example, "Spring") is only related to one occurrence of COURSE. This last relationship tells the reader that "In the spring INSY 410 is offered only one time."

The following diagram (in the typical E-R format) shows how the three objects "COURSE," "CLASS," and "has" participate in describing the business environment.

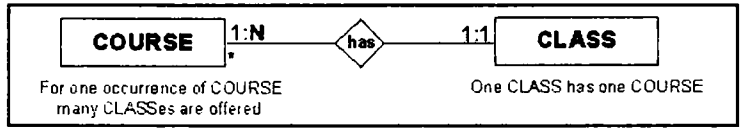


Fig. 6.50

Note: The above examples depict the relationships only in the specified context. The focus of the user and the business actually determines the real meaning and cardinality in a relationship.

6.15.22 Relationship Modality

Modality indicates the minimum number of entity occurrences that must participate in order for the relationship to exist. In other words, modality states that in a particular relationship a specific entity either must have an occurrence (mandatory) or may not have an occurrence (optional). In the E-R diagram, a bar is drawn on the relationship line to illustrate a mandatory occurrence, while a circle is drawn for an optional occurrence. Modality can be mandatory on both sides of the relationship, optional on both sides, or mandatory on one side and optional on the other. Modality can also be used to enforce rules.

6.16 Entities Attributes and Relation (EAR) Models

The EAR model is one of the most common and successful types of data model around. Its basic elements are called Entities, Attributes and Relationships, hence the name EAR model, or occasionally EAR model.

The modeling constructs are as follows:

1. **Entity :** An entity is any 'thing' about which data can be stored. For example, if the system needs to store data about customers or products, then the model would have customer or product entities. Although the definition is stated in terms of what must be stored, in fact it is retrieval of the data that is the fundamental requirement.
2. **Attributes :** The attributes of an entity are those facts that need to be stored about the entity. For example, the attributes of a customer might include the account number, name, address and credit limit.
3. **Relationships :** Relationships exist between various entities within a system. For example, there may be a relationship between the customer and an order.
4. **Normalisation :** For such a data model to be considered valid it must conform to a set of rules. A valid data model is one that is fully normalized and the process of converting an invalid model into a valid one is called normalisation.

The aim of normalisation is to ensure that each fact is only recorded in one place so that facts cannot be inconsistent and the performance of updates cannot produce anomalies by updating one copy of the fact but not another. The rules may be expressed in more formal terms, but they are not considered suitable in a book at this level.

Each occurrence of an entity, for example each individual customer, must be uniquely identifiable by means of a key containing one or more attributes. the customer's full name or account number might serve as a key, for example. Other attributes (non-key attributes) may be regarded as fact relating to the key can be obtained.

The process of normalisation ensures that in each entity of the final model every non-key attribute is a fact about the key, the whole key and nothing but the key.

To round off this description of data models here (Fig.. 6.51) is a diagram of a simple EAR model concerning customers making orders for products.

```
| CUSTOMER Account No.,
main adore,
credit limit,
credit) | Place order
| CUSTOMER ORDER
(Order number,
account number,
date) |
```

```
Contains details
| ORDER DETAIL
(Order number,
product code,
No. requisite) |
Is ordered |
PRODUCT
(Product code,
product description) |
```

The Entity Relationship Diagram

We can express the overall logical structure of a database graphically with an E-R diagram. Its components are:

- Rectangles representing entity sets.
- Ellipses representing attributes.
- Diamonds representing relationship sets.
- Lines linking attributes to entity sets and entity sets to relationship sets.

In the text, lines may be directed (have an arrow on the end) to signify mapping cardinalities for relationship sets. Fig. 6.52 to 6.53 show some examples.

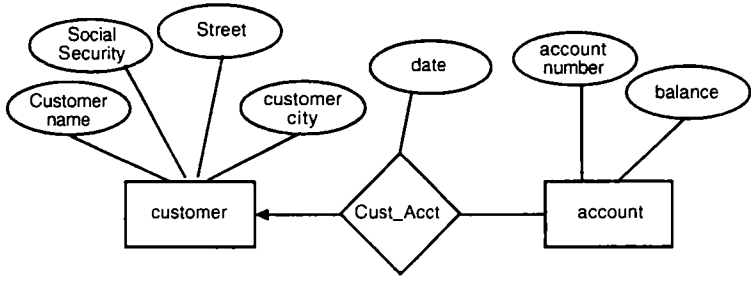


Fig. 6.52: One-to-many from customer to account

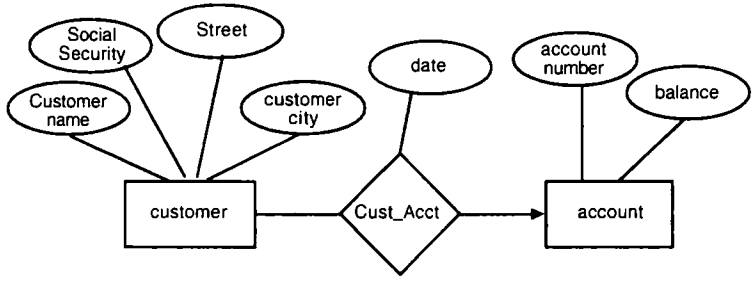


Fig. 6.53: Many-to-one from customer to account

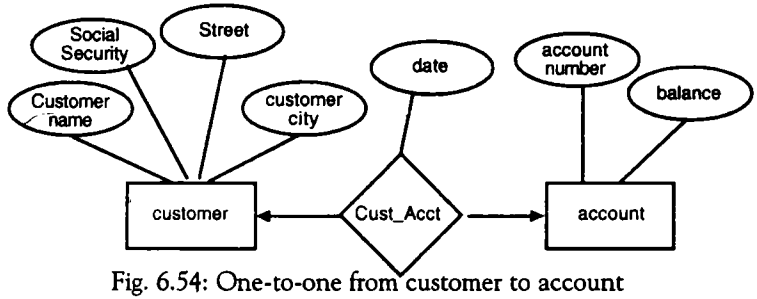


Fig. 6.54: One-to-one from customer to account

The arrow positioning is simple once you get it straight in your mind, so do some examples. Think of the arrow head as pointing to the entity that "one" refers to.

6.17 Entity Relationship Diagrams

To visually record the entities and the relationships between them, an entity relationship diagram, or ERD, is drawn. An ERD is a pictorial representation of the entities and the relationships between them. It allows the participants in the meeting to easily see the information structure of the application. Later, the project team uses the ERD to design the database and tables. Knowing how to read an ERD is very important. If there are any mistakes or relationships missing, the application will fail in that respect. Although somewhat cryptic, learning to read an ERD comes quickly.

The informal definition of Entity-Relationship (ER) diagrams is directly taken from in order to demonstrate the expressiveness of our specification language and the reasoning capabilities of our DL system. Fig. 6.57 shows an example ER diagram specifying a relationship between a pilot and an aircraft. We assume a few primitive concepts (denoted in slanted font) and spatial relations (touching, containing, linked_with, text_value) representing geometrical objects (rectangle, circle, diamond, line, text) and their relationships.

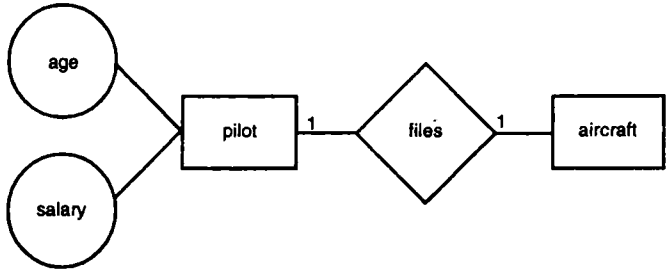


Fig.. 6.57: A simple entity-relationship diagram

- **Connectors**

A relationship-entity connection is a line that touches exactly one text label (expressing cardinality) and two other regions (rectangle or diamond). A cardinality is a text string with values chosen from the set ER6.GIF.

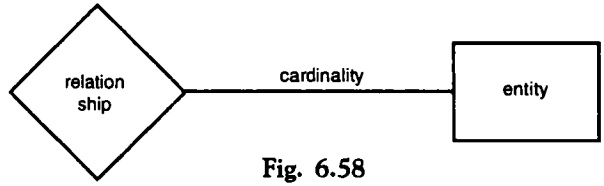


Fig. 6.58

An attribute-entity connection is a line that touches only two regions (circle or rectangle) and no text string.

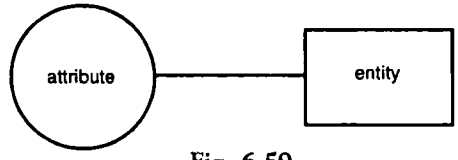


Fig. 6.59

attribute_entity ≡
(*line* ∧ ($\exists_{=2}$ touching) ∧
(∀ touching (*circle* ∨ *rectangle*)) ∧
($\exists_{=1}$ touching *rectangle*) ∧ ($\exists_{=1}$ touching *circle*);

Fig. 6.60

- **Entities**

An entity is a rectangle that contains its name. It touches at least one relationship-entity and optionally some attribute- entity connectors. It is linked with at least one diamond.

named_region ≡
(*region* ∧ ($\exists_{=1}$ containing) ∧ (∀ containing *text*))

entity ≡
(*rectangle* ∧ named_region ∧
($\exists_{\geq 1}$ touching relationship_entity) ∧
(∀ touching (attribute_entity ∨ relationship_entity)) ∧
($\exists_{\geq 1}$ linked_with *diamond*) ∧
(∀ linked_with (*circle* ∨ *diamond*)))

Fig. 6.61

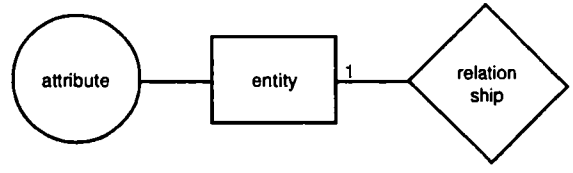


Fig. 6.62

- **Relationships**

A relationship is a diamond that contains its name. It touches one relationship-entity and optionally some attribute-entity connectors. It is linked with two entities.

relationship ≡
(*diamond* ∧ named_region ∧
($\exists_{=2}$ linked_with) ∧ (∀ linked_with entity) ∧
($\exists_{=2}$ touching) ∧ (∀ touching relationship_entity) ∧
($\exists_{\leq 2}$ touching (= (touching ∘ text_value) 1)) ∧
($\exists_{\leq 1}$ touching (= (touching ∘ text_value) m)) ∧
($\exists_{\leq 1}$ touching (= (touching ∘ text_value) n)))

Fig. 6.63

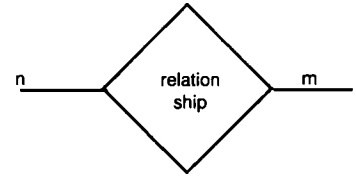


Fig. 6.64

- **Attributes**

An attribute is a circle that contains its name. It touches one attribute-entity connector and is linked with an entity.

attribute ≡
(*circle* ∧ named_region ∧
($\exists_{=1}$ linked_with) ∧ (∀ linked_with entity))

Fig. 6.65

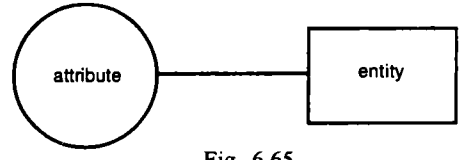


Fig. 6.65

Each entity is drawn in a box. Each relationship is drawn as a line between entities. The relationship between Plant and Watering is drawn on the ERD as follows:

Fig. 6.67

Since a relationship is between two entities, an ERD shows how one entity relates to the other, and vice versa. Reading an ERD relationship means you have to read it from one entity to the other, and then from the other to the first. Each style and mark on the relationship line has some significance to the relationship and its reading. Half the relationship line belongs to the entity on that side of the line. The other half belongs to the other entity on the other side of the line.

When you read a relationship, start with one entity and note the line style starting at that entity. Ignore the latter half of the line's style, since it's there for you to come back the other way. A solid line at an entity represents a mandatory relationship. In the example above, each Watering must be for one and only one Plant. A dotted line at an entity represents an optional relationship. Each Plant may be given one or more Waterings.

The way in which the relationship line connects to an entity is significant. If it connects with a single line, it represents one and only one occurrence of that entity. In the example, each Watering must be for one and only one Plant. If the relationship line connects with three prongs, i.e., a crowsfoot, it represents one or more of the entities. Each Plant may be given one or more Waterings. As long as both statements are true, then you know you have modeled the relationship properly.

In the relationship between Plant and Watering, there are two relationship statements. One is: each Watering must be for one and only one Plant. These are the parts of the ERD which that statement uses:

Fig. 6.68

The second statement is: each Plant may be given one or more Waterings. The parts of the ERD which that statement uses are:

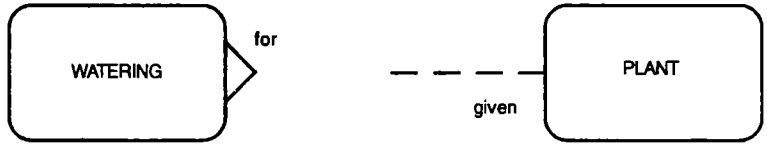


Fig. 6.79

After some experience, you learn to ask the appropriate questions to determine if two entities are related to each other, and the degree of that relationship. After agreeing on the entities and their relationships, the process of identifying more entities, describing them, and determining their relationships continues until all of the services of the application have been examined. The data model remains software and hardware independent.

6.18 Other Styles of E-R Diagram

The text uses one particular style of diagram. Many variations exist.

Some of the variations you will see are:

- Diamonds being omitted - a link between entities indicates a relationship.
- Less symbols, clearer picture.
- What happens with descriptive attributes?
- In this case, we have to create an intersection entity to possess the attributes.
- Numbers instead of arrowheads indicating cardinality.

- Symbols, 1, n and m used.
- E.g. 1 to 1, 1 to n, n to m.
- Easier to understand than arrowheads.

A range of numbers indicating optionality of relationship. e.g (0,1) indicates minimum zero (optional), maximum 1. Can also use (0,n), (1,1) or (1,n). Typically used on near end of link - confusing at first, but gives more information. E.g. entity 1 (0,1) — (1,n) entity 2 indicates that entity 1 is related to between 0 and 1 occurrences of entity 2 (optional). Entity 2 is related to at least 1 and possibly many occurrences of entity 1 (mandatory). Multivalued attributes may be indicated in some manner. Means attribute can have more than one value. For examples hobbies. Has to be normalized later on. Composite attributes. Derived attributes. Subclasses and superclasses. Generalisation and specialisation.

6.18.1 Roles in E-R Diagrams

The function that an entity plays in a relationship is called its role. Roles are normally explicit and not specified.

They are useful when the meaning of a relationship set needs clarification.

For example, the entity sets of a relationship may not be distinct. The relationship works-for might be ordered pairs of employees (first is manager, second is worker).

In the E-R diagram, this can be shown by labelling the lines connecting entities (rectangles) to relationships (diamonds).

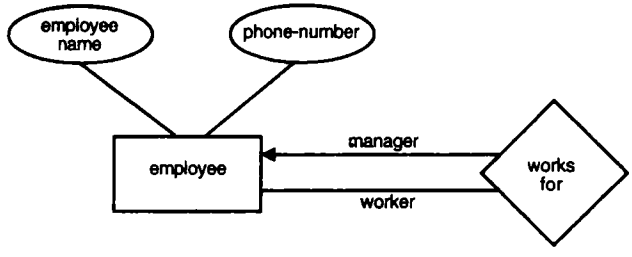


Fig. 6.70: E-R Diagram with Role Indicators

6.18.2 Weak Entity Sets in E-R Diagrams

A weak entity set is indicated by a doubly-outlined box. For example, the previously-mentioned weak entity set transaction is dependent on the strong entity set account via the relationship set log.

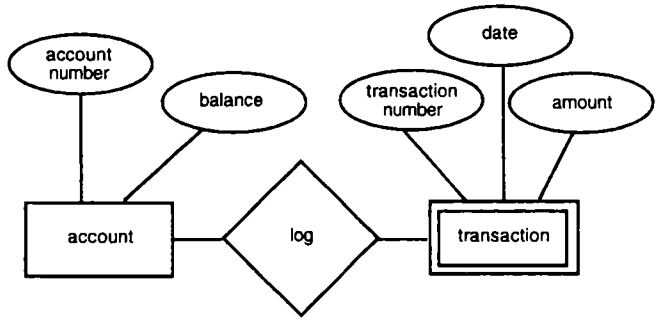


Fig. 6.72 Shows this Example.

Fig. 6.72: E-R Diagram with a weak entity set.

6.18.3 Nonbinary Relationships

Non-binary relationships can easily be represented. Fig. 6.73 shows an example.

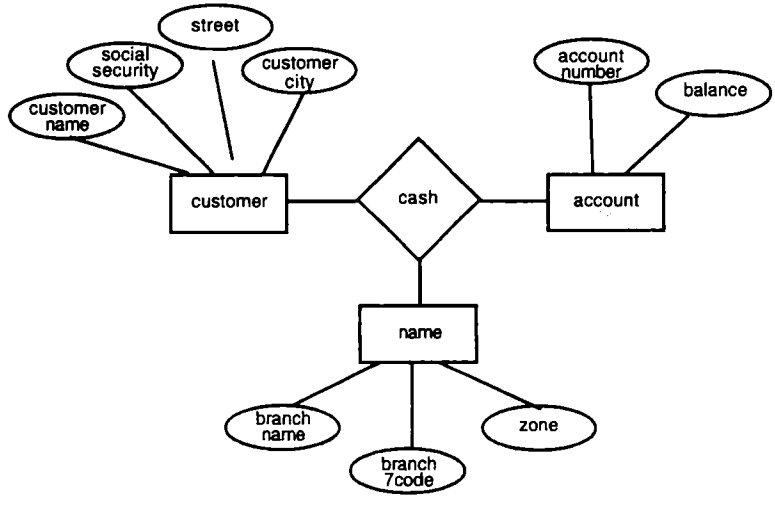


Fig. 6.73: E-R Diagram with a Ternary Relationship

This E-R diagram says that a customer may have several accounts, each located in a specific bank branch, and that an account may belong to several different customers.

6.18.4 Reducing E-R Diagrams to Tables

A database conforming to an E-R diagram can be represented by a collection of tables. We will use the E-R diagram of Fig. 6.74) as our example.

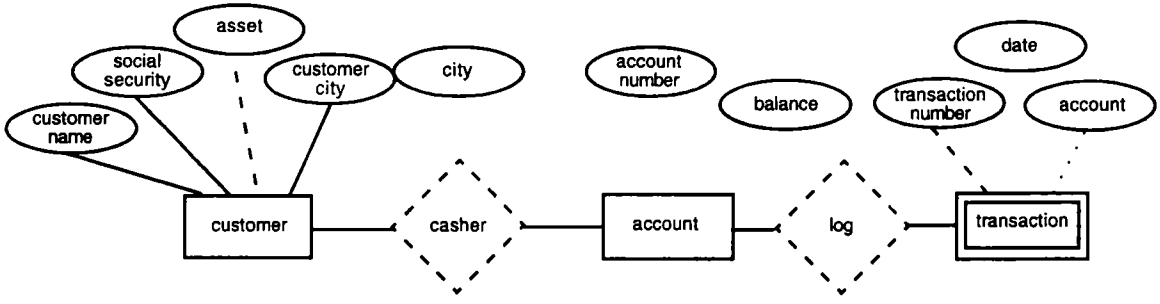


Fig. 6.74: E-R Diagram with strong and weak entity sets

For each entity set and relationship set, there is a unique table which is assigned the name of the corresponding set. Each table has a number of columns with unique names.

6.18.5 Representation of Strong Entity Sets

We use a table with one column for each attribute of the set. Each row in the table corresponds to one entity of the entity set. For the entity set account. We can add, delete and modify rows (to reflect changes in the real world). A row of a table will consist of an n-tuple where n is the number of attributes.

Actually, the table contains a subset of the set of all possible rows. We refer to the set of all possible rows as the cartesian product of the sets of all attribute values.

We may denote this as

$$D_1 \times D_2 \text{ or } \times_{i=1}^{2} D_i$$

For the account table, where D_i and a6.gif denote the set of all account numbers and all account balances, respectively. In general, for a table of n columns, we may denote the cartesian product of a6.gif by

$$\times_{i=1}^{n} D_i$$

6.18.6 Representation of Weak Entity Sets

For a weak entity set, we add columns to the table corresponding to the primary key of the strong entity set on which the weak set is dependent.

For example, the weak entity set transaction has three attributes: transaction-number, date and amount. The primary key of account (on which transaction depends) is account-number.

6.18.7 Representation of Relationship Sets

Let R be a relationship set involving entity sets $E_1, E_2, \ldots, E_m$

The table corresponding to the relationship set R has the following attributes:

$$\bigcup_{i=1}^{m} \text{primary-key}(E_i)$$

Fig. 6.75

If the relationship has k descriptive attributes, we add them too:

$$\bigcup_{i=1}^{m} \text{primary-key}(E_i) \bigcup \{a_1, a_2, \ldots, a_k\}$$

Fig. 6.76

An example:

The relationship set CustAcct involves the entity sets customer and account. Their respective primary keys are S.I.N. and account-number. CustAcct also has a descriptive attribute, date.

Linking a Weak to a Strong Entity

These relationship sets are many-to-one, and have no descriptive attributes. The primary key of the weak entity set is the primary key of the strong entity set it is existence-dependent on, plus its discriminator.

The table for the relationship set would have the same attributes, and is thus redundant.

6.18.8 Generalisation

Consider extending the entity set account by classifying accounts as being either savings-account or chequing-account.

Each of these is described by the attributes of account plus additional attributes. (savings has interest-rate and chequing has overdraft-amount.)

We can express the similarities between the entity sets by generalisation. This is the process of forming containment relationships between a higher-level entity set and one or more lower-level entity sets.

In E-R diagrams, generalisation is shown by a triangle, as shown in Figure 6.77.

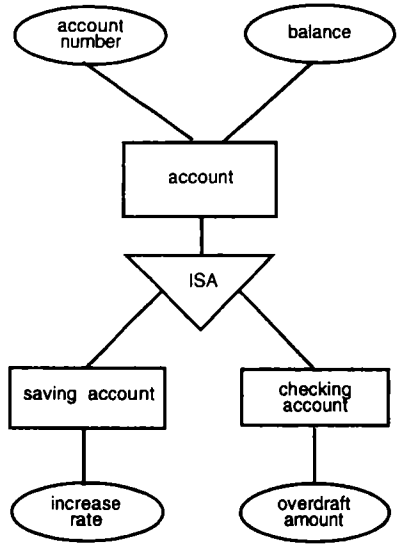


Fig. 6.77: Generalisation

Generalisation hides differences and emphasizes similarities. Distinction made through attribute inheritance. Attributes of higher-level entity are inherited by lower-level entities.

Two methods for conversion to a table form:

- Create a table for the high-level entity, plus tables for the lower-level entities containing also their specific attributes.
- Create only tables for the lower-level entities.

6.18.9 Aggregation

The E-R model cannot express relationships among relationships.

When would we need such a thing?

Consider a DB with information about employees who work on a particular project and use a number of machines doing that work. We get the E-R diagram shown in Fig. 6.78.

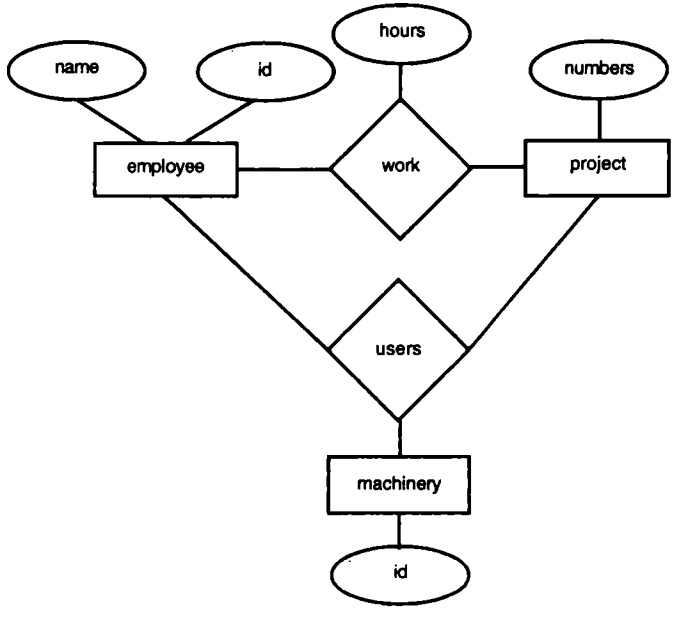


Fig. 6.78: E-R diagram with redundant relationships

Relationship sets work and uses could be combined into a single set. However, they shouldn't be, as this would obscure the logical structure of this scheme. The solution is to use aggregation.

An abstraction through which relationships are treated as higher- level entities. For our example, we treat the relationship set work and the entity sets employee and project as a higher-level entity set called work.

Fig. 6.79 shows the E-R Diagram with Aggregation.

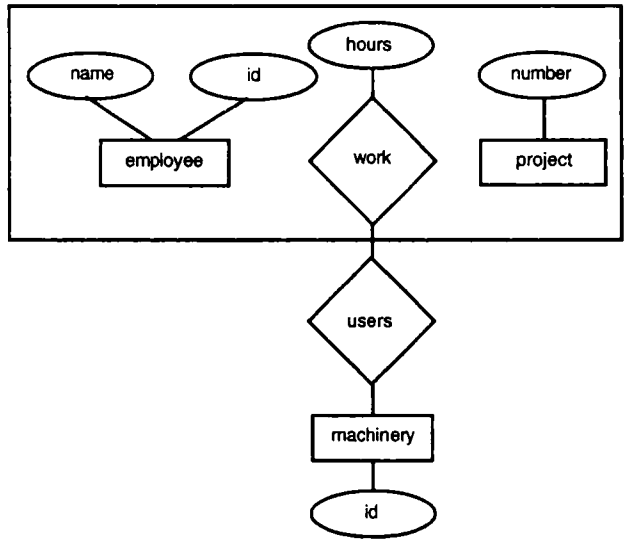


Fig. 6.79: E-R Diagram with Aggregation

Transforming an E-R diagram with aggregation into tabular form is easy. We create a table for each entity and relationship set as before. The table for relationship set uses contains a column for each attribute in the primary key of machinery and work.

6.18.10 Design of an E-R Database Scheme

The E-R data model provides a wide range of choice in designing a database scheme to accurately model some real-world situation.

Some of the decisions to be made are:

- Using a ternary relationship versus two binary relationships.
- Whether an entity set or a relationship set best fit a real- world concept.
- Whether to use an attribute or an entity set.
- Use of a strong or weak entity set.
- Appropriateness of generalisation.
- Appropriateness of aggregation.

6.18.11 Mapping Cardinalities

The ternary relationship of Fig. 6.79 could be replaced by a pair of binary relationships, as shown in Fig. 6.80.

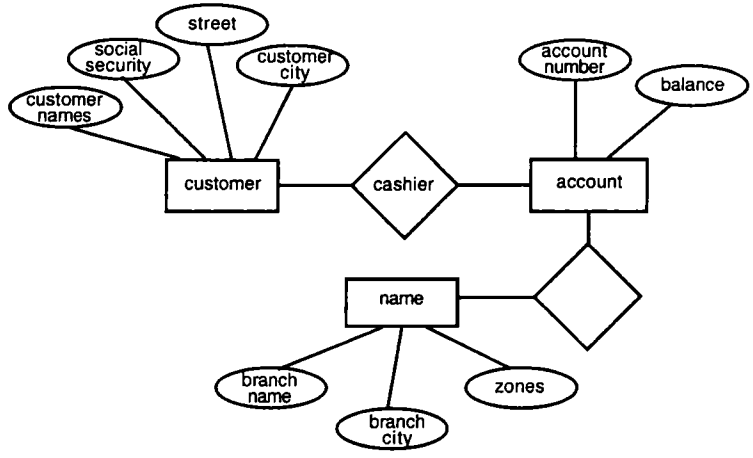


Fig. 6.80: Representation of Fig. 6.76 using binary relationships

However, there is a distinction between the two representations:

- In Fig. 6.76, relationship between a customer and account can be made only if there is a corresponding branch.
- In Fig. 6.85, an account can be related to either a customer or a branch alone.
- The design of Fig. 6.81 is more appropriate, as in the banking world we expect to have an account relate to both a customer and a branch.
- Use of Entity or Relationship Sets
- It is not always clear whether an object is best represented by an entity set or a relationship set.
- Both Fig. 6.81 and Fig. 6.85 show account as an entity.
- Fig. 6.83 shows how we might model an account as a relationship between a customer and a branch.

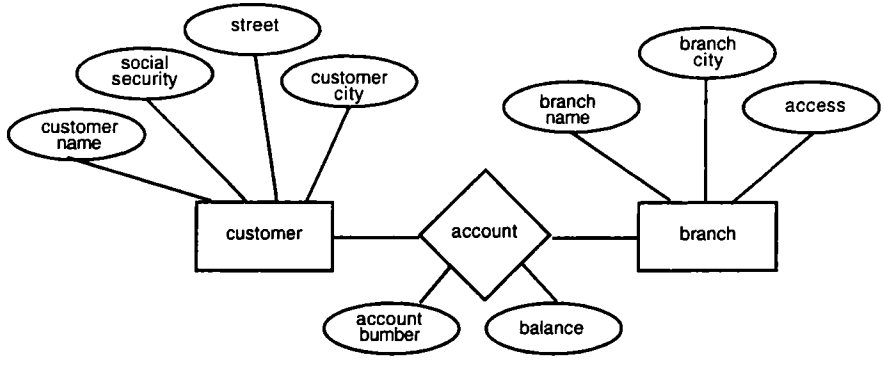


Fig. 6.81: E-R Diagram with Account as a Relationship set

This new representation cannot model adequately the situation where customers may have joint accounts. (Why not?) If every account is held by only one customer, this method works.

6.18.12 Use of Extended E-R Features

We have seen weak entity sets, generalisation and aggregation. Designers must decide when these features are appropriate.

Strong entity sets and their dependent weak entity sets may be regarded as a single "object" in the database, as weak entities are existence-dependent on a strong entity. It is possible to treat an aggregated entity set as a single unit without concern for its inner structure details.

Generalisation contributes to modularity by allowing common attributes of similar entity sets to be represented in one place in an E-R diagram. Excessive use of the features can introduce unnecessary complexity into the design.

6.19 The Data Dictionary

A data dictionary is a list of formal definitions for:

- Complex data structures
- Entities
- Domains
- Atomic data items

The dictionary comes from data flow diagrams, use cases, or other analyses.

It can be in hand written notes, document files, or a repository tool.

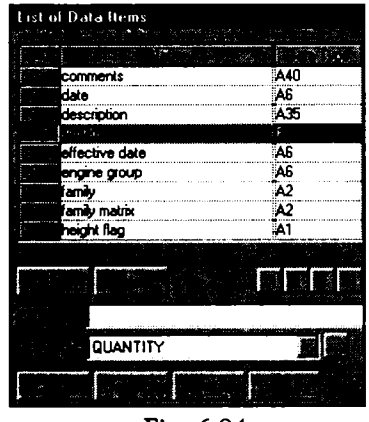


Fig. 6.84

6.20 Modeling from Expert Knowledge

- Traditional systems analysis often relies on personal knowledge without formal methods.
- Every data model is primarily the work of experts who understand the problem space.

This expertise should be:

- Treasured for its irreplacable insight and experience
- Used to supplement (not replace) formal techniques
- Carefully cross-checked against other sources
- Thoroughly documented for source and reasoning

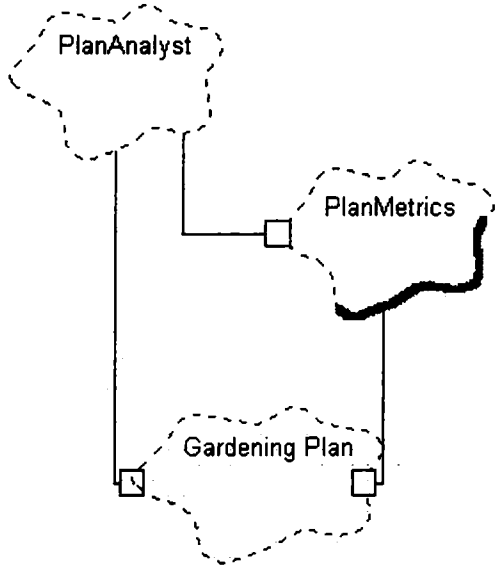


Fig. 6.85

6.21 Entity Patterns

Data models reveal familiar patterns such as:

- Recursion
- Look-up tables
- Multiple children
- Bill of materials

Learn to recognize patterns and reapply past solutions to familiar problems.

There are published sources which contain extensive collections of model patterns. See David Hay, Data Model Patterns, Conventions of Thought.

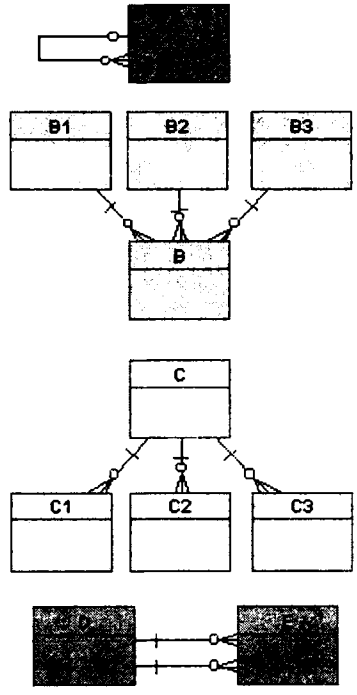


Fig. 6.86

6.22 Evolving the Logical Model

- Normalize structures
- Populate attributes
- Aggregate data items into new entities
- Nominate candidate keys
- Re-Normalize on the new candidate keys

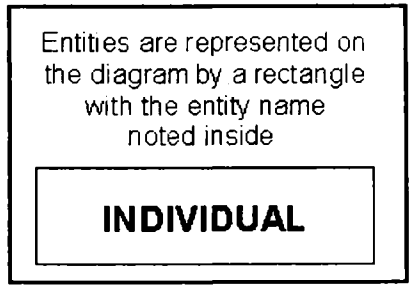


Fig. 6.87

6.22.1 Normalizing to a Logical Model

- Every raindrop, every snowflake, every hailstone
- Has a single speck of dust at the core.
- Every logical entity has a single idea at its core.

The essence of normalisation is one entity = one idea:

- A customer is a person or organisation who buys from us.
- A service order holds one customer request for service.

Examine complex data structures for hidden entities in:

Nouns - tangible or intangible

Adjectives whose value is one of a known list ... female |male; green | yellow | red; 6' | 8' | 10'

Embedded ideas which can exist on their own populate the entities with attributes ...

6.22.2 Relational Keys

The term key is much abused and misunderstood.

A key of a relation is any combination of columns whose values uniquely identify each tuple (row) in that relation.

A key is not an index and an index is not a key. This is a lingering confusion from non-relational methods.

Relational keys are more formally (and awkwardly) called Candidate Keys.

6.22.3 What are Candidate Keys?

A candidate key is any set of one table's columns whose combined value is unique throughout that table.

In the U.S. each state has a unique code - one candidate key.

Each state name is also unique - another candidate key. And so is the order of admission to the union.

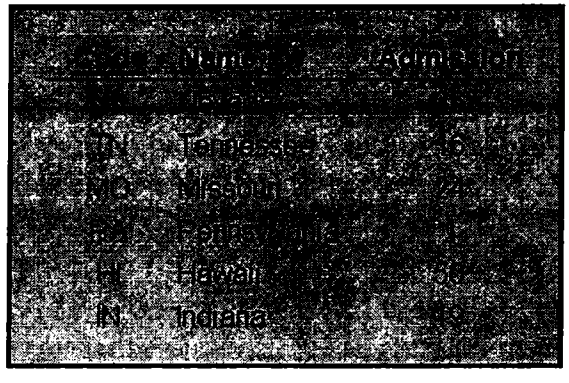


Fig. 6.88

Since both code and name are unique, code and name together are also unique. That's another candidate key - seven with all the combinations. - As a candidate for selection as the one identifier or primary key.

A candidate key usually holds the core idea inside an entity:

- This state table is about states, which are known by their names.

A candidate key always expresses a business rule of uniqueness:

- Every state has a unique state code for mailing.
- A table or entity with no candidate key is probably not normalized. and almost certainly not useful in an information system.
- A candidate key is unique.
- Social security number is unique. Were you born with one?
- A candidate key's value must exist. It cannot be null.
- Your driver's license number is unique. Can it change?
- The value of a candidate key must be stable. It's value cannot change outside the control of the system.
- The value of a candidate key is unique, extant, and stable.

6.22.4 Re-Normalize on the Candidate Key

After at least one candidate key has been noted, every attribute and relationship of the entity must be tested -

- Does this property depend solely and completely
- On the candidate key?
- If not, move the property (normalize it) to the entity where
- It depends solely and completely on the candidate key.

6.22.5 Entity Identifiers

We choose one candidate key of an entity as the identifier. Since every candidate key is unique, extant, and stable, choosing one candidate as the identifier is optional . An identifier is not required for a valid entity but it is very hard to use the resulting table without some candidate key.

Code	Name	Admission
NV	Nevada	36
TN	Tennessee	16
MO	Missouri	24
PA	Pennsylvania	1
HI	Hawaii	50
IN	Indiana	19

Fig. 6.89

We can select and join on any candidate key with equally valid results. So why name an identifier?

When the logical model transforms into a physical model, an entity's identifier evolves into a table's primary key. The DBMS uses primary keys to maintain referential integrity.

6.22.6 Selecting an Entity Identifier

Since any candidate key can be an identifier, how do you choose one? Many say to use a natural key. What is a natural key?

Picabo Street – the American downhill ski racer – did not have a name until she was three. Auto manufacturers had to invent the VIN because there was no natural key (datawise) for a car. No key is natural, except perhaps atomic number.

- Technical Keys

Many designers prefer technical keys, also called pseudo, surrogate, serial, or synthetic keys. Code, name, and serial are all synthetic keys – invented identifiers.

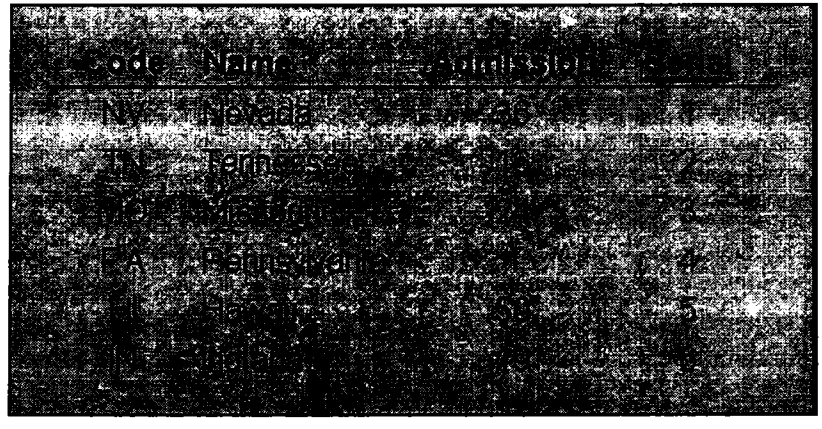


Fig. 6.90

Since 12 other states would argue that Pennsylvania was not the first state, admission is a questionable candidate key. Technical keys are often serial numbers assigned by the system. Some RDBMS products provide an automatic serial number data type. Some SA/SD authors insist you should never use technical keys.

Most OO authors insist you should always use technical keys (OIDs)

Use technical keys when they work in your model!

6.22.7 Child Key Options

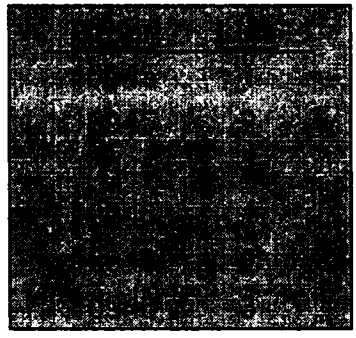


Fig. 6.91

Should a child inherit the identifier of its parent? Parent key and child sub-key are concatenated.

- Key segments may seem more "natural".
- Fewer joins are required for selects.
- The key gets longer at each child level.

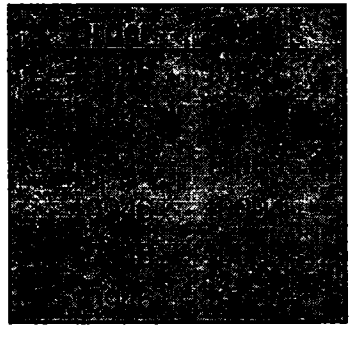


Fig. 6.92

Should every table have its own serial key?

* Every key is the same construction -
* Very valuable for generic structures.
* More joins are required to select.

Both work. Experiment in your model.

Merging Foreign Keys

(also known as Folding or Unifying)

6.23 Transforming from Logical to Physical

In this session we will not cover the tasks of transforming a logical data model into a physical model specialized for:

* Target RDBMS product(s)
* Site conFig.uration(s)
* Most CASE data modeling products provide some degree of logical to physical model mapping.
* Few CASE products satisfy DBAs who are responsible for keeping your database running.

6.24 Creating an Object Model

Creating an object model requires a great deal of planning and communication as well as a variety of tools and techniques. Since tools and techniques are ineffective without proper input, users who know the business extremely well (called SME's, or subject matter experts) should be selected to participate in the modeling process. Close communication between SME's and designers is crucial on a database design project. Although all designers may not use the same methods, for this course the following stages of object modeling will be explored:

- Define the focus of the model.
- Create an entity-relationship diagram.
- Verify information and data with the user.
- Build a data dictionary.

6.24.1 Defining the Focus

Before a diagram can be created, users and developers must understand the focus of the model. The purpose of focus is to determine which objects must be diagrammed and to decide what they will be called. Focus can be determined by asking a few questions, such as:

- As modeling begins, general guidelines and perspectives should be set. With these in mind, focus lists can help to identify statements that are within focus, out of focus, and borderline. Determining out of focus statements eliminates some areas from further consideration, while borderline statements help to improve focus. During this exercise, information requirements and issues are recorded as they surface.

6.24.2 Developing an E-R Diagram

The E-R technique was introduced by Peter Chen in 1976 as a tool for modeling data. The diagram is not a substitute for the basic relational (table-structured) model, which will be discussed in a later lesson, but provides a foundation for that model. Following is one possible development strategy for developing an E-R model:

- Identify entities.
- Identify relationships between entities.
- Identify attributes of entities.
- Identify attributes of relationships.
- Graphically represent all entities, relationships, and attributes in an E-R diagram.

The purpose of the diagram is to encourage dialogue between users and designers and to provide a graphic structure of the users' data requirements. During a diagramming session, the designer must keep discussions on a nontechnical level. Focus lists, information requirements lists, and issues lists can continue to help define user requirements.

In sessions or interviews, the requirements can be stated in simple sentences. With the following guidelines, these sentences can be easily converted into E-R diagramming structures:

- Nouns become entities.
- Verbs become relationships.
- Adjectives become attributes of entities.
- Adverbs become attributes of relationships.
- Phrases imply cardinality and modality.

George Tillman has provided an English-to-ER conversion chart that is very helpful in determining E-R structures from everyday language. For example, during sessions facilitators ask questions to encourage simple responses about business objects. "How does a student register for a course?" can lead to "An advisor registers a student" or "The registrar's office registers a student." These sentences can be converted to an E-R diagram. During this exercise, the facilitator guides the group's conversation, but draws what the user perceives.

English to E-R conversion chart

Source: "Building a Logical Data Model," by George Tillman, DBMS Fundamentals, July 1995, pg. 70.

What to look for: E-R Component: Common Noun Transitive Verb Gerund Adjective Adverb Proper Entity Relationship Associative Entity Proper Entity Attribute Relationship Attribute (Associative Entity) Words such as: many at least one only one at most Cardinality Words such as: must can may not Modality

Words such as: and but Conjunction Words such as: or either nor neither, nor Exclusion

6.24.3 Veryfying Data

Equally important in the diagramming process is iteration. During sessions, the facilitator often restates and questions the users' statements, sometimes from a slightly different perspective, for understanding and verification. If the diagram changes, the facilitator repeats the process, leading users to examine objects and relationships until they are confident that the diagram is accurate. Developers should ask detailed questions of users, and users should provide their own lists of questions.

Designing an object model is not an exact science. Business facts are often complex, and reality may be difficult to define. At times facilitators and designers may need to "read between the lines." However, a user's perception of the model can be influenced by the way the designer portrays objects. Iteration helps to ensure that the model reflects the user's view.

6.24.4 Defining the Data Dictionary

We have already learnt a lot about data dictionary. The data dictionary provides information about objects that may be hard to represent in a diagram. Nevertheless, this text is as important

as the diagram itself. Use of an accurate, complete, and current data dictionary allows data and programs to remain independent and flexible. A complete data dictionary contains data structure descriptions, user data, views, tables, indexes, users, properties, application data, and access privileges. However, at the object modeling level, the dictionary only provides information and rules about objects. Other terms used to described the dictionary include metadata, data directory, systems database, systems catalog, and a database of the database.

The data dictionary contains additional information about objects on the model. It might contain the following:

- Entity Name
- Description
- Type
- Attributes FINANCIAL STATUS
- An enrollment qualification.
- Entity Subtype
- Loans
- Grants
- Scholarships
- Tuition Reimbursement
- Relationship Name
- Description
- Type
- Entities involved
- Cardinality
- Modality INDIVIDUAL pays FEE
- To determine the fees paid by a student.
- Binary
- INDIVIDUAL, FEE
- One individual can pay many fees
- One type of fee can be paid by many individuals.
- Mandatory-optional

6.24.5 Many-to-Many Relationships

There are different types of relationships. The greenhouse plant application example showed a one-to-many and a many-to-one relationship, both between Plant and Watering. Two other relationships commonly found in data models are one-to-one and many-to-many. One-to-one relationships are between two entities where both are related to each other, once and only once for each instance of either. In a many-to-many relationship, multiple occurrences of one entity are related to one occurrence of another, and vice versa.

An example of a many-to-many relationship in the greenhouse plant application is between the Plant and Additive entities. Each plant may be treated with one or more Additives. Each Additive may be given to one or more Plants. The ERD for this relationship is shown below.

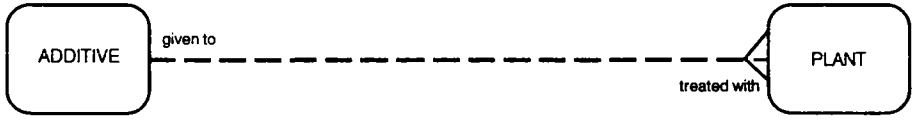


Fig. 6.93

Many-to-many relationships cannot be directly converted into database tables and relationships. This is a restriction of the database systems, not of the application. The development team has to resolve the many-to-many relationship before it can continue with the database development. If you identify a many- to-many relationship in your analysis meeting, you should try to resolve it in the meeting. The participants can usually find a fitting entity to provide the resolution.

To resolve a many-to-many relationship means to convert it into two one-to-many, many-to-one relationships. A new entity comes between the two original entities, and this new entity is referred to as an intersection entity. It allows for every possible matched occurrence of the two entities. Sometimes the intersection entity represents a point or passage in time.

The Plant-Additive many-to-many relationship above is resolved in the following ERD diagram:

Fig. 6.94

With these new relationships, Plant is now related to Treatment. Each Plant may be given one or more Treatments. Each Treatment must be given to one and only one Plant. Additive is also related to Treatment. Each Additive may be used in one or more Treatments. Each Treatment must be comprised of one and only one Additive. With these two new relationships, Treatment cannot exist without Plant and Additive. Treatment can occur multiple times, once for each treatment of a plant additive. To keep each Treatment unique, a new attribute is defined. Treatment now has application date and time attributes. They are the unique identifiers or the primary key of Treatment. Other attributes of Treatment are quantity and potency of the additive.

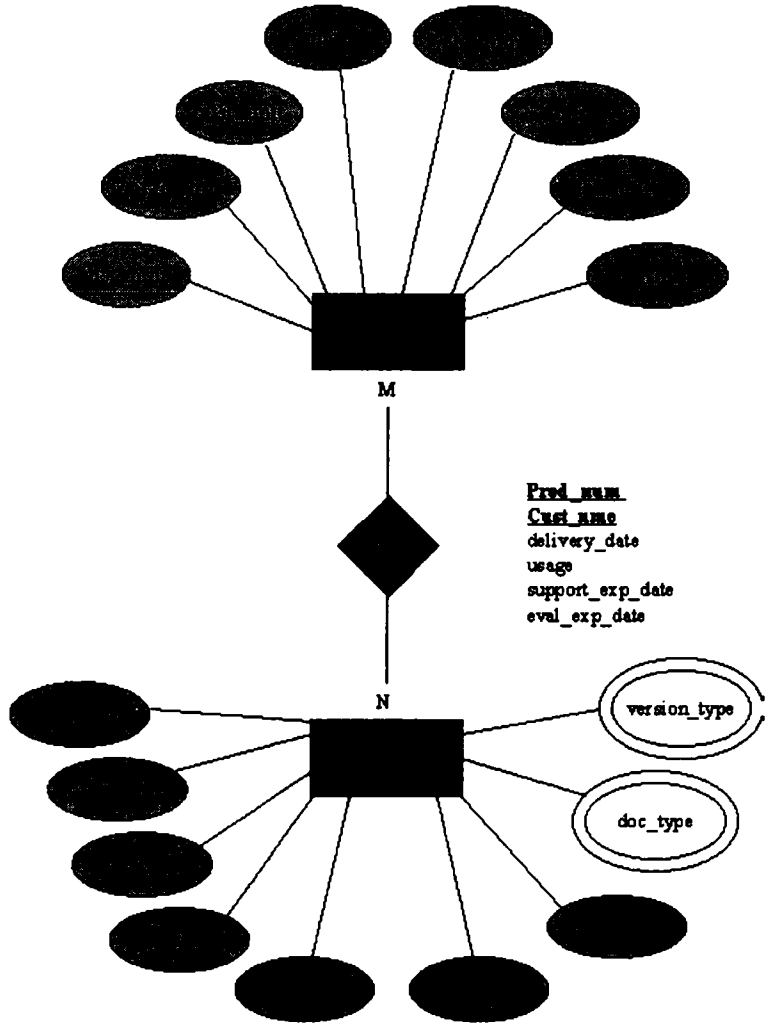


Fig. 6.95

- A relationship is an association between instances of one or more entity types.
- Most relationships are binary.
- An example is the PlacesOrder relationship that exists between occurrences of a CUSTOMER entity and a CUSTOMER_ORDER entity

1. Binary Relationships
   - This type of relationship involves two entities.
   - We say the relationship degree is equal to 2 because two entities participate in the relationship.
   - Examples include the following shown in Fig. 6.96
     Binary EMPLOYEE 1:1 PARKING_PLACE
     Binary PRODUCT_LINE 1:N PRODUCT

Binary ORDER N:N PRODUCT

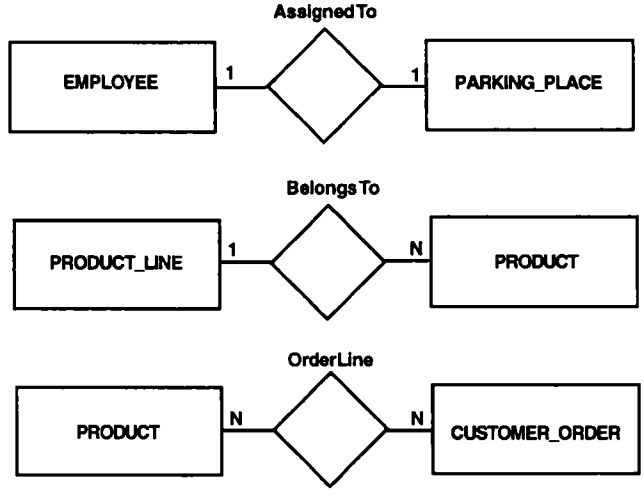


Fig. 6.96

- The maximum cardinality (1:1, 1:M, M:N) expresses the number of occurrences of one entity related to another entity.
- Note how intersection attributes are modelled. Consider the example of modeling the DesiredPrice versus PurchasePrice attributes for the relationship between CUSTOMER_ORDER and PRODUCT entities shown in Fig. 6.97.

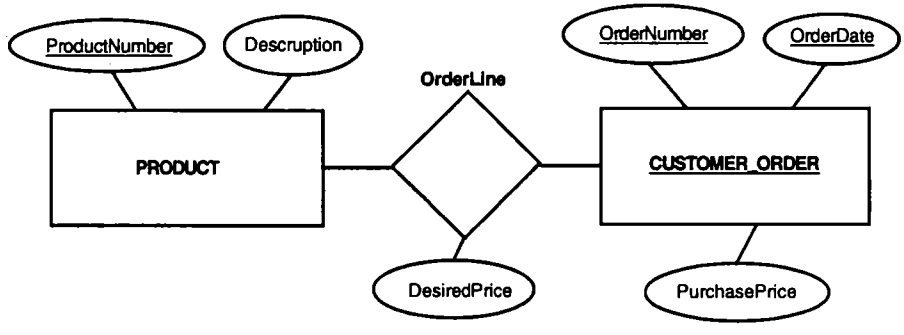


Fig.. 6.97

2. Unary Relationships
   - The unary relationship has degree = 1.
   - This represents an association between occurrences of a single entity.

Examples include the following with the cardinalities noted in Fig. 6.98:

Unary PERSON 1:1 PERSON (Marriage)

Unary EMPLOYEE 1:N EMPLOYEE (Supervise)

Unary ITEM N:N ITEM (BillofMaterials)

- Note how intersection attributes are modelled.

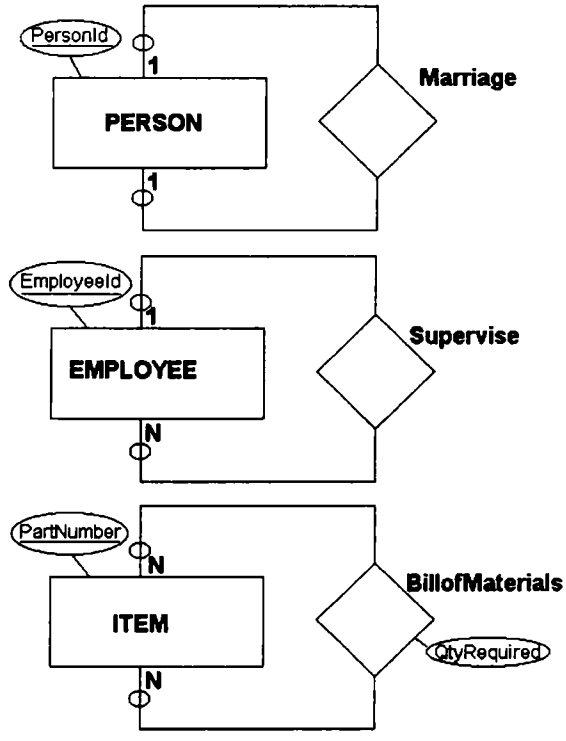


Fig. 6.98

3. Ternary Relationships
   - This models the association between occurrences of three entities at the same time. Degree=6.
   - These are sometimes modeled incorrectly as multiple binary relationships.
   - As an example, consider the N:N:N Ternary Shipment Relationship that exists among the ITEM, VENDOR, and WAREHOUSE entities shown in Fig. 6.99
   - Note how intersection attributes such as QuantityShipped are modeled.

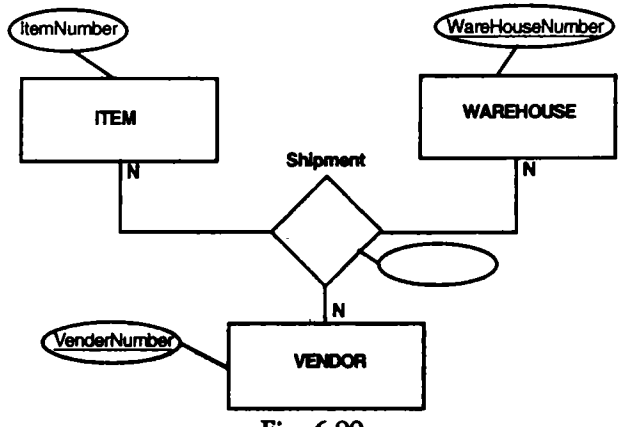


Fig. 6.99

4. Gerund
   * When is a relationship an entity or an entity a relationship. Sometimes you can't determine whether something you need to model is an entity or a relationship, especially when there are data attributes involved. *Some experts in E-R modeling claims there is, in fact, no substantial difference between an entity and a relationship. *Consider the model of SHIPMENT as a Gerund shown in Fig. 7. *A Gerund looks like a relationship, but has its own primary key that is not part of the primary key of one of the entities that participate in the relationship.
   * SHIPMENT has a primary key of ShipmentNumber.
   * Note how the intersection attribute QuantityShipped is modeled.

This model enables the shipment of multiple ITEM occurrences per SHIPMENT.

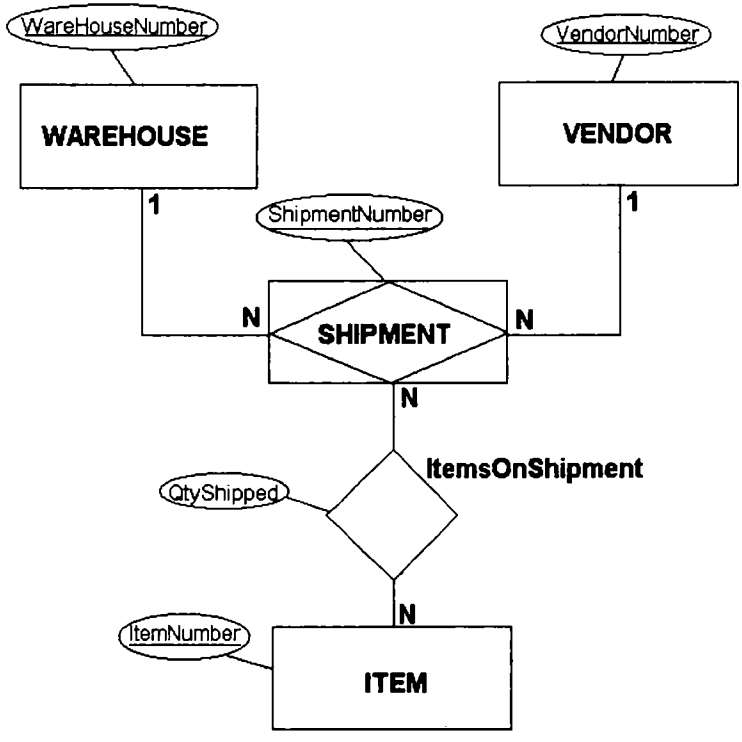


Fig. 6.100

6.24.6 Minimum Cardinality

We previously examined maximum cardinality.

- Cardinality needs to be expressed as a range of values. Maximum cardinality can be one or many.
- Minimum cardinality is also called optional/mandatory cardinality and expresses how many occurrences of one entity must, at a minimum, be associated with occurrences of another entity.

- The binary relationship between MOVIE and MOVIE_COPY shown in Fig. 6.101 has both maximum and minimum cardinality expressed.
- Minimum: For a given Movie, the store may not have any copies (optional).
- Maximum: For a given Movie Copy, there is at most one Movie.
- Minimum: For a given Movie Copy, there must exist at least one Movie (mandatory).

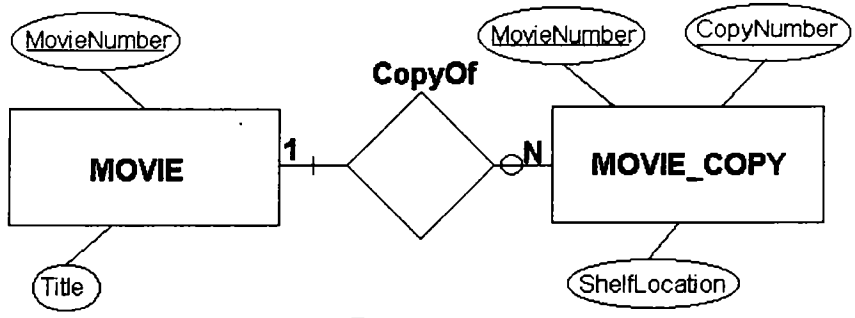


Fig. 6.101

Maximum: For a given Movie, the store may have one or more copies.

- What are the cardinalities between the following relationships?:
  - PATIENT - PATIENT_HISTORY
  - EMPLOYEE - PROJECT
  - PERSON - PERSON (marriage)

6.24.7 Existence Dependency (Weak Entity)

- The MOVIE and MOVIE_COPY example also is a case of existence dependency.
- This means an occurrence of one entity cannot exist unless there is an occurrence of a related entity.
- This usually happens for binary N:N relationships where the cardinality is mandatory-one.
- Weak entities often do not have a natural identifier (candidate key).
- The primary key of the parent entity is used as part of the primary key of the dependent child entity.
- Note the primary key of the MOVIE_COPY entity is a composite key.

This situation is also called an identifying relationship. Benefits include:

1. Data integrity of the existence dependent entity are enforced.
2. Ease of access for related dependent entities via part of the composite key.

6.24.8 Multi-valued Attributes

- Previously we noted the existence of multivalued attributes.
- As an example consider the Skill attribute of EMPLOYEE. Skill may be modeled as a multivalued attribute as shown in Fig. 6.102.
- During database modeling, it is often desirable to decompose this situation into two separate entities since a multivalued attribute often results when we capture only a single data attribute of what would otherwise be modeled as an entity.

- Here SKILL is an abstract entity. We might improve on this model by defining a system generated primary key identifier SkillCode.

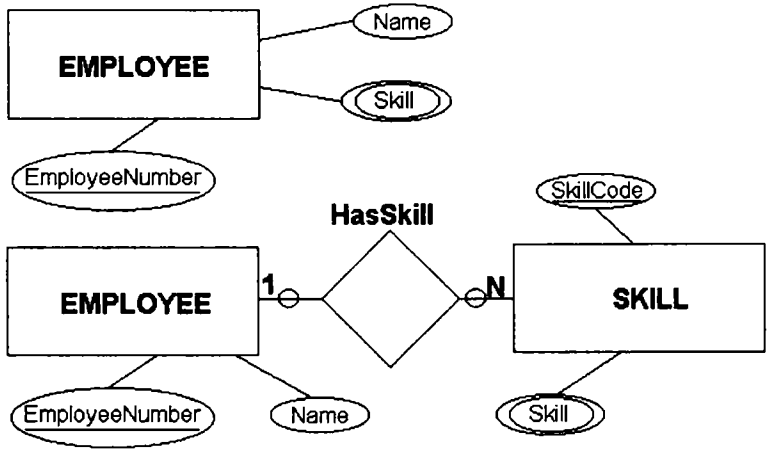


Fig. 6.102

Repeating Groups (Multivalued Attribute Sets).

- Consider a situation shown in Fig. 6.106 where a PATIENT entity has attributes which are related to each other.
- Note the solution by dividing the model into two entities, PATIENT and PATIENT_HISTORY.
- Is this a situation where there is a weak entity? Which one?
- Is there an identifying relationship (look at the cardinality)?

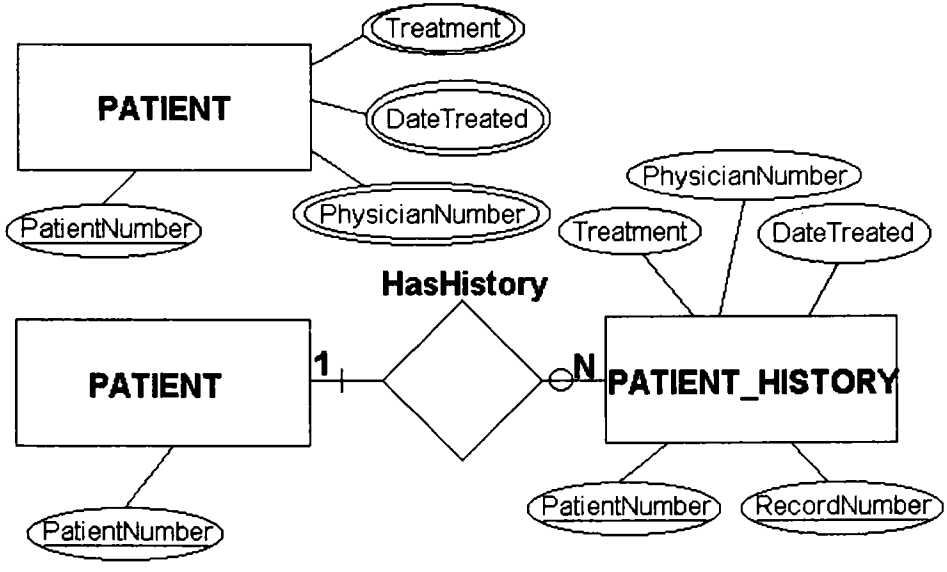


Fig. 6.103

6.24.9 Modeling Time Dependent Data

- A typical approach where data are subject to change over time, but where we must track all values, both previous and current, of an attribute is the use of a time stamp attribute.
- Consider the situation for a PRODUCT entity.
- Note that a solution can be obtained by defining two entities, PRODUCT and PRICE_HISTORY where the diagram is very similar to that shown in Fig. 10. Can you draw this diagram? *What would be the primary key attributes?

6.24.10 Generalisation Hierarchy — Subtypes and Supertypes

- This is the concept of categorizing or generalizing between types and subtypes of entities.
- As an example, a CAR entity can have several subtypes, e.g. CONVERTIBLE, COMPACT, SEDAN, etc.
- Consider The EMPLOYEE supertype entity shown in Fig. 11. This entity can have several different subtype entities (HOURLY and SALARIED), each with distinct properties not shared by other subtypes.
- The Supertype EMPLOYEE stores all properties that subtypes have in common.
- Note the ISA symbol used to diagram the Generalisation Hierarchy indicates that this relationship is a Generalisation Hierarchy. Some versions of the E-R model do not use this same notation.

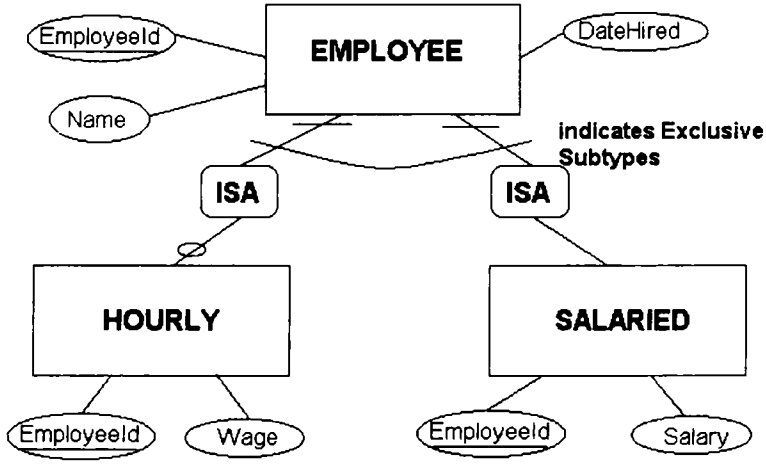


Fig.. 6.104

Inheritance and Primary Keys for Generalisation Hierarchies.

- Subtypes inherit the attributes of the Supertype.
- The primary key of the supertype and subtype are always identical.
- The Maximum:Minimum cardinality between the supertype and subtype are always 1: 0.

6.24.11 Exclusive Relationship for Generalisation Hierarchies

- The subtypes are usually (but may not be) mutually exclusive with no overlap.
- In this case, each instance of a Supertype is associated with exactly one instance of a Subtype, e.g. an employee is either SALARY or HOURLY.
- Note how this is diagrammed.

6.24.12 Non-Exclusive Relationship

- A Supertype may have more than one Subtype entity instance.
- **Example:** A VEHICLE supertype may have overlapping subtypes of AUTOMOBILE, TRUCK, and 4_WHEEL_DRIVE (an AUTOMOBILE and TRUCK may both be 4_WHEEL_DRIVE).
- Practice by drawing the diagram. Create some attributes applicable to each Subtype entity.

Business Rules

- This topic deals with modeling additional information that is generally not amenable to diagramming techniques.
- The CASE tool used may allow storage of business rules expressed as simple logical statements and may enforce such rules during system execution.

Entity Integrity (Uniqueness Integrity).

- Each instance of an entity type must have a unique identifier that is not null.
- This may also be termed Primary Key Integrity or Uniqueness Integrity.

Referential Integrity.

- Integrity rules that exist between occurrences of entities where relationships exist, for example, existence dependency is a type of referential integrity.
- **Example:** An occurrence of a CUSTOMER entity may or may not have an associated occurrence of an CUSTOMER_ORDER entity; but each occurrence of an CUSTOMER_ORDER entity must have one and only one associated occurrence of a CUSTOMER entity.

Domain Integrity.

- A Domain is a defined list or range of valid values for a specific attribute or set of associated attributes.
- This constraint on valid values for attributes is termed Domain Integrity.
- Examples include data type, length, format, range, allowable values, meaning, uniqueness, and null/not-null.

Triggering Operations.

- Triggering Operations is a broad class of other business rules that protect the validity of attribute values.

- These usually apply to the insertion, deletion, and update operations operations that manipulate data.
- For example, it may be desirable to have the database automatically delete (termed a cascade delete) all active CUSTOMER_ORDER occurrences whenever the associated CUSTOMER entity is deleted.
- Triggering Operations are covered in detail later in the course.

6.25 Populating a Conceptual Data Model

- Diagram entities
- Diagram relationships
- Look for hierarchies, aka:
  - Sub-types / super-types
  - Specialisation /generalisation
  - Class hierarchies

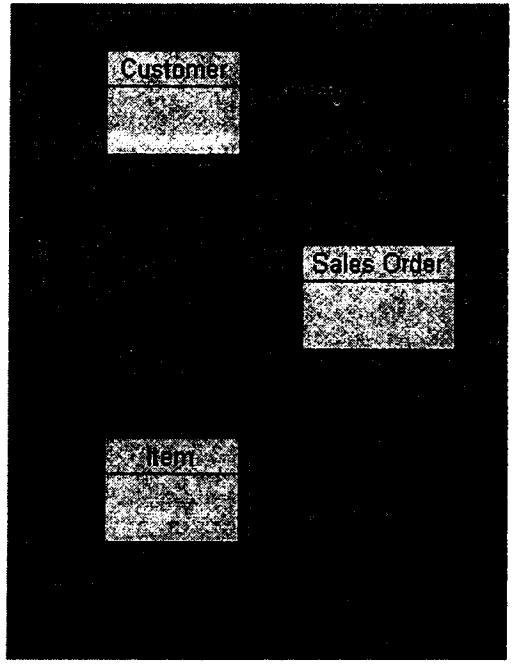


Fig. 6.105

6.25.1 Entity Variations

In modeling systems, certain objects can not be classified as clearly defined, independent entities. To interpret these entities, modelers have developed several diagramming techniques. The following variations are some of the most common.

6.25.2 Entity Hierarchies

Hierarchies represent how entities may be broken down. Two types of entity hierarchies are partitioning and decomposition. Some reasons that entity hierarchies are created include:

RELATIONSHIP	RELATIONSHIP OCCURRENCE
ADVISOR determines CLASS	Dr. John Brown determines INSY312
CLASS located in CLASSROOM	INSY430 located in Room 8
INDIVIDUAL registers for CLASS	Mary Jones registers for BUSN202

Fig. 6.106

- Relationships may not apply to all entity occurrences.
- Attributes may not apply to all entity occurrences.
- Clarification of the diagram.
- Connections between diagrams.

6.25.3 Partitioning

Partitioning involves dividing an entity into subclasses, often referred to as supertypes and subtypes. A relationship nickname of "is a" usually indicates that an entity should be partitioned. Partitioning can be done from the top down (designate the supertype, then break down the subtypes) or from the bottom up (illustrate the subtypes, then designate a supertype). Subtypes may have many levels, and all subtypes must be shown on the diagram (a subtype of "Other" can be used if needed). An entity can be a supertype for more than one subtype structure, or a subtype for more than one supertype.

While all attributes of the supertype apply to the subtype, the reverse is not true. Therefore, attributes should be placed at the highest level they describe. In addition, each hierarchy should have a partitioning attribute that indicates the reason for the partition.

6.25.4 Decomposition

Decomposition involves breaking an entity into the parts from which it is composed. A good indication that an entity needs to be decomposed is the relationship nickname "consists of." In other words, if one entity consists of another entity, it is probably of the same entity class and should be decomposed. As with partitioning, decomposed entities can involve multiple structures and should be exhaustive ("Other" can be used). Attributes for decomposed structures, as for partitioned structures, should be placed at the highest level possible.

6.25.5 Weak Entities

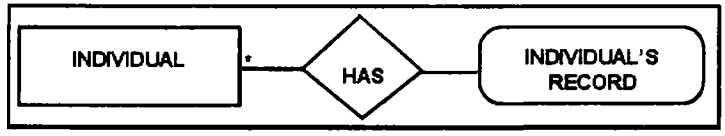


Fig. 6.107

A weak entity is one whose existence in the diagram depends on another entity. For example, in the student registration model, the entity CLASS could not exist without the entity COURSE. A weak entity is represented on the E-R diagram by a round-cornered rectangle.

6.25.6 Characteristic Entity

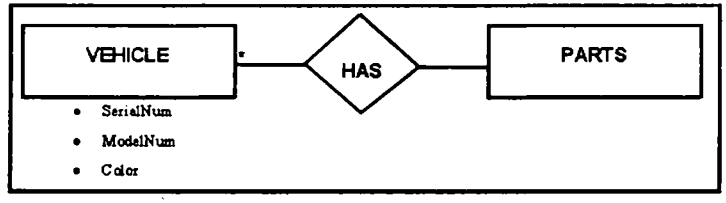


Fig. 6.108

A characteristic entity is an entity attribute that can be considered as an entity itself. Whether the attribute becomes an entity depends on the focus of the user. However, a relationship nickname of "characterizes" usually indicates the need for a characteristic entity. Since it depends on the entity it describes, a characteristic entity is diagrammed as a weak entity.

6.25.7 Associative Entity

Fig. 6.109

An associative entity is an entity that has developed from a relationship for one of the following reasons:

- The relationship is involved in other relationships.
- The relationship needs to be subtyped.
- The user considers the relationship to be an entity.
- The relationship has a many-to-many cardinality.

An associative entity name is formed by changing the relationship verb to a singular noun. Both entities in the original relationship must remain in the diagram for the associative entity to exist. Although it is considered a dependent entity, often the associative entity is diagrammed by placing a rectangle around the existing relationship diamond.

6.25.8 Relationship Variations

Relationship variations can exist as well as entity variations. In modeling, certain relationships do not fall clearly into a binary relationship pattern. Modelers have also developed techniques to define these variations. Some of the most common instances are mentioned in the following sub sections.

6.25.9 Entities with Multiple Relationships

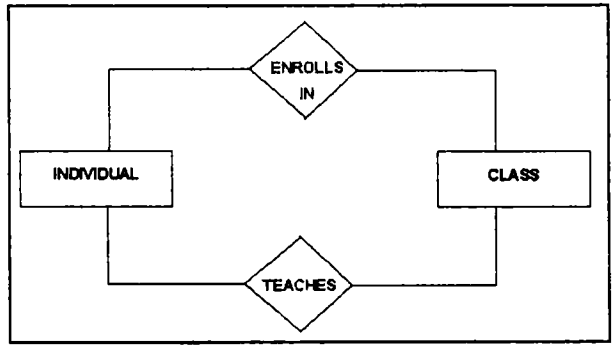


Fig. 6.110

Often, a set of entities may be associated in more than one way. Since it is a precise statement of an association between entities, each relationship must be clearly defined. The result can be more than one relationship between the same entities.

6.25.10 Multi-member Relationship Links

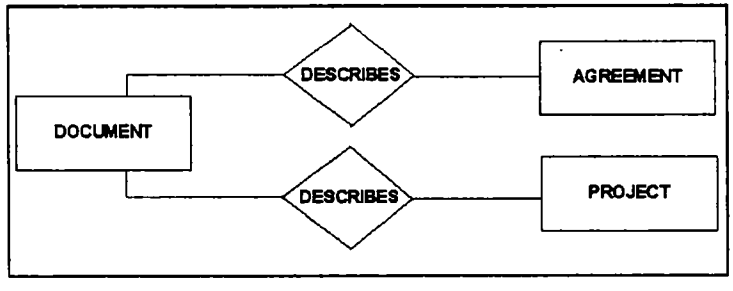


Fig. 6.111

Sometimes, an entity can be involved in two relationships that seem the same. For instance, the relationship name and the attributes may be the same. However, each relationship deviation must be shown on the diagram.

6.25.11 Relationship Roles

A role is a part played by an entity when it participates in a relationship. Roles help establish the true meaning of a relationship and should be noted if clarification is needed. Roles played by individuals and organisations are most often shown on an E-R diagram.

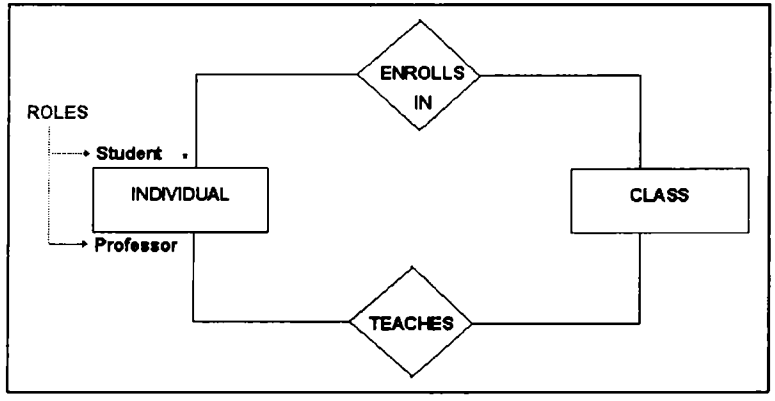


Fig. 6.112

6.25.12 Recursive Relationships

Relationships among entities of the same type or class are called recursive relationships. Other names for these relationships include bill of material structures or nested structures. Relationship roles can be used to clarify recursive relationships.

6.26 Data Modeling Guidelines

The following guidelines were adapted from the article "Data Modeling Rules of Thumb" by George Tillman:

- Use abbreviations in a diagram only when necessary.
- Rework or eliminate confusing elements from the diagram.
- Do not diagram entities that contain no attributes or only one attribute.
- Use entity names that are meaningful to the diagram and the user, but are logical and consistent.
- Do not diagram attribute values.
- Do not diagram data to be calculated, such as totals.
- Do not nest attributes.
- Do not put attributes into groups; for example, "city" and "state" should remain separate items.
- Avoid diagramming reports, processes, screens, or views.
- Diagram only business rules or policies that comply with the structure and meaning of the diagram.

6.27 Normalisation

Normalisation of a relational data model is concerned with finding the simplest structure for a given set of data. It deals with dependency between attributes and also tries to avoid loss of general information when records are inserted or deleted.

For example, consider the following relation (Fig. 6.113):

SOIL1 (Land System, Soil Type, Erodibility)

Land System	Soil Type	Erodibility
Faraway	Loamy Sand	0.10
Limestone	Sandy Loam	0.25
Nundooka	Loamy Sand	0.10
Nuntherungie	Loam	0.35
Old Homestead	Loamy Sand	0.10

Fig. 6.113

Fig. 6.113: Non-normalised relation.

This relation is not normalised since Erodibility is uniquely determined by Soil-Type. Using this non-normalised relation, problems of insertion and deletion anomalies arise:

- The relation between soil type loam and erodibility 0.35 will be lost if the land system record "Nuntherungie" is deleted.
- A new relationship must be inserted between soil type and erodibility if a land system with a new soil type occurs. Now consider the following normalised relations with the same data set (Fig. 6.114):

SOIL2 (Land System, Soil Type)

Land System	Soil Type
Faraway	Loamy Sand
Limestone	Sandy Loam
Nundooka	Loamy Sand
Nuntherungie	Loam
Old Homestead	Loamy Sand

EROSION (Soil Type, Erodibility)

Soil Type	Erodibility
Loamy Sand	0.10
Sandy Loam	0.25
Loam	0.35

Fig. 6.114: Normalised relations.

Here there are two relations instead of one - one to establish soil type for each land system and the other specifies erodibility for each soil type. This example illustrates a third normal form (3NF), which removes dependence between non-prime attributes.

A relational join is the reverse of this normalisation process, where the two relations SOIL2 and EROSION are combined to form SOIL1 (Fig. 6.115).

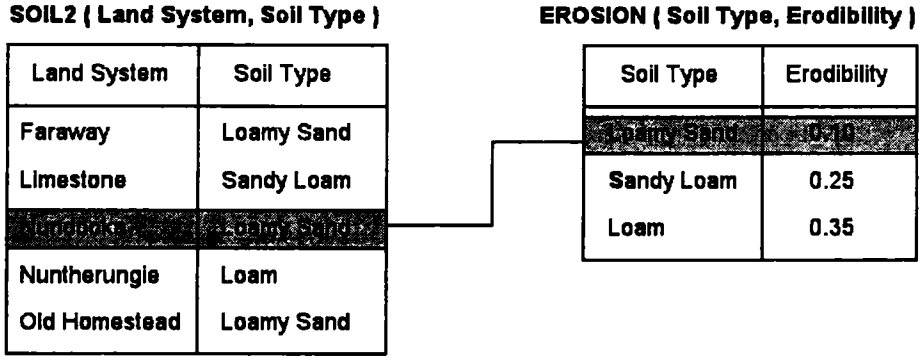


Fig. 6.115: Relational join.

6.28 Representing Data by Coded Values.

In some situations it may be desirable to represent data in storage by coded values i.e. store a code for certain fields e.g. 1=red, 2=blue and so on, but this detracts from data independence and is thus not recommended practice. Actual data values should be stored in a DBMS.

- How Numeric Data can be Stored

Numeric data can be stored in packed decimal or as a character string, binary or demical, fixed or floating point, real or complex, and with varying precision (# digits) and all these still preserve data independence. Any of these aspects may be changed to improve or to conform to a new standard or for many other reasons.

- Storing Character Strings

A character string field may be stored in any of several distinct character codes e.g. ASCII or EBCDIC, while still preserving data independence.

- Stored Record Structure

The application's logical record may contain fields from several distinct stored records i.e. it would be a subset of any given one of those stored records.

6.29 Storage Structures Overview

The internal level of a database system is the level that is concerned with the way the data is actually stored. Physically, databases are stored almost invariably on disks, which are much slower than main memory and the DBMS thus has an objective to minimise the number of disk accesses (disk I/O's) when executing updates/retrievals. A storage structure is defined as an

arrangement of data. Many can be devised, each with different performance characteristics. No single storage structure is optimal for all applications, but a good DBMS should support many different types of storage structures, so that different portions of the database can be stored in different ways, and the storage structure for a given portion can be changed as performance requirements change or become better understood.

A given stored file may be physically implemented in a number of ways for e.g. it may be entirely contained within a single storage volume or spread across several volumes. However, how it is stored should not affect applications in any way, other than in performance thus providing data independence. Thus the stored file structure should be an operating system concern and should be transparent to the users of a DBMS.

6.29.1 Types of Storage Structures.

Many different and varying types of storage structures exist. These vary considerably in their arrangement of the data from one to the other; however there are some fairly standard principles of database access which identify what is involved in the overall process of locating and accessing particular records. We define several commonly occurring storage structures as following.

- Indexing
- Hashing
- Pointer chains
- Compression techniques

The only user who needs to understand the above techniques is the DBA, for other users this is transparent, but if they know the basics, performance will be better.

6.29.2 Sorting : Indexes

Indexes are pointers that are used to improve access and sorting functions of DBMS. They are defined in the database structure along with other information (such as field types and field widths) during the implementation phase. When a field in a table (called the source table) is specified as an index, the DBMS creates another table containing that indexed field plus an assigned cross-reference field. The cross-reference field contains the record address of the indexed record. This cross- reference enables the DBMS to quickly locate and retrieve source data without searching the entire source table.

Without indexes, the DBMS would scan the entire table, reading every row. Because index cross-reference tables are much smaller than the base table, indexed scans are more efficient. However, consider designation of indexes carefully. Although retrieval from indexed tables is faster, a change to an indexed column actually requires the DBMS to make two writes: a change to the base table and a corresponding change to the indexed table. This can slow the performance of the database.

It is not necessary that the indexed field contain unique values. Indexes should not be confused with primary keys, which must contain unique values and will be discussed in a later lesson.

Following is an example of a table (STUDENT) with an indexed field (Student Number) and its cross-reference table.

TABLE 1.5 = STUDENT (indexed on Student Number)

Student Number	Student Course Name	Professor Number	Relative Record Number
100	Sushma	CMP314	Salin Beg 1
105	Jasmine	CMP345	VK Jain 2
100	Hemlata	CMP567	Aditya Sharma 3
115	Karisma	CMP432	Viajay Negi 4

REFERENCE TABLE CREATED FOR STUDENT NUMBER INDEX

Relative Number	Record Student Number
1 100 3 100	2 105 4 115

6.29.3 Advantages of Pointer Chains

The principle advantage of the pointer chain structure is that the insertion/deletion algorithms are somewhat simpler, and possibly more efficient, than the corresponding algorithms for indexing. Also, the structure will probably occupy less storage than the corresponding index structure, because each parent value appears only once instead of multiple times.

6.29.4 Variations On Pointer Chains.

Several variations are available on the pointer chain structure. Firstly, the pointers could be made two way. This would simplify the pointer adjustment necessitated by the operation of deleting a child record. Another extension would be to include a pointer (parent pointer) from each child record direct to the corresponding parent - this would reduce the amount of chain-traversing involved in answering queries. Yet another variation would be not to remove the field contained within the parent file from the child file - this would then make certain retrievals more efficient. However, increased efficiency is nothing to do with the pointer chain structure.

6.29.5 Pointer Chains Storage Structure

The pointer chains storage structure is a type of storage structure which exists at the internal level and which involves two seperate files, a 'parent' file and a 'child' file, with the overall structure being an example of the 'parent/child organisation'. In the example, the student file is the child file, and the address file is the parent file. The parent file has one record for each distinct address, giving the address value as the head of a chain or ring of pointers linking together all records with that address and ending in parent file. This particular storage structure contains several variations, and also has some advantages and disadvantages.

6.29.6 Disadvantages of Pointer Chains

The principal disadvantages of the pointer chain storage structure are firstly that the structure is suitable for one type of query only and may be a hinderance for others i.e. the parent file will probably also require hash addressing in addition to a pointer chain for other queries. Secondly, creating parent/child structures from existing data is non-trivial and in fact such a task would typically involve a database reorganisation. By contrast, it is a comparatively easy matter to create a new index over an existing set of records. Finally, for a given value, the only way to access the 'n'th value is to follow the entire chain. If the records are not clustered appropriately this would therefore involve numberous disk accesses (one for each record) which would hence considerably increase access time.

6.29.7 Example of Pointer Chains

The following is a diagramatic example of pointer chains.

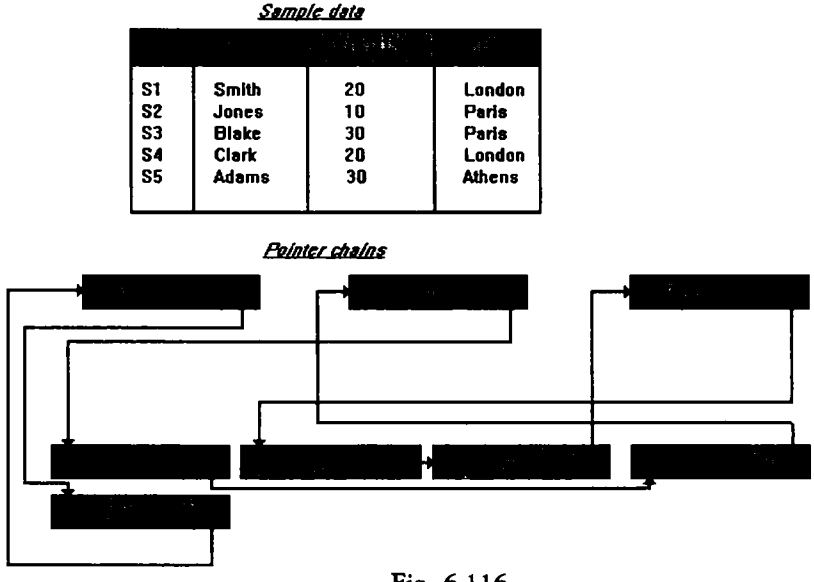


Fig. 6.116

6.30 Storing Data in a File

The data that a program generates usually must be saved and reused at a later time. It would be impractical to keep a single program running for weeks or months at a time if it were used only infrequently. In other circumstances, the computer may not be able to store all the data. For instances, most computers could not store all the data for every one of a bank's depositors. To overcome these difficulties, the data should be stored in a file on either a floppy disk or a hard disk. When there is a very large amount of data to be stored, magnetic tape can be used to store it. For instance, in the first case discussed above, the program would be run and the data stored in a disk file, at which point the execution of the program would terminate.

When the program is run at a later date, the data stored in the file would be read back into the memory, and the program would proceed using the stored data and other new data. In the case of the bank's depositors, the data for all the depositors would be stored on tape or on a large hard disk. Only the data for the depositor whose record was being processed would be read into memory. We shall consider the writing and reading of files stored on a disk. The operating system track of these files. Each file has a name that the operating system uses to identify it (In addition, the operating system must keep track of other things, such as where the file is stored and its size.) The operating system name is the one that appears in the directory listing. Whenever a file is written or read, an area of memory called a buffer is set up to speed the file operations. Suppose one character at a time to and from the file, a sequence of characters is stored in the buffer and transmitted to the file as a group or, conversely, a sequence of characters is transferred to the file as a group or, conversely, a sequence of characters is transferred form the file to the buffer. Each input or output stream must be associated with a buffer. Two procedures are involved in this process. First the buffer must be established, and then it must be associated with the appropriate stream.The file stream.h contains a class type called filebu. When a variable of type filebuf is declared, the appropriate memory allocation is made. The buffer is identified by the declared variable name.

6.31 Necessity of Files

A database is a collection of related data. A telephone book, a checkbook register, a recipe file, and an address book are all collections of related data. A database is created by opening a database application, entering a collection of data, and saving to a file.

The following section describes the structure of a database and defines basic database terms.

- Fields

A field holds one piece of information and is the smallest component part in a database. In an address book, the name of each person is stored next to the label "Name." The name, address, city, and phone number are all separate fields. All names are entered into the "Name" field, all cities are entered into the "City" field, and so on. When this database is displayed in table form, "Name," "City," etc. appear as column headings.

- Field Data Types

Fields can contain many different types of data; for example, "Name" is a text field, while "Zipcode" is a number. When the kind of data stored in a field has clearly defined type, a database can perform sophisticated analyses of the data. For example, mathematical functions can be performed on a "number" field but not on a "text" field. Database applications offer many different data type definitions that vary from package to package and can offer added power to the database. Field Widths

Some database packages require a measurement for each field size. In others the field size can be changed by clicking on its outline and stretching it. Each field size has a size limit. In Access, the size of the field depends upon the type of data (see Field Data Types) stored in the field.

- Field Options

Most databases allow different options for fields in order to make data entry easier or more useful to individual databases. Options such as drop boxes, input masks (such as slashes in a date field or dashes and parentheses in a phone number field), and default values are a few examples of field options that some database applications can utilize.

- Records

A record is a collection of related fields. When you put one person's information—name, address, and phone number—into appropriate fields, you have a record. In table form, a record is one row of information.

- Database or Table

A collection of records is called a database. When displayed on the screen in a table form, it may be called either a table or a database by your database application program. The tabular format is only one way to view your database. In it each record occupies a single horizontal row. In Access, a collection of related records is called a table; several related tables may make up one database. Paradox uses similar terminology. FileMaker, though, does not have tables. Thus, any set of records comprises one database.

- Form or Layout

Data can be entered into a database using a Form. A form separates each record from all other records, showing only one complete record at a time. For their default data entry method, FileMaker Pro and FoxPro use a form, while Paradox and dBase IV use tables rather than forms. FileMaker Pro calls its form a "Layout." Access allows both forms and "datasheet" (table) views for data entry.

- Report

A report is the output of your database. Reports can usually be viewed on the screen or sent to the printer. Reports do not allow the user to change information within the database. However, Report Layouts or Designs are made to be created and customized by the user to show either the full information in the database or only certain fields.

- Finding Information

The amount of information stored in a database can be very large. Often viewing the entire contents of a database in its original format is not useful. The user may want to limit the records to persons living in Chicago, or sort alphabetically by last name. Database programs provide several tools for accomplishing these goals.

- Sorts & Indexes

In most database packages information can be sorted, which usually means arranging it in some order other than the way it was entered. Some database packages ask for designated fields to index in order to "sort" efficiently. For example, maybe names are entered in an address book as people are met, but the address book is more useful if arranged alphabetically. The last name can be "indexed" so the database will automatically sort by last name; whenever a new record is entered it will be sorted by last name with the rest of the records. Sorting arranges the records in a table by a particular field. In this case, the records would be sorted alphabetically according to each person's last name.

- Queries & Find

A query asks for specific information (matching certain criteria) from a database. For example, you could query the addresses in the address book to find only those records whose "city" field matches "Chicago."

In Paradox, dBase IV and FoxPro, the answer to a query appears as a temporary database or table that contains only information specified in the query "criteria." This can be saved as a new table (or database file) which can be analysed like any other database tables.

FileMaker Pro uses its "Find" request mode to accomplish this job. It does not return a temporary database. FileMaker Pro operates a little differently and can be highly effective. Paradox and FoxPro use a "Query by Example" method. Access provides a "Query Builder" to create sophisticated queries.

In a relational database application, queries may be designed to operate on several properly linked databases at once. Database programs can be very flexible (for more information on relational databases, see the section "Relational Database").

- Operators & Symbols

A query or Find searches for values in a field. Expressions that use symbols and operators define the search criteria for the query. For example, to find all addresses in Chicago, the query expression might look like CITY="Chicago". Operators can be familiar mathematical symbols: for example, "=" and "+" are used to build expressions. Database programs may define symbols and operators differently. Each program should have a list of operators and their meanings.

6.32 Working Parts of a DBMS

Each database management application has ways of describing itself. Here are some common parts amongst databases.

- **Images:** Most databases work with Images or an equivalent. On screen a particular view or Image of the data is displayed. If changes are made to this data, they are only a screen Image and no change has occurred to the file on disk. Changes are only written to disk when you explicitly "Save" your work. FileMaker Pro operates differently. It is automatically and continuously saving to disk all changes as they are made, so there is no save command in its File Menu. To protect data, use the "Save a Copy As" command first to preserve the original database, then make changes.
- **Macros & Scripts:** All database applications provide a method for making work easier or quicker. Scripts and Macros are small snippets of code or keystroke usage that can be "played" to quickly perform a set of actions. They save time and effort, especially in doing repetitive tasks. Several common ones usually ship with the application, ready for use, and custom macros and scripts can always be created.
- **Objects:** Objects are the parts of a database. Default objects store, display, and present information. The operation of a table, form, or report is built into the application. They may be changed or used as is. Objects may be very small and simple, or complex. A field that you design and place in a form is an object, and built into this object are the ways

it lets you operate on it. It will allow you to change its size, fill it with data, and allow you to drag it to a new location on the screen. Some applications have special terminology and flexibility for their objects and some do not; most provide a built-in programming language that may be used to create customized objects.

6.33 File Formats

Here is a brief introduction to some of the ways that data files can be formatted.

ASCII vs. binary

ASCII files are ones which can be viewed with simple text editors, such as the "notepad" program. Binary files can only be intelligibly read by a program designed to read the specific format used. Binary files make up for the inconvenience of needing a special program to read them by being much more compact than ASCII files. For example, an image in an ASCII postscript file requires 3-20 times as much disk space for storage as the same image in a binary image file, such as .gif or .jpeg files.

For small amounts of data or information that users may wish to access directly, an ASCII file is an excellent choice. For this class, it is recommended to that you use ASCII files at all times.

6.34 Fixed Format Files

One of the easiest file formats for a programmer to work with is a fixed format file. This is a file which always contains the same type of information at the same place in the file. Thus the code to read such a file is a simple succession of read statements. The disadvantage of a fixed file format is that it is inflexible, meaning that there is no built in way to change the type of information in the file and still be compatable with the original program.

Record Format Another way to structure a file is as a sequence of records each of which is identified by some sort of a keyword. The program reading the file identifies the keyword then calls the routine to read that particular type of information. The program can be designed to skip to the next keyword if it finds an unknown keyword. This lets several programs put different types of information in the same file format and still be compatible with the older programs (which just skip over the unknown data types). A variation on a record based file is a heirachical file format. This is simply a record based file which allows records to be placed inside of other records. This can be a more natural way to store information for some applications.

Maximum: For a given Movie, the store may have one or more copies.

- **Fixed-length and variable-length records**

The question whether to use records of a fixed or variable length is one that usually does not have to be considered in manual systems.

- **Fixed.** Every record in the file will be of the same fixed number of fileds and characters and will never vary in size.
- **Variable.** This means that not all records in the file will be of the same size. This could be for two reasons:

1. Some records could have more fields than others. In an invoicing application, for example (assuming a 6-character field to represent 'total amount for each invoice'), we would add a new field to a customer record for each invoice. So a customer's record would vary in size according to the number of invoices he had been sent.
2. Fields themselves could vary in size. A simple example is 'the name and address' field because it varies widely in size.

Counters

A file may or may not include counters. A counter is simply an integer which is read from the file before the data is read. This number tells the program how much data is going to be read next. This may seem like a trivial point, but it affects how the data can be stored in the program. If the program knows how much data is going to be read, it can allocate the correct amount of memory before reading in the data and use a loop to read the data.

6.35 File Processing Activities

We will need to have access to particular records in the files in order to process them. The major processing activities are given below:

- **Updating** When data on a master record is changed to reflect a current position, e.g. updating a customer ledger record with new orders. Note that the old data on the record is replaced by the new data.
- **Referencing** When access is made to a particular record to ascertain what is contained therein, e.g. reference is made to 'prices' file during an invoicing run. Note that it does not involve any alterations to the record itself.
- **File maintenance** New records must be altered. Customers' addresses also change and new addresses have to be inserted to bring the file up to date. These particular activities come under the heading of 'maintaining' the file. File maintenance can be carried out as a separate run, but the insertions and deletions and deletions of records are sometimes combined with updating.
- **File enquiry or interrogation** This is similar in concept to refernciing. It involves the need to ascertain a piece of information from, say, a master record. For example, a customer may query a Statement sent to thim. A 'file enquiry' will get the data in disputs from the record so that the query may be settled.

6.36 File Organization Methods

6.36.1 Tape Files

- Files organisation on tape

  Organisation of a file on tape is simply a matter of placing the records one after the other ontto the tape. There are two possible arrangements of files:

[a]**Serial** When records are written onto tape without there being any relationship between the records keys. Unsorted transation records would such a file.

[b]**Sequential** When records are written onto tape in sequence according to the record keys.

Examples of sequential files are:

1. Master files.
2. Sorted transaction files.
   - Tape file access

[a] **Serial Files** The only way to access a serial file on tape is SERIALLY. This simply means to say that each record is read from the tape into main storage one after the other in the order they occur on the tape.

[b] **Sequential Files** The method of access used is still SERIAL but of course the fiels is now in sequence, and for this reason the term SEQUENTIAL is often used in describing serial access of a sequential tape file. It is important to note that to process (e.g. update) a sequential master tape file, the transaction file must also be in the sequence of the master file. Access is achieved by first reading the transaction file and then reading the master file until the matching record (using and record keys) is found. Note therefore that if the record required is the twentieth record on the file, in order to get in into storage to process it the computer will first have to read in all nineteen preceding records.

Note. These limited methods of organisation and access have led to tape becoming very much less common than disk as an on-line medium for the storage of master files. Tape continues as a major storage medium for purposes such as off-line data storage and back-up.

6.37 Data Storage Devices

External memory can be of several type but following types are quite popular:

- Magnetic tapes
- Magnetic disk
- Optical Disk
- Audio cassette
- Digital Compact disk

Data is stored on disks and tape in the same way music or television programs are recorded on tape with a difference that in these cases data is stored/recorded digitally. In both cases, a thin plastic film is coated with a thin layer of an oxide of iron-chemically similar to plain old rust. The write/record head of the drive is a sensitive electromagnet, whose magnetic field rapidly changes with the data that is being fed into it. As the magnetic field changes, the head leaves a track of magnetized particles in the film. When the read head passes over these tracks, its field is affected by the fields of the particles, and the data or music can be accurately read or played. The little particles of iron will retain their magnetic tracks for periods of us to several years, but can also be reused, for new data, over and over again. Oddly enough, ordinary audio cassette tapes can be used as external memory for computers. They are not 'random-access' of course and this makes them slow to use. But cassette tapes and recorders are cheap and widespread, and many home computers depend on them.

An index for a file works like a catalogue in a library. Cards in alphabetic order tell us where to find books by a particular author. In real-world databases, indices like this might be too large to be efficient. We will look at more sophisticated indexing techniques.

There are two kinds of indices:

- Ordered indices: indices are based on a sorted ordering of the values.
- Hash indices: indices are based on the values being distributed uniformly across a range of buckets. The buckets to which a value is assigned is determined by a function, called a hash function.

We will consider several indexing techniques. No one technique is the best. Each technique is best suited for a particular database application.

Methods will be evaluated on:

(a) **Access Types**— Types of access that are supported efficiently, e.g., value-based search or range search.

(b) **Access Time**— Time to find a particular data item or set of items.

(c) **Insertion Time**— Time taken to insert a new data item (includes time to find the right place to insert).

(d) **Deletion Time**— Time to delete an item (includes time taken to find item, as well as to update the index structure).

(e) **Space Overhead**— Additional space occupied by an index structure.

We may have more than one index or hash function for a file. (The library may have card catalogues by author, subject or title.) The attribute or set of attributes used to look up records in a file is called the search key (not to be confused with primary key, etc.).

6.38 File Organisation

A file is organized logically as a sequence of records. Records are mapped onto disk blocks. Files are provided as a basic construct in operating systems, so we assume the existence of an underlying file system. Blocks are of a fixed size determined by the operating system. Record sizes vary. In relational database, tuples of distinct relations may be of different sizes. One approach to mapping database to files is to store records of one length in a given file. An alternative is to structure files to accommodate variable- length records. (Fixed-length is easier to implement.)

6.38.1 Overview of Physical Storage Media

1. Several types of data storage exist in most computer systems.

They vary in speed of access, cost per unit of data, and reliability.

- Cache: most costly and fastest form of storage. Usually very small, and managed by the operating system.
- Main Memory (MM): the storage area for data available to be operated on.
  (a)General-purpose machine instructions operate on main memory.
  (b)Contents of main memory are usually lost in a power failure or " crash".
  (c) Usually too small (even with megabytes) and too expensive to store the entire database.
    - Flash memory: EEPROM (electrically erasable programmable read-only memory).

(a) Data in flash memory survive from power failure.

(b) Reading data from flash memory takes about 10 nano-secs (roughly as fast as from main memory), and writing data into flash memory is more complicated: write-once takes about 4-10 microsecs.

(c) To overwrite what has been written, one has to first erase the entire bank of the memory. It may support only a limited number of erase cycles (to).

(d) It has found its popularity as a replacement for disks for storing small volumes of data (5-10 megabytes).

- Magnetic-disk storage: primary medium for long-term storage.

(a) Typically the entire database is stored on disk.

(b) Data must be moved from disk to main memory in order for the data to be operated on.

(c) After operations are performed, data must be copied back to disk if any changes were made.

(d) Disk storage is called direct access storage as it is possible to read data on the disk in any order (unlike sequential access).

(e) Disk storage usually survives power failures and system crashes.

- Optical storage: CD-ROM (compact-disk read-only memory), WORM (write-once read-many) disk (for archival storage of data), and Juke box (containing a few drives and numerous disks loaded on demand).
- Tape Storage: used primarily for backup and archival data.

Cheaper, but much slower access, since tape must be read sequentially from the beginning. Used as protection from disk failures!

The higher levels are expensive (cost per bit), fast (access time), but the capacity is smaller.

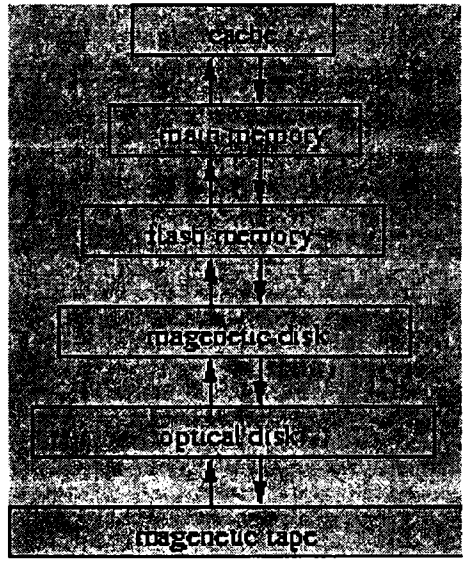


Fig. 6.117: Storage-device hierarchy

Another classification: Primary, secondary, and tertiary storage.

- **Primary Storage:** The fastest storage media, such as cash and main memory.
- **Secondary (or on-line) Storage:** The next level of the hierarchy, e.g., magnetic disks.
- **Tertiary (or off-line) Storage:** Magnetic tapes and optical disk juke boxes.

Volatility of storage. Volatile storage loses its contents when the power is removed. Without power backup, data in the volatile storage (the part of the hierarchy from main memory up) must be written to nonvolatile storage for safekeeping.

6.38.2 Grid File

1. A grid structure for queries on two search keys is a 2-dimensional grid, or array, indexed by values for the search keys. Fig. 1.118 shows part of a grid structure for the deposit file.

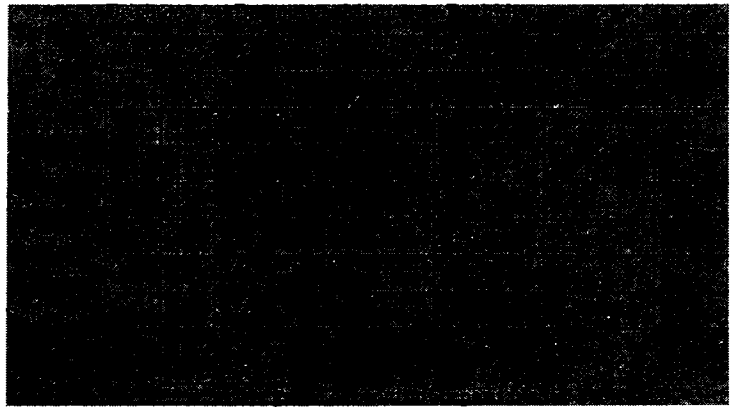


Fig. 6.118: Grid structure for deposit file

A particular entry in the array contains pointers to all records with the specified search key values.

- No special computations need to be done
- Only the right records are accessed
- Can also be used for single search key queries (one column or row)
- Easy to extend to queries on n search keys - construct an n-dimensional array.
- Significant improvement in processing time for multiple-key queries.
- Imposes space overhead.
- Performance overhead on insertion and deletion.

6.38.3 Clustering File Organisation

1. One relation per file, with fixed-length record, is good for small databases, which also reduces the code size.
2. **Many large-scale** DB systems do not rely directly on the underlying operating system for file management. One large OS file is allocated to DB system and all relations are stored in one file.

3. To efficiently execute queries involving , one may store the depositor tuple for each cname near the customer tuple for the corresponding cname, as shown in Fig. 10.19. 6.This structure mixes together tuples from two relations, but allows for efficient processing of the join.
4. If the customer has many accounts which cannot fit in one block, the remaining records appear on nearby blocks. This file structure, called clustering, allows us to read many of the required records using one block read.
5. Our use of clustering enhances the processing of a particular join but may result in slow processing of other types of queries, such as selection on customer.

For example, the query

```
aaaaaaaaaaa_select *
```

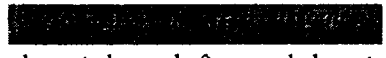
from customernow requires more block accesses as our customer relation is now interspersed with the deposit relation.

Thus it is a trade-off, depending on the types of query that the database designer believes to be most frequent. Careful use of clustering may produce significant performance gain.

6.38.4 Natural Join Operation

1. Another way to reduce the size of temporary results is to choose an optimal ordering of the join operations.
2. Natural join is associative:

3. Although these expressions are equivalent, the costs of computing them may differ. *Look again at our expression



* We see that we can compute deposit branch first and then join with the first part. *However, deposit branch is likely to be a large relation as it contains one tuple for every account. *The other part,

is probably a small relation (comparatively).

So, if we compute

first, we get a reasonably small relation. It has one tuple for each account held by a resident of Port Chester. This temporary relation is much smaller than deposit branch.

4. Natural join is commutative:

Thus we could rewrite our relational algebra expression as:

- But there are no common attributes between customer and branch, so this is a Cartesian product.
- Lots of tuples!
- If a user entered this expression, we would want to use the associativity and commutativity of natural join to transform this into the more efficient expression we have derived earlier (join with deposit first, then with branch).

6.39 Clustered Indexes

There can only be one clustered index per table. There is a simple physical reason for this. While the upper parts (commonly referred to in SQL Server documentation as nonleaf levels) of the clustered index binary tree structure are organized just like the nonclustered index binary tree structures, the bottom level of the clustered index binary tree are the actual 2-KB data pages associated with the table. There are two performance implications here:

Retrieval of SQL data based on key search with a clustered index requires no pointer jump (with a likely nonsequential change of location on the hard disk) to get to the associated data page because the leaf level of the clustered index is already the associated data page. The leaf level of the clustered index is sorted by the columns that comprise the clustered index. Because the leaf level of the clustered index contains the actual 2-KB data pages of the table, this means the row data of the entire table is physically arranged on the disk drive in the order determined by the clustered index. This provides a potential I/O performance advantage when fetching a significant number of rows from this table (at least more than 16 KB) based on the value of the clustered index, because sequential disk I/O is being used (unless page splitting is occuring on this table, which will be discussed later in the section The importance of FillFactor and PAD_INDEX). That is why it is necessary to pick the clustered index of a table based on a column that will be used to perform range scans to retrieve a large number of rows.

6.40 Non-clustered Indexes

Non-clustered indexes are most useful for fetching few rows with good selectivity from large SQL Server tables based on a key value. As mentioned before, nonclustered indexes are binary trees formed out of 2-KB index pages. The bottom or leaf level of the binary tree of index pages contains all the data from the columns that comprised that index from the leaf level 2-KB page to the associated 2-KB data page. When a non-clustered index is used to retrieve information from a table based on a match with the key value, the index binary tree is traversed until the a key match is found at the leaf level of the index. Then a pointer jump is made if columns from the table are needed that did not form part of the index. This pointer jump will likely require a nonsequential I/O operation on the disk. It might even require the data to be read from another disk if the table and its accompanying index binary tree(s) are large. If multiple pointers lead to the same 2-KB data page, less of an I/O performance penalty will be paid because it is only necessary to read the 2- KB page into the data cache once. For each row returned for an SQL query that involves searching with a nonclustered index, one pointer jump is required. These pointer jumps are the reason that nonclustered indexes are better suited for

SQL queries that return only one or a few rows from the table. Queries that require many rows to be returned are better served with a clustered index.

6.41 Covering Indexes

A special case of non-clustered index is the covering index. The definition of a covering index is a nonclustered index built upon all the columns required to satisfy an SQL query, both in the selection criteria and the WHERE predicate. Covering indexes can save a huge amount of I/O and bring a lot of performance to a query. But it is necessary to balance the costs of creating a new index (and another binary tree index structure that needs to be updated every time a row is written or updated) against the I/O performance gain the covering index will bring. If a covering index will greatly benefit a query or set of queries that will be run often on SQL Server, the creation of that covering index may be worth the cost.

Example of a Covering Index

```
Select col1,col3 from table1 where col2 = 'value'
```

The index created, called "indexname1" in this example, is a covering index because it includes all columns from the SELECT statement and the WHERE predicate. This means that during the execution of this query, SQL Server does not need to access the data pages associated with table1. SQL Server can obtain all of the information required to satisfy the query by using the index called indexname1. Once SQL Server has traversed the binary tree associated with indexname1 and found the range of index keys where col2 is equal to 'value', SQL Server knows that it can fetch all of required data (col1,col2,col3) from the leaf level (bottom level) of the covering index. This provides I/O performance in two ways:

SQL Server obtains all required data from an index page, not a data page, so the data is more compressed and SQL Server saves disk I/O operations. The covering index has organized all of the required data by col2 physically on the disk. This allows the hard drives to return all of the index rows associated with the where predicate (col2 = 'value') in sequential order. This gives us better I/O performance. In essence, a covering index, from a disk I/O standpoint, becomes a clustered index for this query and any other query that can be satisfied by the columns in the covering index. Please note also that the index was created with col2 first in the CREATE INDEX statement. This is important to remember.

The SQL Server 6.5 query optimizer only makes use of the first column of a compound index such as this. So, if one of the other columns had been specified as the first column of the compound index, SQL Server would have ignored the index within the context of the example query above.

In general, if the covering index is small in terms of the number of bytes from all the columns in the index compared to the number of bytes in a single row of that table, it may make sense to use a covering index.

When a SQL Server environment involves many SQL queries on a given table, these queries ask for a large proportion of the columns of the table, and because it is not possible to reduce the set of columns requested, it may be very difficult to rely on covering indexes for help. Having many indexes (of any kind) on a table will slow down SQL Server at write time because insert/update/delete activity requires updating associated information in the index binary tree structures.

6.42 Index Selection

How indexes are chosen significantly affects the amount of disk I/O generated and, subsequently, performance. The previous sections described why nonclustered indexes are good for retrieval of a small number of rows and clustered indexes are good for range scans. Here is some additional information about index selection.

For indexes that will contain more than a single column, be sure to put the most selective column first. This is very important in helping the SQL Server query optimizer use the index effectively. Also try to keep indexes as compact (fewest number of columns and bytes) as possible.

In the case of nonclustered indexes, selectivity is important, because if a nonclustered index is created on a large table with only a few unique values, using that nonclustered index will not save I/O during data retrieval. In fact, using the index would likely cause much more I/O than a sequential table scan of the table. Some examples of good candidates for a nonclustered index are invoice numbers, unique customer numbers, social security numbers, and telephone numbers.

Clustered indexes are much better than nonclustered indexes for queries matching columns that don't have a lot of unique values because clustered indexes physically order table data by clustered index order, allowing for sequential 16-KB I/O on the key values. Remember that it is important to get rid of page splitting on a clustered index to ensure sequential I/O. Some examples of possible candidates for a clustered index include states, company branches, date of sale, ZIP codes, and customer district. It would tend to be a waste to define a clustered index on the columns that have very unique values unless typical queries on the system fetch large sequential ranges of the unique values. The key question to ask when trying to pick the best column from each table to create the clustered index on is, "Will there be a lot of queries that need to fetch a large number of rows based on the order of this column?" The answer is very specific to each SQL Server environment. One company may do a lot of queries based on ranges of dates whereas another company may do a lot of queries based on ranges of bank branches.

Samples of WHERE predicates that benefit from clustered indexes:

```
.WHERE <column_name> > some_value
.WHERE <column_name> BETWEEN some_value AND some_value
.WHERE <column_name> < some_value
```

6.43 Database Design

Database design is a process consisting of many steps. Although different designers may use different methods, generally the process follows a basic pattern of data modeling, a description of the data to the database, and physical implementation of the database. This basic process can be broken down into five phases:

- Planning
- Analysis

- Design
- Implementation
- Maintenance

The Table 6.1 lists sthe five phases of database design, along with the purpose and some tools and/or techniques of each.

Table 6.1 Five phases of database design

Phase	Purpose	Tools and Techniques
Planning	Define what to accomplish	• Focus lists • Issue lists • Goals
Analysis	Define objects of importance	• Business models • Diagrams • Logical data models
Design	Define how the database is built	• Tables or files • Structure • Constraints • Physical models
Phase	**Purpose**	**Tools and Techniques**
Implementation	Define the database to the DBMS	• DBMS program • Application design • Views, queries, reports • SQL
Maintenance	Manage and control the database	• Data dictionary • Security functions • Structure management • Documentation

It is important to note that these phases often overlap and that some techniques and tools may be used in more than one stage, especially between analysis and design. For example, although the data dictionary is usually introduced in the analysis stage, it is often carried over into the design phase and later used by the database administrator in the maintenance stage.

Keep in mind that many database terms are used interchangeably and that tools and techniques are often perceptual. Also, remember that database design is nondeterministic; in other words, there is no "right" design. The true goal of database design is to create a well-structured database that represents the user's perspective of the business and provides the user with a productive business tool.

When thinking about good database design, it is important that you keep data retrieval, storage and modification efficiency in mind. It will pay off one thousand fold if you take a week or two to simply play with different arrangements of data. You will find that certain table structures will provide easier and more intuitive access than others.

Tables should describe only one subject, have distinct fields, contain no redundant data, and have a field with unique values so that the table can be related to others.

You should also keep in mind future expansion of the database and make sure that your design is easily extensible. Typically, you will go through a requirements phase in which you should simply sit with the problem, interview users, and achieve an intuition about the data and the project.

Next, you should spend time modeling the data, preferably using some standard methodology like ER Diagramming. However, even if you do not model in any traditional way, you can still play with different ideas and think about the pros and cons. Finally, you should try out your ideas and hone them through limited trials.

Hopefully, you will also choose a database with full- functionality such as security and concurrency control (making sure that two users cannot simultaneously change a record). There are many excellent choices available in the market today from freeware to commercial products.

Of course, as we said above, you will probably be coming onto a project with an already existing database. This is the case for most web developers. In this case, you should work closely with the database administrator to define the database.

6.43.1 Selecting Your Data

To determine what data to include in your database, begin by identifying and analyzing your existing data sources, processes, and outputs. This analysis helps identify the depth and range of the necessary data. Ask who will be using the system, who — if anyone — currently provides the type of information it will provide, how these people are managed, and what their division does for the company. Interviewing these people and documenting their interaction with the data is vital for determining the system's data requirements.

The next way to research what data you'll need is to look at the systems being replaced or consolidated. These systems may have provided the same or a similar type of functionality that your system is to provide. What was the critical data factor or data function difficulty that led to their demise? Analyzing and documenting these systems' functionality is critical for your database system's success. If you fail to replace some data or functionality provided by the old system, you may end up having to undertake costly reengineering later, even if your new system has additional features or other important capabilities.

Another way to come up with breakthrough processing or data ideas is to think about what is "impossible" or what the perfect database system would do. You are designing a new system, so you might as well try to make the best system possible. Listen to input from different levels of management, dedicated users of the existing system, and especially people who do not like the system or its concept. People who don't like the system can help you identify potential flaws and weaknesses.

You should completely document and categorize all this analysis. The documentation, or metadata, should include information on the data source, where it's used, activity against it, its primary and secondary keys, and any relationships with other data. The analysis efforts should focus on identifying data groupings, duplicate data, important keys, relationships, and timeliness of the data. As business intelligence requirements grow and OLAP database tools become easier and less intrusive, timeliness and "transaction dimension" information (such as who, what, when, where, and under what conditions or promotion) becomes much more valuable. Analysis should also determine the aggregate knowledge that the data represents and ensure that it corresponds to the new system's overall processing objectives. This knowledge should match the system's mission statement and contain all the data knowledge necessary to provide the functionality and features desired.

- What Data to Track

For example, consider "Name." Depending on the uses of the database, it might be useful to split "Name" into separate first and last name fields, and salutations (Mr., Ms., etc.) and suffixes (Jr., III, etc.). Where the Data Goes

Organize related sets of data into tables that are compact. Data sets that are unique belong in a table separate from data sets that repeat. See the Client (unique) and Invoice (repeats) databases below.

- Data Uses

Consider the possible ways the data might be used. There are reports, but what about writing letters? Dialing phones? Making telephone directories and mailing lists?

- Relating Tables

Fields that establish a link between the databases (tables) are essential. Consider using unique and arbitrary numbers (codes) to identify people, parts, invoices, etc. Using identifying numbers in all tables makes it possible to relate tables, to join them to create reports that contain information from two or three or more tables. This is what a relational database application has the ability to do.

- Format

This word is used to refer to an empty form on which information can be recorded in pre-determined fields. The form can be on paper, or a layout on a word processor, or it can be an entry layout within a database program. A completed format is called a "record". The term "standard format" is used to refer to a set of fields with prescribed scope notes (see definition below) or rules of entry.

- DATA ENTRY

General instructions

To record information in a Standard Format involves a three-stage process:

- Collecting the documentation
- Analysing the information
- Recording the data

The documentation can be simply the statement of an eye witness, or it can consist of a whole collection of evidence about a particular event. The first step is to become familiar with the overall contents of the documentation.

Analysing the information and recording the data would normally be done as one process. Work from the list of field names and the Scope Notes for the selected Format. Find the specific information – e.g. the place where the event took place – from the documentation and record it in the appropriate field in the form prescribed by the Scope Notes. Work systematically through the fields, using the Scope Notes and the Supporting Documents as reference tools.

6.43.2 Normalization

Once you have taken all the metadata into consideration, you're ready to begin the database design normalization process. This process breaks the data into groups, identifying keys, repeating groups, and distinct elements. Normalization puts the keys and data together properly so that it can be retrieved, updated, inserted, and deleted without jeopardizing the information's integrity. Validating the data integrity of the logical modeling process can be quite time consuming, but it is critical to ensuring that interaction between data groups is correct. Following your shop's standard database normalization process is the best way to get everyone to agree and endorse your design. And it is important to make your logical design the simplest possible representation of the data so everyone understands what knowledge and information the database represents.

Through your database normalization design process, eliminate repeating groups to expose relationships within the data. Document these relationships and dependencies and develop referential integrity constraints to govern them in your physical database design. Determination of the proper physical design for relationship constraint rules, such as Delete Restrict, Cascade, or Set Null, should be handled carefully to retain the integrity of the knowledge within the database. You can ask several questions to determine the proper referential integrity constraint: Is the data valid without other data? Is other data dependent on a child or parent relationship? Is only part of that other data affected by the data relationship? For example, you could represent a customer purchasing a piece of merchandise in a data relationship. The purchase could not exist on its own or without the customer. The merchandise transaction is also dependent on the purchase time and, potentially, the price. You could implement these relationships in several different ways: by not letting customers be deleted from the database if they have transactions (Delete Restrict); by cascading or deleting all customer transaction data when the customer information is deleted (Cascade); or by only setting the transaction customer information to null when the customer data is deleted (Set Null).

6.43.3 Identifying Domains

Another factor critical to the success of any database design effort is identifying the data's domain, range, and indicator or code values and ensuring that these elements are compatible across data groups and system interfaces. You will need to conduct domain recognition to determine how the data is represented — for example, whether as a number, character field,

audio, image, or video. To determine the domain of a data element, ask people who work with the data for typical and extreme examples of its use. Also, to ensure that you are identifying the proper domain definitions, make sure that any composite data is broken down to its smallest elements. Don't make the common mistake of misrepresenting the data to conform to an older system's incorrect definition. Misrepresenting the data can cause tremendous data cleansing difficulties when adding data from outside sources for data warehousing or marketing systems.

Logical database analysis should also identify data range or scope to facilitate physical database definitions. Certain physical data element implementations, such as SMALLINT, can only represent up to a certain range of values. Understanding an element's range is also critical when using the element to define physical database partitioning or spread very large databases out for DASD I/O and parallelism considerations. Range information is especially important for indicator or code data elements: You can avoid programming and data population nightmares by staying with common consistent definitions. Standardization and validation software tools are a tremendous help on this front. Using common industry codes or abbreviations, such as stock keeping unit numbers, can also be a great help in settling tedious data range disputes.

6.43.4 Naming Standards

Another important element of successful database design is the use of proper naming standards or conventions. Every IS organization has standards, but often they aren't used because of internal politics, disputes, outdated names, or lengthy integration processes. Have your design team address standards issues at the very beginning of the database design effort so you have time to battle through the various opinions to a resolution. Working with a repository, standard abbreviation list, or existing interface or system can help guide your team in developing naming standards. Because these naming standards will exist for the lifetime of your system, it's important to ensure that they make sense and are easy to understand.

6.43.5 Denormalization and the Rules of Reconstruction

Software and Database engineering are complex activities that require planning and control to be successful. By the time the DBA is called up to tune the indices of a database it is probably already too late. Efficiency should be designed into the data structure before the data is actually put on disk. Since the invention of CASE tools there is usually a missing step in the database design. The logical database design is set up in the modeling tool and then the DDL is generated. The table design of the physical database is the entity design of the logical database. Then when tuning is required, data is moved around on disk, indices are applied, freespace is modified, and more CPU memory is assigned.

The DBMS level tuning steps are valid and will continue to be used. But, there has been a missing step in the database design process.

6.43.6 Physical Design of Databases

The word 'denormalization' is used to describe changes to the table design that cause the physical tables to differ from the normalized entity relationship diagram. 'Denormalization' does not mean that anything goes. Denormalization does not mean chaos. The development of

properly denormalized data structures follows software engineering principles that insure that information will not be lost. If the table is read-only (periodically refreshed from the system-of-record) then the rules are looser. Star schemas and hyper-cubes are read-only denormalizations. If the data is to be distributed and/or segmented and added-to, changed, or deleted from then the reconstruction described below must be followed. Fundamentally, a single principal must be followed. If the individual table is updated in more than one system, it should be possible to reconstruct the original table as if the data was never reformatted or taken apart.

6.43.7 Denormalization

There are many techniques for denormalizing a relational database design. These include -

1. **Duplicated data** - This is the technique of making copies of data whole or in part and storing and utilizing both the original and the copy(s). This technique is great unless you want to update it. This is the area of distributed updates and synchronization. Whole texts have been written on this subject. The general idea is that extra-DBMS processes must insure integrity and accuracy. Stored joins are an example of duplicated data.
2 **Derived data** - The issues with storing derived data are accuracy and timeliness. When the base data changes the derivation(s) must change accordingly. When the semantics of the derived columns is 'current balance' you have one sort of accuracy problem. When the semantics of the derived column is average sales by product, salesman, and division, and month; and the salesman are constantly being reassigned. You have another accuracy problem. Also many designers store the derivation in tables containing inappropriate keys. When derivations are not stored with their logical (functionally dependent) keys subsequent (tertiary) derivations are inaccurate. Also many derivations are non-additive (percents, highest, lowest, etc). This subject deserves many chapters in data warehousing texts. See references to summary data and slowly changing dimensions.
3. **Surrogate keys** - There is a problem with very long and compound keys in that they are hard to use when writing queries and they generate inefficient indices. If the table has a very long key and also has many rows this can generate a "show stopper" situation. If the table has a maximum of 100,000,000 rows and a fifty byte real compound key, assigning a 10 digit surrogate key (and indexing on it) will increase performance dramatically. Imagine the situation where the fifty byte key is used in an equi-join! The real key(s) should not be deleted after the surrogate key is added. This would make reversing out the surrogate key impossible. And would offend the Rule of Reconstruction (see below). Usually the long real key is made up of many sub- keys that are useful in their own right.
4. **Over Normalization (Vertical partitioning/segmentation)** - This is the technique of splitting the original logical table into two or more physical tables. By assigning some of the columns to one physical table and some to another. Both tables end up with the same number of rows and have the same keys (see "Rule of Reconstruction", below). Grossly this will increase performance since the individual tables are now smaller. In most DBMSs the negative affect of long column length is non-liner. The query time against a 1000 byte row length table can be more than twice the query time against a 500 byte row length table. So arbitrary

vertical partitioning will cause much better performance against each of the separate partitions. If you are constantly joining the partitions, over normalization is self-defeating. Therefore, the trick is to cluster the columns together that are used together.

5. **Horizontal segmentation** - This is the technique of storing some of the rows in one table and some in another. Many modern DBMSs can do this automatically. When the criteria for segmentation is non-simple, segmentation must still be done programmatically. Of course, update anomalies occur when rows occur in more that one segment.
6. **Stored Joins** - This is the technique of joining two or more tables together and storing the answer set as an additional table. This is one of the most common denormalizations. If the stored join table is never updated, there is no problem with this. Since this always generates duplicate data, updates are a problem. Look out for query anomalies when a measurement column is on the many side of the relation being joined.
7. **Recurring data groups (vector data)** - When there is a fixed small number of subordinate tables associated with a table collapsing the subordinate table into the parent table will increase performance. Care must be taken that the logical key of the subordinate table is not deleted or obscured. Otherwise the join is not reversible and the "Rule of Reconstruction" is offended.

6.43.8 Rule of Reconstruction

When the Rule of Reconstruction is ignored and the data updated, the data is corrupted.

Codd's Rule of Reconstruction is a corollary to 'Lossless-Join Decomposition'. Lossless Decomposition is a method for creating well-formed normalizations from non-normalized database designs. The Rule of Reconstruction is basically the same idea in reverse. Well-formed non-normalized structures are created from normalized tables.

Read-only database (data warehouses, data marts, etc.) gain flexibility and quality if they also approximate this approach. The designer should think in terms of a series of transformations via SQL. If the physical database design is not based on a well- formed (Third Normal Form) logical database design you cannot know if the Rule of Reconstruction is being followed or not. There seems to be some kind of data entropic law here. If data is not carefully designed and managed through time, it slides into chaos. Constructions based upon algebra and set theory are fragile. In my experience, if you don't know and can't prove the design is correct, it is very likely incorrect.

In reality, it is more subtle than this. For example, if the business changes and the logical data model is not updated and the corresponding physical database design is not updated, you will more than likely get the same effect as a poorly formed denormalization. Add, Changes, and Deletes will corrupt the data relative to the present real business. Future queries will then get the wrong answer relative to the present real business. This is the "Silent Killer." As a designer, you start out right, change nothing, and now you are wrong.

In the textbook theoretical literature written on the Relational Model, the operators used on databases are Insert, Replace, Delete, Retrieve, Select, Project, and Join (and to be complete - Product, Union, Intersection, Difference, and Divide). This causes some confusion with those

of us that know SQL. The Retrieve, Select, Project, and Join functions are all performed by the SQL SELECT operator. Just to reduce the amount of re- statement and/or translation from Codd's original source, the following uses the Relational Model terminology.

The following is the Relational Model's definition of Select, Project, and Join:

- **Select** - The Select function takes whole rows from a single table and creates an answer set consisting of some (or all) of the rows
- **Project** - The Project function takes whole columns for a single table and creates answer set consisting of some (or all) of the columns.
- **Join** - The Join function takes whole rows form two or more tables and creates an answer set consisting of concatenated rows that pass the join criteria. The join criteria is usually that two or more columns in the source tables' rows are equal.

Any combination of relational operators can be applied to re-form how the data is distributed, provided that the total transformation is reversible. The issue is verifying this reversibility. To not cause inaccuracies and anomalies, each updateable physical data store must be reversible to the well- formed normalized data model. The following is not intended to be exhaustive and consists of only a few of examples of well-formed denormalizations.

6.43.9 Over Normalization

This is achieved via the project relational operator. Each projection to be stored in some table, possibly another database, must include the primary key of the relation from which the projection is made. Then, each and every projection is well- formed with no duplicate rows. A table can be split into any number of projections. Note that, at any time, the original table can be recovered using equi-join with respect to the primary keys of the new denormalized tables being joined. In applying relational technology to the management of a denormalized database, it is essential that it be possible the relational operators can be used to decompose relations in the global database into relations for the various denormalized target tables. In other words you should be able to use SQL for the decomposition. This doesn't mean that you must use SQL for every decomposition, only that you could. You might have a non-SQL Extract Transform and Load (ETL) engine interfacing one or more OLTP systems with an Operational Data Store.

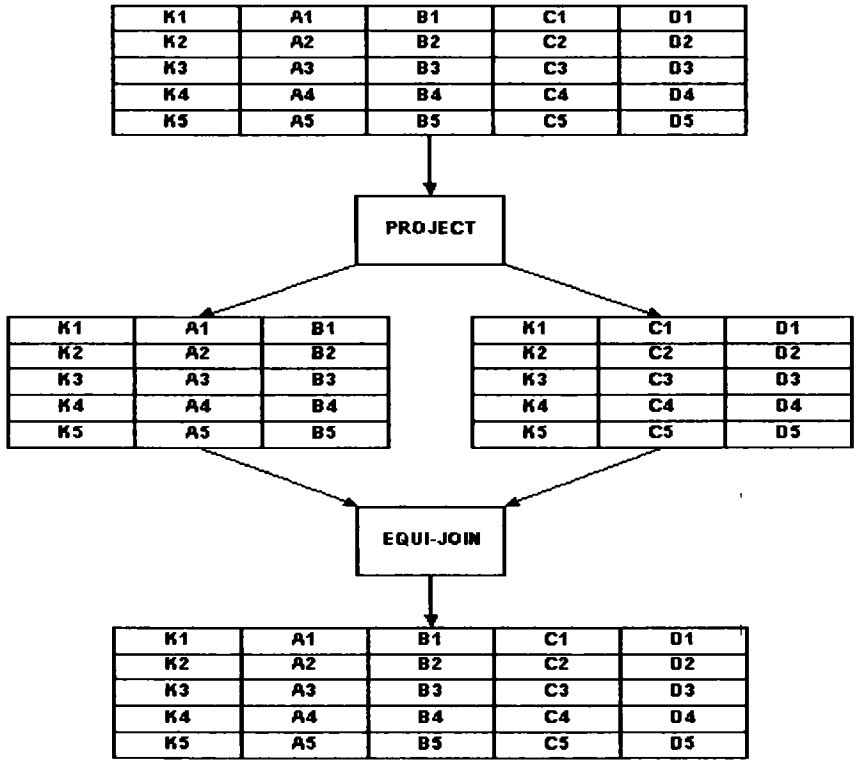


Fig 6.119 Over Normalization

If the key in the above example was very long and compound, a surrogate key could be substituted. And one of the vertical segments could be stored with the surrogate key only.

- Horizontal Segmentation

The select relational operator is used to insert some rows of a table into another table, and other rows to other tables. The selection of rows cannot be arbitrary. The selection of rows must be made using the select operator. When a table is partitioned by rows to be stored in other tables, union is used to recover the original table.

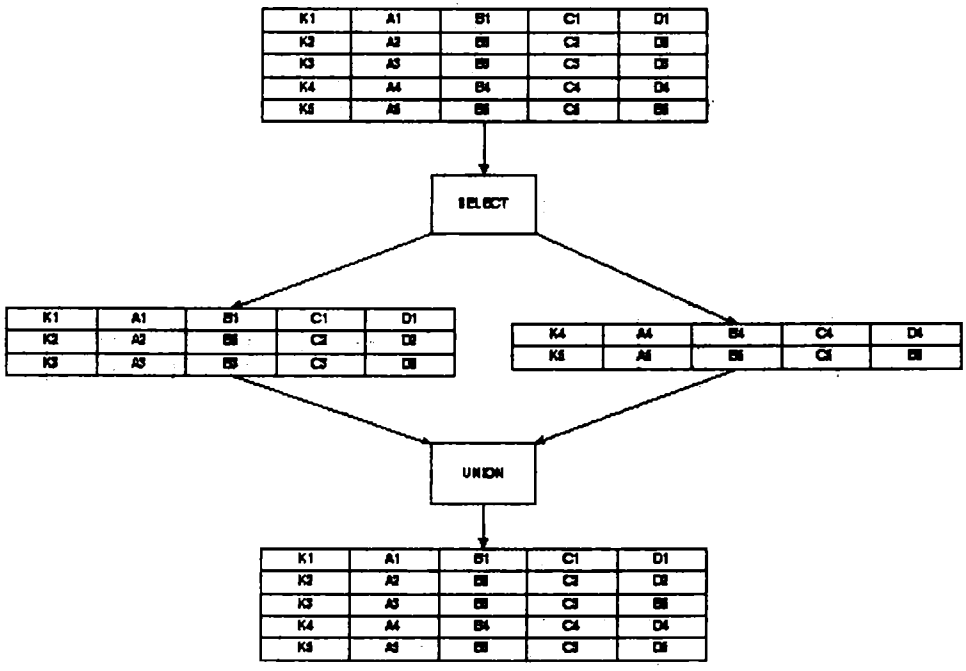


Fig 6.120 Horizontal Segmentation

- Stored Joins

The use of stored joins as a denormalization will formally offend the rule of reconstruction if the joined-table is updated. Since this is a very common if not the most common denormalization some discussion is appropriate. If you must update a stored join, non- relational programmatic measures must be taken to see that data integrity is maintained. A simple example of this is to not update the duplicate columns. It is always dangerous to have the data integrity rules outside the data domain. When other update programs are added to the system they must also contain the non-relational data integrity processes. It is better software engineering to have all the data integrity rules in the DBMS in the form of constraints, triggers, and stored procedures. Quality almost always implies simplicity in usage and a 'minimum number of moving parts.' If you have to write complex code with a relational database to get the right answer you are allowed to leap to the conclusion that the database design does not map to the business.

6.44 Reverse Engineering of Databases

It is rare that the data architect encounters a true 'green field' situation, with the luxury of being able to develop a new database from scratch, employing first principles, and using established principles and techniques of data administration. More often than not, the architect is faced with modifying an existing system, or at least with understanding an existing system in order to extract data from it. An invaluable aid in such circumstances will be an E-R diagram and/or data dictionary. However, it is often the case that the database is poorly documented, with not a hint of an E-R diagram, let alone having the luxury of a data dictionary. On these occasions, there is little alternative but to attempt to obtain a data model by a process of reverse

engineering from the database. In this note, we will outline a methodology that we have used on a number of occasions to some success and hope it may be of some assistance to you when faced with this situation.

The first step in the reverse engineering process is NOT to look at fancy database tools which have a 'reverse engineering' facility. This is like getting in to your car and turning on the ignition without having figured out where you are going!

The real first step in the reverse engineering process is to understand the system. Ask the questions: what does it do?; what are its inputs/outputs?; what data transformation occurs and how is this likely to be achieved? This is done in the time honoured tradition of talking to developers and users; reading what documentation exists; looking at screens, reports etc. In short, treat the database as a classical black box and from its inputs/outputs formulate a tentative hypothesis as to what sort of data may be stored within and what sort of data transformations are involved. You will probably be wrong! But primed with this knowledge, the process of reverse engineering is more focused and less tentative.

Next, as with any difficult problem, is to see whether it can be broken down into smaller, more managable chunks. Can the system be readily split into identifiable sub-systems? A distinction must be made between physical and logical implementation. Where a separation has been enforced, by means of tablespaces (Oracle) or databases (Sybase), then there is a clear partition of areas and each can be treated separately. However, it is often the case that several logical sub-schemas are physically located within the same database/tablespace, particularly if some loose coupling, i.e. a shared table, exists between them. In this instance, unless the sub-schema tables have already been identified, it is not possible initially to treat them independantly. During the reverse engineering process these sub- schemas will tend to be found naturally, by inspection of the relationships amongst tables. Each sub-schema can then be analysed in some detail.

Having acquired a preliminary understanding of the system, its data and the types of entities that might be expected, now is the time to consider tools. It is possible to reverse engineer with the database vendors own meta data query tools and some simple desktop utilities, spreadsheets etc., but frankly, this is a tedious job. A number of database design tools exist which specifically provide reverse engineering facilities by extracting the databases's meta data and producing an E-R diagram and a data dictionary. I feel that for realistic sized databases such a tool is necessary. But do not be misled. Although, these tools generally provide a first cut at an ER diagram and give a data dictionary of tables/columns/triggers/stored procedures etc., there will be two areas where problems tend to arise.

Firstly, the layout of the automatically produced diagram will not reflect the logical arrangement of the tables. Typically, the layout of the diagram will have to be adjusted so that the entities are rearranged in a way, which reflects the logical and functional groupings within the database. Usually, the tools will allow you to do this in a simple point and click manner. This is trivial but time consuming.

More importantly, the relationships automatically inferred between entities will usually be tentative. If you are lucky the database developer will have utilised any self documentation facilities within the database, however, this is a rarety (if the developer was that thorough it is unlikely that you would have to be doing a reverse engineering exercise!). Some reverse engineering tools can utilise such self documentation features in order to infer relationships. In the absence of explicit relationship information some other means of deriving this information is required. Some tools will attempt to infer the relationships by assuming that primary keys match unique indexes on tables and that foreign key relationships are thus defined either by non-unique indexes or by matching field names. The results of such automated relationship generation may be useful as a guide but will generally be wrong. Typically more relationships are produced than is really the case and the directions of the relationships may be incorrect. Techniques for pruning these 'relationships' are discussed below.

Assuming that a suitable tool has been used and a first cut schema produced, then tables should be grouped together by related business areas. It is worth scanning the table names to see if some naming convention has been employed which makes it obvious that tables are related in some way. An alternative strategy, is to isolate those tables which seem to have a large number of relationships with other tables. The level of connectivity between tables is usually not constant, the most significant tables often have many relationships with other tables which are not so well connected. There will often be only a few relationships between the 'hot' tables. Redraw the E-R diagram to show clusters of tables tightly coupled with only a few relationships between clusters. Each cluster will tend to correspond to a particular business process function.

There will always be a number of tables which do not seem to be related to anything. It is better to put these to one side and arranging in some systematic maner, perhaps by name.

During this grouping phase full use should be made of any additional information that is available, for example functionality may dictate that certain tables will be related, or inputs/ outputs will determine the attributes of some tables.

With the tables grouped together by business function, the identification of the purpose of each table will greatly assist in the determination/validation of relationships. In order to help with this I have found it useful to classify tables acording to the following taxonomy:

Table 6.2: A Taxonomy of Table Types

Type	Description Typical Characteristics Relationships to other Tables
Business Entities	Maps directly onto real business entities, e.g. personnel, product. Often named after concrete nouns with relevance to the business area. Primary Key (PK) often a simple reference number/code, e.g. employee_number. Usually has Foreign Key (FK) relationships to other Business Entity tables. Cardinality same as business entity. Likely to be normalised to 3NF.
Reference Data	Allowable list of codes, pick lists, domains. Number of fields << number of values. Small no. of rows compared to related business entities. PK often code/integer single fields. Description/text column often included. The PK will be a FK to one or more business entity table(s).
Assocation	Resolve many to many relationships. Small number of columns, most of which, perhaps all, are part of the PK. May be description column. Links two or more business entity tables. PK consists of a combination of FK from two (or more) business entity tables.
Summary/Reporting	Data Summarised/aggregarted data Has a composite PK consisting of dimensions along which the non PK columns can be grouped e.g. time, type, employee. Heavily indexed with wide indexes. Non key field tend to be numeric and additive. Large number of entries. Not directly linked to Business Entity tables. Columns within PK will be linked to reference tables. Interface Used to transfer data from one system to another, e.g. bulk inserts of data into database or transfer data out of database. No PK. Typically not indexed. Not normalised. Large number of rows. Mimics data feeds in/out. All fields may be char. Standalone.
Intermediate	Result Used as persistent storage for applications. Keeps data in a convenient form for application. Denormalised/restructured/derived data that is already available in the business entity tables. Not heavily indexed. Number of rows of a similar order to business entity tables. Standalone - but often share same PK as a business entity table.
Helper	These are similar to the intermediate result tables, the difference is that additional business information is stored which is not application specific but of general use. For example, when storing hierarchies it is often useful to construct a table of all valid paths through the hierarchy - strictly speaking this contributes no new information but greatly simplifies some queries. Similar to intermediate result tables but tend to be normalised with respect to business entity tables. Will be linked to one or other business entity tables.
Audit/History	Used to store a time history of changes/events to data. PK will contain time stamp columns, e.g. start/end dates. Other columns will also be found in business entity tables. Will not be explicitly linked to business entity tables but will "mirror" them.
System/Control	Used to control behaviour/access/security operation of system. Columns tend to be simple codes or integers. Small number of fields and rows. No obvious relation to business entity tables. Standalone.

Establishing the type of each table within a schema, according to the above classification, enables its location within the topology of a schema and its connectivity with other tables to be determined or validated. For example, if a table appears to have no relationships with other tables it may be a system/control table, or an audit/history table.

Alternatively, it could be that it is a reference table onto a business entity table and that the appropriate relationship has not yet been identified. Attempting to classify the table according to the above taxonomy, by inspection of its columns and indexes and general structure will often clarify the situation.

There is no doubt that correct identification of relationships is the hardest part of the reverse engineering process, but is the most important if the description of the database is to be more than a simple listing of tables and columns. Without this, a true understanding of the database cannot be achieved. In addition to the table taxonomy, there are a few "rules of thumb" to be useful:

Primary keys will often correspond to unique indexes. Be aware though of physical implementation issues, such as the use of index covering, where additional columns which are not properly part of the primary key, are added to the index in order to improve query performance; or where a number of alternate indexes are defined to support common joins/queries. Foreign key relationships are harder to determine, however some implementation features that can provide an indication of these relationships are:

- **Matching columns names** - where a column is a primary key in one table and occurs in another table, then a foreign key relationship may exist. If a column is in the primary key of both tables then the nature of the tables (as in the table taxonomy) needs to be examined in order to determine whether a FK relationship really exists.
- **Referential constraints** - either imposed explicitly via constraints or programmatically via triggers or stored procedures. Non-unique indexes - often these are defined in order to expedite joins across tables, commonly these will be along FK relationships.
- **Views** - in a similar vein as to non-unique indexes, views will often be formed by the merging of tables with FK relationships. If the database has a full realistic population of data, (as in a production system), this can greatly assist the foreign key validation process by providing a means of checking the overlap of values between differenct columns. For example, if we think that table/column t2(c2) is a foreign key from table/column t1(C1), we can use the following SQL to determine whether or not all the values of c2 are constrained within c1: select count(*) from t1 where not exists (select * from t2 where t2.c2 = t8.c1). If a value greater than zero is retrieved then there cannot be a foreign key relationship between t2(c2) and t1(c1).

A word of caution: It often happens that in real databases, extraneous 'special' values are purposely added which do not strictly conform to FK relationships, for example to indicate a 'Not applicable' or 'Invalid' condition (these are often introduced in order to prevent confusion as to the meaning of the 'NULL' value). It is also quite likely that inconsistent data is

unintentionally present. The fact that you are having to perform reverse engineering could imply that referential integrity has not been particularly well enforced in the past and that there is a lot of 'dirty' data in the database. So we may have the situation where t2.c2 <> t8.c1 for just one example, whilst in 99% of the time there is a perfect correspondence. How you choose to deal with this depends upon the purpose of the reverse engineering exercise. If you are trying to determine the logical model then you should treat an almost perfect correspondence as a FK.

By now most of the pieces of the jigsaw puzzle are now in place and a tentative outline schema should have emerged.

From this point forward the reverse engineering process is now more art than science. It is necessary to use all the available information to put the pieces together: the business process, mapping of business entities onto database tables, business constraints onto relationships and the interaction between the database and other systems. As the tables slot into place, it is always necessary to validate the model by comparing with the expected functionality and examining real production data.

In this short note, we have outlined some of the techniques that we have used in the past to perform reverse engineering of databases. This is something which we have had to do more often that we would have liked, but by approaching it in a systematic manner the work overall effort required can be guided and hopefully reduced.

6.45 Good Database Design

It is most likely that as a web developer, you will be working with one of the modern relational databases and that you will be able to work in conjunction with an existing database administrator. That is, this tutorial is limited to the "use" of databases rather than to the creation and administration of them. In fact, the creation and administration of databases is a topic well beyond the scope of this tutorial and probably well beyond the scope of your job. After all, database administration is its own entire job description.

However, we have been spending a lot of time going through general database theory because although you may not be designing databases yourself, in order to take the most advantage of them, it is important that you understand how they work. Likewise, it is important that you have a feel for good database design. After all, a database's usefulness is directly proportional to the sleekness and efficiency of its design. And your ability to take advantage of a database is directly proportional to your ability to decipher and internalize that design.

6.46 Designing DBMS for Enterprises

If you ask an application developer what the most important task is in developing new or enhanced applications for institutional data and processes, almost every time they will tell you it is the initial analysis of client requirements. Before purchasing any software and before storing a single byte of data in a database, analysis of the client's requirements is paramount to developing the appropriate solution. More time spent in analysis directly increases the effectiveness of the resulting application. Since the early 1960s, and despite the waves of change since then, one thing has remained constant — the initial analysis is still the most important activity that an application designer undertakes. It gives the developer the chance to design an effective, spectacular application, no holds barred.

This analysis takes on various forms. Usually the application developer has a feeling about what form the analysis should take. It may simply require a phone call to the client asking them "Do you want to add or subtract 5 percent from all the employees' salaries?" Or, it may require the organisation of week-long meetings with clients to collectively analyse their requirements. Overkill is rarely a problem in the analysis stage as it guarantees the involvement of all the relevant people. The worst thing a developer can do is to not include a key person in the requirements analysis. Everyone's knowledge and experience is needed during this analysis. Their presence or absence makes or breaks the success of the analysis.

The participants in the analysis bring their much-needed knowledge and experience into the meeting, but it is also important to ask them to "leave their baggage at the door." Excess baggage such as idealisation of the features or constraints of the current application can impede the design of a new and improved application, one without those same "time- honored" constraints. While the developer recognizes that there are always rules, regulations, and constraints, they must also examine these constraints for their continuing validity within the new application.

CHAPTER 7

NETWORK MODEL

7.1 Network Model Overview

A network data structure can be regarded as an extended form of the hierarchic data structure - the principal distinction between the two being that in a hierarchic structure, a child record has exactly one parent whereas in a network structure, a child record can have any number of parents (possibly even zero). A network database consists of two data sets- a set of records and a set of links- where the record types are made up of fields in the usual way.

Networks are complicated data structures. Operators on network databases are complex, functioning on individual records, and not sets of records. Increased complexity does not mean increased functionality and the network model is not powerful than the relational model. However, a network-based DBMS can provide good performance because its lack of abstraction means it is closer to the storage structured used, though this is at the expense of good user programming. The network model also incorporates certain integrity rules.

7.2 Network Databases

Network or linked databases organize dissimilar records using linked lists to implement relationships between the rocords. These are constructed based on pointers and links between data records (many-to-many relationship).

In many ways, the Network Database model was designed to solve some of the more serious problems with the Hierarchical Database Model. Specifically, the Network model solves the problem of data redundancy by representing relationships in terms of sets rather than hierarchy.

The model had its origins in the Conference on Data Systems Languages (CODASYL), which had created the Data Base Task Group to explore and design a method to replace the hierarchical model.

The actuality, the network model is very similar to the hierarchical model. In fact, the hierarchical model is a subset of the network model. However, instead of using a single-parent tree hierarchy, the network model uses set theory to provide a tree, like hierarchy with the exception that child tables were allowed to have more than one parent. This allowed the network model to support many-to-many relationships.

Visually, a Network Database looks like a hierarchical Database in that you can see it as a type of tree. However, in the case of a Network Database, the look is more like several trees, which share branches. Thus, children can have multiple parents and parents can have multiple children.

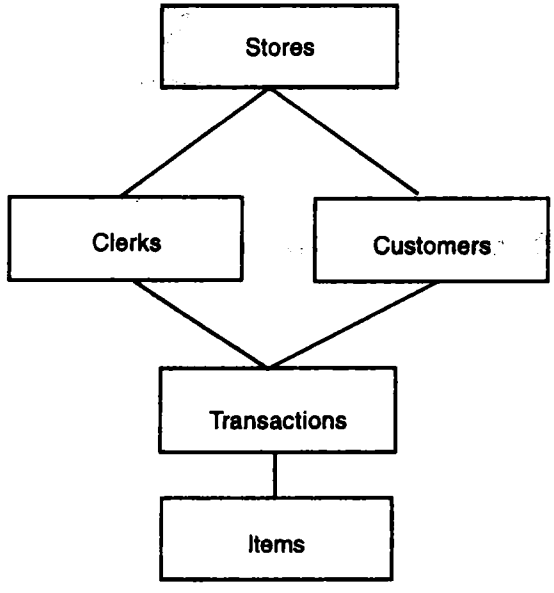


Fig 7.1 : Network Database

Nevertheless, though it was a dramatic improvement, the network model was far from perfect. Most profoundly, the model was difficult to implement and maintain. Most implementations of the network model were used by computer programmers rather than real users. What was needed was a simple model, which could be used by real end users to solve real problems.

Network databases are similar to hierarchical databases for also having a hierarchical structure. However, there are a few key differences, however. Instead of looking like an upside down tree, a network database looks more like a cobweb or interconnected network of records. In network databases, children are called members and parents are called owners. The most important difference is that each child or member can have more than one parent (or owner).

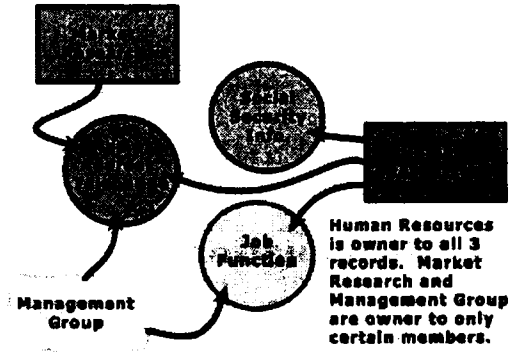


Fig 7.2

Like hierarchical databases, network databases are principally used on mainframe computers. Since more connections can be made between different types of data, network databases are considered more flexible. However, two limitations must be considered when using this kind of database. Similar to hierarchical databases, network databases must be defined in advance. There is also a limit to the number of connections that can be made between records.

7.3 Network and Internet

These are further development of linkage systems with many more record types and linkages. Both the linkage systems require the user to know what linkages have been established in order to know on what basis data can be retrieved. The linkage systems are technically more efficient but do require greater data processing knowledge so their use tends to be restricted to information specialists.

The ability to store and access vast amounts of data in an efficient manner is being used to refine and improve activities at stategic, tactical and operational levels. For example the UK Clearing Banks use databases in marketing. Proviously data were collected on a branch basis maninly relating to current accounts. If a customer had a loan or mortagage this was recorded separately. Now the banks adopt an integrated database approach so that all information about a customer is maintained centrally so that a financial profile of a customer can be derived and used for marketing.

Database marketing in retailing is well developed in America and is being explored in this country. In America it has developed to the point where the detailed buying patterns of individual shoppers are recorded making possible the precise targeting of marketing information, special offers etc.

7.4 Network Database Records

A network DBMS contains a set of multiple occurrences of each of several types of records together with a set of multiple occurrences of each of several types of link. Each occurrence of a record type contains a set of values for fields, as in conventional programming languages. The network model makes no requirements on field values being drawn from certain domains, as is done in the relational model.

7.5 Network Data Manipulation

A network data manipulation language consists of a set of operators for processing data represented in the form of records and links. Examples of such operators include the following.

- Locate a specific record given a value of some of its field(s)
- Move from a parent to its first child in some link
- Move from one child to the next in some link.
- Move from child to parent within some link.
- Create/delete/update a record.
- Connect an existing child into a link.
- Disconnect a child record from a link.
- Disconnect an existing child record from one occurrence of a given link type and reconnect it to another. etc.

These operators are typically all record-level (as suggested by the examples themselves), as in the hierarchic model.

7.6 Network Model Integrity

Like the hierarchic model, the network model includes built-in support for certain types of referential integrity, by virtue of its primary data structure, the link. It is therefore possible, for example, to enforce the rule that a child cannot be inserted unless its parent already exists.

Network Databases

In many ways, the Network Database model was designed to solve some of the more serious problems with the Hierarchical Database Model. Specifically, the Network model solves the problem of data redundancy by representing relationships in terms of sets rather than hierarchy. The model had its origins in the Conference on Data Systems Languages (CODASYL) which had created the Data Base Task Group to explore and design a method to replace the hierarchical model.

The network model is very similar to the hierarchical model actually. In fact, the hierarchical model is a subset of the network model. However, instead of using a single-parent tree hierarchy, the network model uses set theory to provide a tree- like hierarchy with the exception that child tables were allowed to have more than one parent. This allowed the network model to support many-to-many relationships.

Visually, a Network Database looks like a hierarchical Database in that you can see it as a type of tree. However, in the case of a Network Database, the look is more like several trees which share branches. Thus, children can have multiple parents and parents can have multiple children.

Nevertheless, though it was a dramatic improvement, the network model was far from perfect. Most profoundly, the model was difficult to implement and maintain. Most implementations of the network model were used by computer programmers rather than real users. What was needed was a simple model which could be used by real end users to solve real problems.

Network databases are similar to hierarchical databases by also having a hierarchical structure. There are a few key differences, however. Instead of looking like an upside down tree, a network database looks more like a cobweb or interconnected network of records. In network databases, children are called members and parents are called owners. The most important difference is that each child or member can have more than one parent (or owner).

Like hierarchical databases, network databases are principally used on mainframe computers. Since more connections can be made between different types of data, network databases are considered more flexible. However, two limitations must be considered when using this kind of database. Similar to hierarchical databases, network databases must be defined in advance. There is also a limit to the number of connections that can be made between records.

DATABASES FOR WEB

8.1 Designing Data Bases for Web

It allows a lot of flexibility in how your data is stored and represented, yet it still lets you build up fairly comprehensive structured data objects. A lot of pre-built libraries handle these structures, and some languages are moving toward providing native support: Perl and Omnimark come to mind. However, the problem is the source of your data. Do you expect people to provide you with existing XML? Not too likely. Also, there might be too much flexibility. But you may be willing to give up some of that control for the sake of speed and simplicity.

8.2 Database Servers

You've heard the names before: Oracle, Sybase, Microsoft SQL Server, Informix, Ingress. Conceptually, these programs offer a grab bag of features and technologies that synergize nicely:

- They store data in tables. Tables are remarkably similar to fixed-width text files, but their fields (also known as "columns") can contain many different structured data types. Examples are integer, character string, money, date, and Binary.
- Large OBject (BLOB).
- They provide administrative facilities to manage tables.
- Tables and administrative facilities are guarded through sophisticated user/password/domain protection schemes.

You can interact with data through powerful-yet-relatively-easy languages, SQL for example. Even better, you can build stored SQL statements that your users can take advantage of without

having to know the language themselves. A final advantage of databases is that so many people before you have used them; you will find that virtually all of your seed data will already be in some kind of database format.

8.3 Why the Web?

Why do somebody will want to be on the Web? Because that's where your audience is. To be fair, we should take a look at why our audience is on the Web. Also, maybe you are not developing for the world at large. You could be building a corporate intranet Web site. In that kind of situation, you get to decide how you want to distribute information to your users. (Well, your manager gets to decide, but that's a different story.) Let me give you a few points to consider.

Even in a corporate environment, Web browsers are everywhere. That makes the Web a really convenient information vehicle. Browsers are portable. You don't need your computer and your personal software at a particular physical location to get to your data. And neither do your users. Web interfaces to databases are very quick and easy to program, even compared to such rapid development languages as Visual Basic. You don't have to concern yourself with the logistics of distributing custom software to either internal users or to an infinite number of external (Internet) users. You don't have to worry about supplying everyone with upgrades. Instead, you upgrade the production software on the server, and instantly everyone benefits. You don't have to train people to use your software - almost everyone knows how to work a Web browser and fill out forms. Sure, proprietary technologies that help you create database interface systems exist, like Visual Basic or Visual C++, Powerbuilder or Developer 2000. You can create some great interfaces with these tools. But most of the time, the pro-Web reasons we mention above win out, so that's what we are going to teach you how to do.

8.4 Apache Web Server

The Apache Web server is quite a piece of work. Put simply, its role is to allow users to request Web information from your computer. Most of the time this information is in HTML documents and image files (GIFs and JPEGs primarily), but all modern Web servers are required to have CGI (Web programming) support as well. That is the simple version of Apache's job. But if a particular application or demand crops up, you might need to take that simple version and configure the bejeesus out of it.

Apache is powerful, highly configurable, and heavily documented.

Custom Apache Configurations

Three reasons why we will be making our custom Apache configurations:

- Our choice of DocumentRoot directory
- ePerl
- mod_perl

DocumentRoot is an easy enough idea to grasp. Let us say we have set up the domain www.example.com. Then, URLs that might "hang off" it would look like:

```
http://www.example.com/webmonkey/day3.html
```

It it better if you decide on setting up the my DocumentRoot as /web/docs/, so the file corresponding to that URL would be /web/docs/webmonkey/day3.html. Assuming you took my advice from yesterday as to where to install Apache, you will have to edit /usr/src/apache_8.2.6/conf/srm.conf to set DocumentRoot to /web/docs. (Don't worry, once you look at the file, it's obvious how to do this.) You will also need to do a corresponding edit to /usr/src/apache_8.2.6/conf/access.conf. (Again, no big deal.) At the Unix prompt, be sure to actually create your DocumentRoot (likely /web/docs/) directory, too:

```
mkdir /web; mkdir /web/docs
```

Perl and ePerl together with mod_perl, make up a world-class development toolkit for the creation of database-activated Web pages. Yesterday's instructions suggested using mod_perl's automatic Apache build feature, so you should now have a file called /usr/src/apache_8.2.6/src/httpd.

OK. Now here's what you do:

Step 1:

Create a symbolic link between

```
/usr/src/apache_8.2.6/src/httpd and /usr/sbin/httpd.
```

The idea behind the creation of a symbolic link is that your Linux operating system expects httpd to reside in the /usr/sbin directory, but any future re-builds of httpd will put it in /usr/src/apache_8.2.6/src instead. This trick will save you from having to remember to copy httpd into /usr/sbin each and every time.

```
cd /usr/sbin
ln -s /usr/src/apache_8.2.6/src/httpd httpd
```

Step 2: Check to make sure that your build includes mod_perl.

We'll do a quick check right now, just to make sure. Type in the following command:

```
/usr/sbin/httpd -v
```

You should get a small message that says something like:

Server version Apache/8.2.6 mod_perl/8.08.

Step 3: Configure your /etc/rc.d/rc.M file.

This file is like the good old DOS autoexec.bat file. It runs as soon as your Linux server starts, making sure that all the programs that need to be initialized and run at boot-time are started properly. If your rc.M file doesn't already have the following, you should put it in yourself:

```
# Start Web server:
if [ -x /etc/rc.d/rc.httpd ]; then
. /etc/rc.d/rc.httpd
fi
```

All this bit of code says is that the Linux start-up process should activate the contents of the file /etc/rc.d/rc.httpd (to start up your httpd). So here's what should go in that:

```
echo httpd
/usr/sbin/httpd -f /usr/src/apache_8.2.6/conf/httpd.conf &
```

You probably won't have this file yet, so go ahead and create it.

Step 4: Prepare Apache to handle CGI and ePerl documents. Get ready – you are in for a truckload of minor configuration tweaks. Navigate back into the /usr/src/apache_8.2.6/conf directory, and then:

In access.conf, make sure the text in bold is added:

```
<Directory /web/docs>
Options Indexes FollowSymLinks ExecCGI
AllowOverride None
order allow,deny
allow from all
</Directory>
```

Now, edit srm.conf:

DirectoryIndex index.iphtml index.cgi index.html

Also, you will want to "uncomment" a line so that the final version looks like:

```
AddHandler cgi-script .cgi
```

In httpd.conf, add the following just above the <VirtualHost> area:

```
Perlrequire /web/docs/startup.perl
PerlModule Apache::ePerl
<Files ~ ".+\.iphtml$">
SetHandler perl-script
PerlHandler Apache::ePerl
<Files>
```

Now, you will need a startup.perl file in /web/docs. Here, copy mine:

```
#!/usr/bin/perl
use strict;
use Apache::Registry;
use CGI;
use DBI ();
1;
```

To help you understand what we did in Step 4, here's a quick post-mortem.

By default, Apache won't let CGI programs run in any directory other than your duly-designed /cgi-bin/. When security is your priority, this makes a lot of sense. But, since our priority is Web database programming, the configs in Step 4 tell Apache that it's OK to run CGI programs anywhere underneath DocumentRoot.

Also, we taught Apache what to do when it runs across files with the funny extension .iphtml. These are "internally parsed HTML" files, and now Apache knows that they're to be handed off to ePerl for further processing. Finally, we told Apache to run a file called startup.perl as soon as the server initializes. This loads a few highly useful modules into the server's memory once and for all, so each subsequent program that needs these modules can access them without bothering to re-load them.

8.5 MySQL and That Whole Database/Server Thing

Database servers are memory-resident daemons that respond to requests, store data in smart ways, and provide administrative interfaces to make sure that the data is handled only in an

authorized manner. We will now use MySQL to practice these ideas. Once you've got MySQL built, you will have a lot less file configuration fiddling than Apache required. Once you've gone through the full MySQL installation procedure (including running /usr/local/src/mysql-VERSION/scripts/mysql_install_db), only one more task remains: set it up as a memory-resident daemon. This procedure is identical to what we did for httpd. Edit your /etc/rc.d/rc.M file to include the code:

```
# Start mysql database server:
if [ -x /etc/rc.d/rc.mysql ]; then
. /etc/rc.d/rc.mysql
fi
```

Create a corresponding rc.mysql file - very straightforward:

```
/usr/local/bin/mysql.server start
```

When we are working with MySQL, we use two programs all the time: /usr/local/bin/mysql and /usr/local/bin/mysqlshow. Don't worry about typing all that in - mysql and mysqlshow should work just fine, since /usr/local/bin is part of the command path environment variable.

Let us try it:

```
rdice:# mysqlshow
```

Databases
mysql
test

This is what you should see (assuming you have Perl and Data::ShowTable correctly installed). This output shows us that, at the highest level, MySQL organizes its data into databases. The two shown above are created automatically by MySQL and each serve a special purpose: mysql is used by MySQL itself to organize the program's own internal settings, while test is made available to all users as a sort of scratch-pad area. It is fully functional, but it comes with no protection or authorization schemes. In other words, don't put anything important into test in case one of your rocket scientist coworkers decides to nuke it. Let us do it again, but this time we will ask mysqlshow to tell us the contents of the test database:

rdice:# mysqlshow test

Database: test
Tables

Tables are the next level down after databases. Think of tables as spreadsheets: columns represent data fields and rows represent individual entities or records. From our output, we can see that the test database is empty, which isn't surprising since you have a brand-spanking-new installation of MySQL. We'll use the mysql program to work interactively with MySQL and actually put some things in there.

8.6 MySQL, SQL, DDL, and DML

For the most part, interacting with MySQL means speaking its language: SQL, the structured query language. By and large, SQL is divided into two main parts. The first is DDL, the data definition language. You use this part of SQL to tell MySQL how to set up tables. There is also DML, the data manipulation language, which you use to get at the data housed in your tables. Here's the plan. we will start up mysql, create a table, put some data in it, and then review the data that we just put in.

```
rdice:# mysql test
Welcome to the MySQL monitor. Commands end with ; or \g.
Your MySQL connection id is 4 to server version: 3.28.28-
gamma-log
Type 'help' for help.
mysql> create table albums
-> title varchar(100), to set up a table where
-> artist varchar(100),
-> released date); musical info
Query OK, 0 rows affected (0.07 sec)
Now, here comes some DML to tell MySQL to insert a record into
the albums table
mysql> insert into albums(title,artist,released)
-> values('Selling England By The Pound','Genesis','1973-01-01');
Query OK, 1 row affected (0.08 sec)
note the * - it means "all columns" in SQL-speak
mysql> select * from albums;
```

title	artist	released
Selling England By The Pound	Genesis	1973-01-01

1 row in set (0.06 sec)

The exact syntax of all SQL commands accepted by MySQL is covered very well in the MySQL documentation, so it will all be there when you need it. But even today's small session covers a lot of the fundamentals, so it's worth your while to take a close look at it. To create a table, the basic syntax is:

```
create table TABLENAME (
COLUMN1 column1datatype,
COLUMN2 column2datatype,
...
COLUMN_x column_xdatatype
)
```

The formatting is optional. we find that spreading it across several lines makes it easier to read, but it's not mandatory. The allowed column data types can be found in the MySQL documentation. In this example, we use the "date" data type as well as variable length character strings, which represent album and artist names. "varchar" strings can be of any length up to a pre-declared maximum: in this case, 100.

To end a statement in mysql, you type a semicolon followed by ENTER. mysql will then tell you the status of your "query" (its term for any command), and how long it took to process. The insert syntax is even more straightforward:

```
insert into TABLE(column1,column2,...,column_x)
values(value1,value2,...,value_x)
```

Character strings must be quoted, but that's to be expected from any programming language. select is the workhorse statement of SQL. It is used all the time, and it can be quite sophisticated when necessary. It can be used for simple tasks (as shown in the above example), but we will take it a bit more seriously now.

If we had a few hundred albums stored in the database, and we only wanted to see the ones that were made by Genesis, we could easily create a select statement that would let me get at exactly what we wanted:

```
select title, released
from albums
where artist = 'Genesis'
order by released
```

This time, we specified the columns that interested me by name rather than with a * wild card. Since the line where artist = 'Genesis' will restrict the records MySQL returns to those albums performed by Genesis, it wasn't necessary to see the artist column explicitly named in the select list. Also, we thought it would be keen to have MySQL order the records it returns by release date, just in case we didn't know that Wind & Wuthering came before We Can't Dance. (shudder) In addition to insert and select, there are two other basic DML commands: update, which lets you change the data stored in a table row, and delete, which lets you remove a row from a table altogether.

Here is how to use these commands:

```
I've just arbitrarily decided that the release date of
Selling England is now 1 February rather than 1 January.
mysql> update albums
-> set released = '1973-02-01'
-> where title = 'Selling England by the Pound'
-> and artist = 'Genesis';
Query OK, 1 row affected (0.03 sec)
mysql> select * from albums;
```

title	artist	released
Selling England by the Pound	Genesis	1973-02-01
Wind & Wuthering	Genesis	1976-01-01
We Can't Dance	Genesis	1991-01-01

```
3 rows in set (0.00 sec)
What the heck. I never really liked Genesis anyhow ...
mysql> delete from albums where artist = 'Genesis';
Query OK, 3 rows affected (0.00 sec)
```

```
mysql> select * from albums;
Empty set (0.00 sec)
```

With both "update" and "delete" commands, you need to specify a where clause. (Otherwise every row in your table will end up modified or deleted, which is something to watch out for!) A pretty good SQL tutorial exists on the Web and it is recommanded that you go ahead and give it a go. It isn't 100 percent applicable to MySQL, but that's only because there are different dialects of SQL. This tutorial covers one that's fairly close to MySQL's dialect, but with a few minor differences (these can be accommodated by the MySQL SQL documentation).

Let us now take one last look at

```
rdice:# mysqlshow test
Database: test
   Tables
   albums
rdice:# mysqlshow test albums
```

Database: test Table: albums Rows: 0

Field	Type	Null	Key	Default	Extra
title	varchar(100)	YES			
artist	varchar(100)	YES			
released	date	YES			

mysqlshow has taken note of all the work we have done in mysql. When you ask mysqlshow to display table information, it tells you nothing about the data in the table apart from the number of rows currently residing in it. What it does tell you is the structure of the columns. We are familiar with "Field" and "Type" already, and the other columns relate to more advanced table information.

We now know the basics of how to work with a database. All that's left to do is come up with a way to "glue" these SQL techniques to the Web. We will do this with DBI, Perl's DataBase Interface. Essentially, we will write SQL statements directly into ePerl programs, and enable them with Perl's DBI module.

Don't worry - we have already got it set up and ready to go. All that remains is the coding.

8.7 The Embedded Web-Programming Philosophy

A few years ago, Web programming started out with the Common Gateway Interface, or CGI. Here is a quick review of the basics concepts of CGI.

When a user makes a CGI request of a Web server, something in the URL will tip off the server to process it as a CGI request. The hint in the URL could look like one of the following examples. The URL requested by the user is in a /cgi-bin/ directory:

```
http://www.somewhere.com/cgi-bin/arandomcgiprogram
```

The Web server might be configured to automatically recognize certain file extensions as being CGI executables: http://www.erehwon.org/gosearch.py .py commonly denotes Python programs, another popular language for Web programming. The file extension might be a "generic" CGI extension:

```
http://www.xyz.net/dosomething.cgi
```

In these cases, the Web server "hands off" to a program specified by the URL, spawning it as a child process and providing it with the information it needs to be a "Web program": generally environment variables and standard input (STDIN).

The program will run and produce information, which it sends to standard output (STDOUT). Usually, the program will then create a minimal amount of HTTP header information, at the very least, as part of its output.

The Web server will "capture" the STDOUT stream and redirect it to the user via the Web. The user's browser will interpret the information according to the HTTP header. This will usually be HTML text, but CGI programs can just as easily create byte streams to be reconstructed as JPEG images or RealAudio feeds. The canonical simple C program is:

```
#include <stdio.h>
int main () {
print("Hello, world!\n");
}
```

we can turn this into a CGI program in a snap by simply adding an HTTP header.

```
#include <stdio.h>
int main () {
print("Content-type: text/plain\n\n");
print("Hello, world!\n");
}
```

Now all that's left to do is compile this code and put the resulting binary in my Web directory structure and then set permissions appropriately. CGI is still used quite a bit in the Web world, but serious complaints have been made about it.

Spawning a child process is hard work and costs time and memory. This problem with speed has prompted many a complaint from producers of high-traffic Web sites. Web servers contain a lot more information than just environment variables and STDIN. It would be handy sometimes for Web programs to gain access to these extra.

The whole classical programming paradigm has proven cumbersome for most Web-programming needs. What you are really trying to do is write a program that will intelligently compose HTML on your behalf. So, why does it look like computer code? Why can't it look more like HTML? More modern ways of programming Web applications have arisen over the past few years. Their roots lie in server-parsed HTML, or .shtml, a programming option that's been on the scene since almost the beginning, but was never powerful enough for serious application programming. These languages and techniques revolve around embedding programming code into HTML files. Some popular examples are:

- Active Server Pages (.asp files), which are used by the Microsoft IIS Web server. ASP

files can be activated with several different scripting engines, including VBScript, JavaScript, and PerlScript.

- Allaire Cold Fusion, a very handy commercial Web-development environment. Though it started off only being available on the Windows NT side of the house, lately it is cropped up on the UNIX side as well.
- Meta-HTML, a "free software" product available for UNIX systems. It supports ODBC, as well as a native interface to mSQL, and provides software plug-ins to Netscape and Apache Web servers.

Here we are not going to touch any of these; instead, we will deliver Ralf Engelschall's ePerl, a program that allows you to embed Perl source code into text documents. It provides a few special facilities to help out with HTML in particular, since it's such a pervasive file format. Also, it integrates with mod_perl/Apache, which we did with the Apache configurations yesterday. The mod_perl/Apache combination addresses the speed and access-to-server-internals failings in the CGI-programming approach, and ePerl deals with the clumsiness of standard programming languages in the creation of HTML.

ePerl - Perl meets HTML

To be truly proficient with ePerl, you should know both HTML and Perl. You do know both, right?

Well, in case you don't, consider this a quick introduction. Let us start off by looking at a simple HTML document.

```
<HTML>
<HEAD><TITLE>A simple HTML document</TITLE></HEAD>
<BODY>
<P>
```

This is about as simple as it gets. No big deal.

```
</BODY>
</HTML>
```

Now, we will put in some embedded Perl code:

```
<HTML>
<HEAD><TITLE>A slightly less simple ePerl-HTML
document</TITLE></HEAD>
<BODY>
<P>
```

Just before a chunk of embedded Perl ...

```
<HR>
<?
my $index;
foreach $index ( 1 .. 10 ) {
print "Currently on loop index: $index\n";
}
!>
<HR>
<P>
... and now, we're just after the ePerl.
</BODY>
```

```
</HTML>
```

You can take a look at the output of this little gem here, but, really, the outcome should be obvious. we just set up a simple "for" loop to output a message indicating what loop iteration was performed. The ePerl start delimiter is <?, and the ePerl end delimiter is !>. Also, you can still output to the Web - inside an ePerl block - with the usual Perl print statement. ePerl really is that simple, as long as you know Perl. Only a few other things should be kept in mind with ePerl Web programming.

ePerl automatically detects the nature of your Web documents and outputs an appropriate HTTP content header. If your document is HTML, it outputs text/html. Otherwise, it simply outputs text/plain. When it comes to inserting the values of variables into HTML, ePerl offers a shortcut. The special delimiting sequence is:

```
<?=$VARIABLE!>
```

Following the !> end delimiting sequence, you can add // to prevent the <? ... !> block from outputting a new line. This can help clean up the HTML source that ultimately outputs to the Web. You can include other files into ePerl with an #include statement.

Here we should consider two more points before going any further with ePerl. They both have to do with mod_perl/Apache. mod_perl is a Perl interpreter linked into the Apache Web server. So, any Perl programs that mod_perl runs won't actually be their own program. They will be treated as a sort of subroutine of a primary, hidden Perl process. It speeds things up, but it also engenders a number of nasty name space issues: You don't want variables with the same name in different Perl scripts "colliding" with each other. ePerl is very strict about ensuring that this doesn't happen. To keep ePerl happy, declare all variables as my. This keyword tells Perl to place the variables in a private name space. It works. That's all we need here.

Second, you have to be careful with the use of #include statements in ePerl, since a problem can arise. Consider a small ePerl file, TT>time.iphtml:

```
<?
print scalar localtime;
!>//
```

We might like to include this into a larger ePerl .iphtml file:

```
<HTML>
<HEAD><TITLE>The Current Time</TITLE></HEAD>
<BODY>
<P>
```

The current time is: <B>

```
#include time.iphtml
</B>
</BODY>
</HTML>
```

A technique mod_perl uses to help speed up the operation of Perl programs under Apache is to cache their compiled states. Perl programs work by undergoing a two-phase interpret/compile, which needs to happen before running each and every Perl program. But mod_perl "remembers" the results of previous interpret/compile stages. It will only bother running through this stage again if the time stamp of the file has changed since it was last compiled, which is a very

reasonable thing to do. However, what would happen if the file time.iphtml was changed? mod_perl isn't very sensitive to changes made to included files. Thus you can expect erratic behavior in such circumstances.

8.8 DBI – The DataBase Interface for Perl

DBI is a very popular Perl module. No wonder. It is the gateway between Perl and SQL-driven databases. It allows you to perform database-administration functions from within Perl programs and, more importantly, issue SQL commands from within your Perl source code. For each database, there is a DataBase Driver (DBD) that links the generic DBI interface to your specific database server. This way, Perl programs written with DBI are quite portable. You can go from using one database server to another without having to change more than a few lines of Perl code. To gain access to the magic of DBI, you just need a line that says use DBI; at the beginning of your Perl program. If you have your mod_perl/Apache combination set up the way We learnt earlier, then you don't actually have to include the use DBI; statement in each ePerl file, since the start-up.perl file we discussed this facility to all Perl/ePerl programs that run through mod_perl. Actual DBI Perl programming can get rather repetitive and algorithmic, which makes it easier for me to explain.

First, create an object which will act as a "database handle." This object gives you something to reference all future SQL queries against, since it defines your database.

```
$dbh = DBI->connect('DBI:mysql:test:localhost', '','') or
die $DBI::errstr;
```

The three parameters involved with DBI->connect are $database, $username, and $password. Since we are using the test database - which we set up yesterday - we don't need to specify a username and password. You can use the string constant DBI:mysql:test:localhost as your $database parameter, but only in this very limited case. If you start doing different things with different databases, you will need to refer to DBI and DBD documentation to help you decide on a new $database string. Next, write some SQL code and put it into a variable. For instance, consider the following example:

```
$SQL = <<"EOT";
select title, released
from albums
where artist = 'Genesis'
order by released
EOT
```

This variable will become the core of a client-side cursor. A cursor is a special kind of advanced SQL query which is executed one row at a time. That's not really what's going on here; the query is indeed executed all at once, but our Perl program only has the ability to step through the results of the query row by row, so it "feels" like a cursor to applications programmers like us. The client-side cursor is declared and executed with:

```
$cursor = $dbh->prepare($SQL);
$cursor->execute;
Now, we step through it one row at a time:
while ( @columns = $cursor->fetchrow ) {
```

```
print ( ( map { "[$_]" } @columns ) , "\n");
}
```

All that we are doing with this Perl code is printing out each entry in the columns array - the values of which are extracted from the $cursor row with the $cursor->fetchrow method - surrounded by square brackets, []. Obviously, we could have put anything in the while loop, not just print statements. Lastly, to recycle system resources and make clean disconnections, we want to close-off our cursor and database handle.

```
$cursor->finish;
$dbh->disconnect;
```

If the particular SQL command you wanted to execute was something other than a select statement, you wouldn't have to bother with the while ($cursor->fetchrow) { ... } loop. Since you didn't actually request any returned information, no rows for you to loop through are here. Let's say I didn't delete my Genesis information in the albums database yesterday. we can take all of these ideas and turn them into a Web-ready ePerl program.

```
<?
use DBI; # in case you don't have the startup.perl file going
my $dbh = DBI->connect('DBI:mysql:test:localhost',
'','')
or die $DBI::errstr;
my $SQL = <<"EOT";
select title, released
from albums
where artist = 'Genesis'
order by released
EOT
my $cursor = $dbh->prepare($SQL);
$cursor->execute;
!>//
<HTML>
<HEAD><TITLE>ePerl/DBI/HTML Integration
Example</TITLE></HEAD>
<BODY>
<P>
```

The results returned from the database query regarding Genesis albums in the database are ...

```
<HR>
<TABLE BORDER>
<TR><TH COLSPAN=2>Albums by Genesis</TH></TR>
<TR><TH>Title</TH><TH>Release
Date</TH></TR>
<?
my @columns;
while ( @columns = $cursor->fetchrow ) {
print ( "<TR>",( map { "<TD>$_</TD>" }
@columns ) , "</TR>\n");
}
!>//
</TABLE>
```

```
<HR>
<P>
... and that's it!
</BODY>
</HTML>
<?
$cursor->finish;
$dbh->disconnect;
!>//
```

8.9 The Unavoidable CGI.pm

You will find Lincoln Stein's CGI.pm Perl module popping up in all sorts of discussions about Perl CGI programming. In general, CGI.pm utilizes a philosophy that's completely opposite.

From the beginning, my intention was to help you create interactive, dynamically generated Web pages that look like Web pages from a developer's point of view. That's what embedded HTML programming is all about. And hopefully this also makes it easier and faster to code and organize sites.

The CGI.pm viewpoint is to abstract the heck out of your Web programming. To give you a feel for what it's like to construct a Web page using CGI.pm.

```
#!/usr/bin/perl
use CGI;
$query = new CGI;
print $query->header(),
$query->start_html(-title=>'Made with CGI.pm'),
'This is what I mean by ',
$query->b('abstracted'),
'.',
$query->end_html();
exit 0;
```

The output of this program is, predictably enough, a simple HTTP header, a starting HTML block (including a <TITLE>), a bit of text, and finally the standard HTML closing block. Note that this is all generated through the CGI object $query and its associated methods:

```
Content-type: text/html
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML//EN">
<HTML><HEAD><TITLE>Made with CGI.pm</TITLE>
</HEAD><BODY>This is what I mean by
<B>abstracted</B>.</BODY></HTML>
```

we try to avoid this type of programming (maybe we are just not smart enough to abstract quite this far). However, it can come in handy at times. What is intersting like about CGI.pm is not the abstraction, but the tools it provides. CGI.pm includes a terrific cookie handler and form parameter decoding methods.

We all know forms, right? It's HTML, but with teeth. View the page source to see how we put this one together.

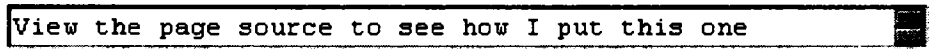


Fig. 8.1

As a Web programmer, forms are the main method of getting information from your users. When a user clicks the Submit button, the browser packs up the information that was typed into that form, encodes it, and ships it off to your waiting CGI (or CGI-ish) program. It is that program's responsibility to un-pack, decode, and use the contents of the form.

This can be a real pain, but it doesn't have to be. Why not let CGI.pm take care of it for you? Form parameters can be extracted using the param method:

```
#!/usr/bin/perl
#
# BAR.cgi is a simple Perl CGI program using CGI.pm.
#
# A Web page with a form containing the FOO textarea
# is meant to submit to BAR.cgi (pay attention to
# the "param" method).
#
use CGI;
$query = new CGI;
print $query->header(),
$query->start_html(-title=>'Test of the param method'),
'The value of the FOO parameter is: ',
$query->param('FOO'),
$query->end_html();
exit 0;
```

This is a CGI.pm way of approaching the problem. we will use CGI.pm later on in some ePerl code, but only to take advantage of that nifty param method.

8.10 Database Escape Sequences

we played around with some albums. One of them was "We Can't Dance," which is not a string that you can blindly insert into a database through the mysql program - MySQL uses the ' character as the string delimiter! It's easy to get around this limitation by "escaping" the ' character with a backslash:

```
mysql> insert into albums(title,artist,released)
@values('We Can\'t Dance','Genesis','1991-01-01');
```

However, it is going to be tougher to do this in a Web-databasing context. Often, what you want to insert into the database will be taken straight from a form parameter. You could set up your ePerl code like:

```
$foo = $query->param('foo');
$foo =~ s/'/\\'/g; # this Perl command will substitute ' with \'
$SQL = <<"EOT";
insert into my_table(my_column)
values ('$foo')
EOT
```

This setup is a very bad idea for a few reasons. MySQL "escapes" characters with a backslash, but other database programs use different escape sequences, so hardcoding a backslash will hurt the portability of your program. Another concern with this approach is that it only tests for single quotes - ' - yet there are other characters that need escaping as well.

Which ones? They vary from database to database.

Don't look up these nitpicky details in documentation and don't try to remember them. You will waste time and open yourself up to human error if you do. Let $dbh->quote do it for you, because it will do it right every time.

```
$foo = $dbh->quote($query->param('foo'));
```

$query->param('foo') will return the value of the form text input foo, and $dbh->quote will escape (aka quote) that value as needed, according to the DBD that corresponds with your database. One last thing that $dbh->quote provides is exterior quotes around the string it acts on, which saves you some typing.

Embedding Referential Expressions in Here-Document Strings

In case you haven't come across the term before, here-document strings are created with the syntax:

```
$string = <<"HERE_DOCUMENT";
```

You can type all sorts of stuff in here....

You can also $interpolate variables right into your h-d string.

The here-document string will quit when it runs into the label given at its outset.

```
HERE_DOCUMENT
```

You have seen how they are used, many times before seen me use these plenty of times before. They're handy, especially for making the $SQL string that is passed off to $dbh to create a client-side $cursor. Take a look at our $SQL creation in the $dbh->quote example above. we had to create an intermediate variable, $foo, to interpolate into the here-document string that actually created $SQL. But that's no good. We can do better.

The problem is that, while $dbh->quote returns a string scalar value, it isn't a string scalar value. It is a function (method, actually), and those can't be shoved willy-nilly into here-documents. For example, the following just won't work:

```
$SQL = <<"EOT";
insert into my_table(my_column)
values ($dbh->quote($query->param('foo')))
EOT
```

What we will do instead is "trick" the here-document with the following incantation. (Look out for the information in bold.)

```
$SQL = <<"EOT";
insert into my_table(my_column)
values (${ \($dbh->quote($query->param('foo'))) })
EOT
```

The \ provides a reference to the value provided by $dbh->quote. The ${ ... } de-references the reference. It makes for some ugly syntax, but hey, Perl was never known for being a pretty language.

8.11 Embedding Subroutines

Although ePerl allows you to embed as much Perl into HTML as you want, sometimes it's nice to put large-ish subroutines at the end of your .iphtml files. It is just cleaner that way. Here's an example:

```
<HTML>
<HEAD><TITLE>Embedding a Subroutine</TITLE></HEAD>
<BODY>

<P>
Here's an HTML calendar for the current month:
<B><?=${ \('cal| head -1') }!></B>

<P>
<?=${ \( calendar_table() ) }!>//

</BODY>
</HTML>
<?

sub calendar_table {

#
# I make copious use of the Unix "cal" command here.
# This won't work on DOS-derivative machines.
#
my @cal = 'cal'; # fill the cal array with the output of a shelled
cal command
my $return = '';
shift @cal; # junk the first line... it's not needed
$return .= "<TABLE BORDER>\n";
$return .= "<TR>";
```

And now, here's that example I mentioned that brings these concepts into play. It consists of two programs, selection.iphtml and receive.iphtml.

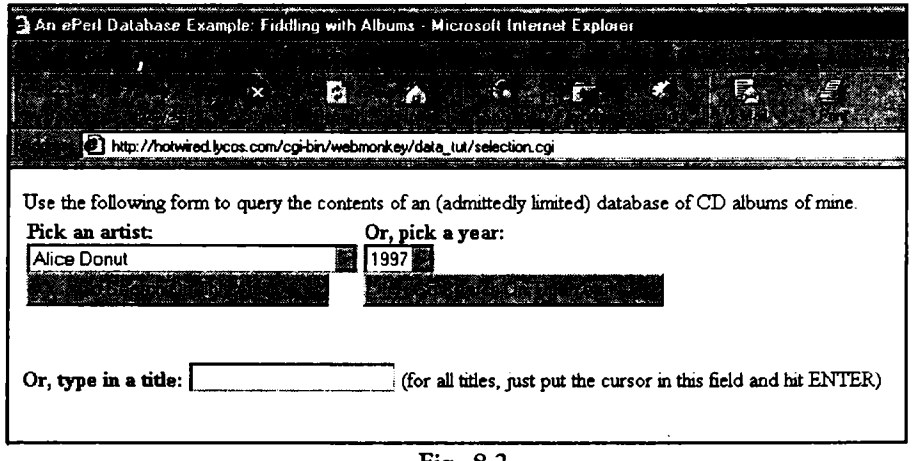


Fig. 8.2

Access selection.iphtml first - it will give you a form, which you might want to play around with. And selection submits to receive.iphtml. Once it gives a listing of the results, a link will take you back to selection.

Unfortunately, we can't demonstrate insert, update, or delete SQL statements here - can't open the database to the world.

Setting Up Forms with SQL Queries - selection.iphtml

```
<?
my $dbh = DBI->connect('DBI:mysql:test:localhost', '','',
{ PrintError => 0}) || die $DBI::errstr;
!>//
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML//EN">
<HTML>
<HEAD>
<TITLE>An ePerl Database Example: Fiddling with Albums</      TITLE>
</HEAD>
<BODY>
<P>
```

Use the following form to query the contents of an (admittedly limited) database of CD albums of mine.

```
 <TABLE>
 <TR>
 <TD><?=${ \( search_by_band ( \$dbh ) ) }!></TD>
 <TD><?=${ \( search_by_year ( \$dbh ) ) }!></TD>
 </TR>
 </TABLE>
 <FORM ACTION=receive.iphtml METHOD=POST>
 <B>Or, type in a title:</B> <INPUT NAME=title SIZE=20>
 (for all titles, just put the cursor in this field and      hit ENTER)
 </FORM>
 </BODY>
 </HTML>
 <?
 $dbh->disconnect;
 !>//
 <?
 sub search_by_band {
 #
 # Note that I passed a reference to the database handle     dbh.
 # This means that, in order to reference it within this
 # subroutine, I'll have to refer to it as $$dbh.
 #
 my $dbh = shift;
 my $return = '';
 #
 # The "distinct" keyword in SQL will only return one row    for a set
of
 # identical matches. "Order by" will sort the returned set
 alphabetically.
 #
```

```
  my $SQL = <<"EOT";
  select distinct artist
  from albums
  order by artist
  EOT
  my $cursor = $$dbh->prepare($SQL);
  $cursor->execute;
  $return .= "<FORM ACTION=receive.iphtml METHOD=POST>\n";
  $return .= "<B>Pick an artist:</B><BR>\n";
  $return .= "<SELECT NAME=artist>\n";
  my @fields;
  while ( @fields = $cursor->fetchrow ) {
  $return .= "<OPTION>$fields[0]\n";
  }
  $return .= "</SELECT><BR>\n";
  $return .= "<INPUT TYPE=SUBMIT NAME=artist_submit VALUE=\"Go  Search on
This Artist!\">\n";
  $return .= "</FORM>\n";
  }
  sub search_by_year {
  my $dbh = shift;
  my $return = '';
  #
  # If COLUMN is defined as a date datum, then year(COLUMN)  will return
only
  # the year portion of the data in the column. "Order by   COLUMN
desc"
  # will reverse the usual sort order.
  #
  my $SQL = <<"EOT";
  select distinct year(released)
  from albums
  order by released desc
  EOT
  my $cursor = $$dbh->prepare($SQL);
  $cursor->execute;
  $return .= "<FORM ACTION=receive.iphtml METHOD=POST>\n";
  $return .= "<B>Or, pick a year:</B><BR>\n";
  $return .= "<SELECT NAME=year>\n";
  my @fields;
  while ( @fields = $cursor->fetchrow ) {
  $return .= "<OPTION>$fields[0]\n";
  }
  $cursor->finish;
  $return .= "</SELECT><BR>\n";
  $return .= "<INPUT TYPE=SUBMIT NAME=year_submit VALUE=\"Go    Search on
This Year!\">\n";
  $return .= "</FORM>\n";
  }
  !>//
```

Processing Form Results with CGI.pm - receive.iphtml

```
<?
my $cgi = new CGI; # to take advantage of the "param"     decoding
method
my $dbh = DBI->connect('DBI:mysql:test:localhost', '','',
{ PrintError => 0}) || die $DBI::errstr;
!>//
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML//EN">
<HTML>
<HEAD>
<TITLE>Results from the Database Search</TITLE>
</HEAD>
<BODY>
<P>
<?=${\( display_query_results(\$dbh, \$cgi) )}!>//
<P>
<A HREF=selection.iphtml>Return to the query page</A>
</BODY>
</HTML>
<?
$dbh->disconnect;
!>//
<?
sub display_query_results {
#
# Note that we passed references to the database handle dbh and the
# cgi object - this means that in order to reference them in this
# subroutine, I'll have to refer to them with $$dbh and $$cgi.
#
my $dbh = shift;
my $cgi = shift;
my $return = '';
my $SQL;
if ( defined($$cgi->param('title')) ) {
my $SQL = <<"EOT";
select title, artist, year(released)
from albums
where ucase(title) like ${ \($$dbh->quote(uc($$cgi-
>param('title')) . '%') ) }
order by title, artist
EOT
#
# I use more complicated SQL in the above statement. SQL
won't
# automatically order the returned results, but it's very
easy to do so with the
# "order by" statement - just specify the columns you want
to appear
# and their order of priority. "Like" allows me to match on
substrings -
# if you provide the title "abc," then all albums titles
beginning with "abc"
```

```
# will be returned. In order to make this comparison case-
insensitive, I
# use ucase(title) in SQL, and uc($$cgi->param('title'), to
put both strings
# into the upper case. The % character is a wildcard, much
like * in Unix
# file name globbing.
#
my $cursor = $$dbh->prepare($SQL);
$cursor->execute;
$return .= "<TABLE BORDER>\n<TR><TH COLSPAN=3>";
$return .= "<B>Matches on the title search for:
<TT><I>${ \($$cgi->param('title') )}</I></TT></B></TH></TR>";
$return .=
"<TR><TH>Title</TH><TH>Artist</TH><TH>Year of Release</TH></TR>\n";
 my @fields;
 while ( @fields = $cursor->fetchrow ) {
 $return .=
"<TR><TD>$fields[0]</TD><TD>$fields[1]</TD><TD>$fields[2]</      TD></TR>\n";
 }
 $cursor->finish;
 $return .= "</TABLE>\n";
 } else {
 if ( defined($$cgi->param('artist_submit')) ) {
 $SQL = <<"EOT";
 select title, year(released)
 from albums
 where artist = ${ \($$dbh->quote($$cgi->param('artist')))      }
 order by released desc, title
 EOT
 } elsif ( defined($$cgi->param('year_submit')) ) {
 $SQL = <<"EOT";
 select artist, title
 from albums
 where year(released) = ${ \($$dbh->quote($$cgi-
 >param('year'))) }
 order by artist, title
 EOT
 }
 my $cursor = $$dbh->prepare($SQL);
 $cursor->execute;
 $return .= "<TABLE BORDER>\n<TR><TH COLSPAN=2>";
 $return .= (defined($$cgi->param('artist_submit'))?
 ("<B>Artist: <TT><I>".
 "${ \($$cgi->param('artist')) }</I>".
"</TT></B></TH></TR>\n<TR>".
 "<TH>Album Title</TH>".
 "<TH>Year of Release</TH></TR>\n"):
 ("<B>Year of Release: <TT><I>".
 "${ \($$cgi->param('year')) }</I>".
"</TT></B></TH></TR>\n<TR>".
"<TH>Artist</TH><TH>Album Title</TH>".
 "</TR>\n"));
```

```
    my @fields;
    while ( @fields = $cursor->fetchrow ) {
    $return .= "<TR><TD>$fields[0]</TD>";
    $return .= "<TD>$fields[1]</TD></TR>\n";
    }
    $cursor->finish;
    $return .= "</TABLE>\n";
    }
    $return;
} !>//
```

8.12 Selecting a Client/Server Application Development Tool

Selecting a client/server development tool is one of the most challenging tasks in the world of client/server. The number and types of tools change so quickly that it's difficult to keep up. Each tool differs greatly in features and capabilities, and it's often difficult to rely on marketing literature for the information you need to make the right selection.

To make matters worse, unlike other enabling client/server technology, the penalty for selecting the wrong tools is severe. Many a failed client/server development project can trace critical problems back to bad tools. What's more, the wrong tool for the job can double the cost of a project, or result in lost end-user productivity.

8.12.1 Considering Application Requirements

First, it is helpful to understand how system requirements fit in. The trick to selecting the right client/server development tool for the job is to go from the application requirements to the tool, and never from the tool to the requirements. This means that you must understand the application requirements completely, and compose a list of features and functions that the tool must possess to meet those requirements or the criteria.

For example, an application may require a connection to an obscure legacy database, as well as a connection to a messaging system and the ability to utilize bar codes. Those requirements (and many others,) become the criteria to select a development tool.

Because most applications have some unique requirements, there is no single set of criteria that you can use to select a client/server development tool. However, there are some common requirements such as repositories, object-oriented development, database connectivity, component integration, cross-platform support, and three-tier and *n*-tier support.

8.12.2 Repositories

Repositories in development tools let you layer things such as validation rules, fonts, colors, and relationships on top of the physical database schema. You build an application on top of a repository, which automatically inherits the characteristics of the metadata. The idea is to set up all this information once and reuse it throughout the application.

Using repositories, maintenance activities become easier because you can perform global changes by changing a single item in a repository (for example, changing the attributes of a column). A good tool propagates the changes throughout an application through the object-oriented concept of inheritance. You can usually override the repository when required.

Some repositories also store application objects along with other data-related repository information. You can reuse these application objects from the repositories, and sometimes repositories are shareable among developers in a project team environment.

Most specialized client/server tools such as Powersoft's PowerBuilder, Symantec's Enterprise Developer, JYACC's JAM 7, and Compuware Corp.'s Uniface provide repositories. However, each tool implements its repository in its own special way. For example, PowerBuilder's repository, known as the Extended Attribute Set, is stored in the database along with the data. This lets you build on the simple schema information in the database. PowerBuilder lets you specify extended attributes for each column, which lets you attach application-oriented information (for example, display formats, validation rules, initial values, and header or label text) to columns for use throughout the PowerBuilder application. Thus, you can specify application standards.

8.12.3 Database Design Facility

You have to build the database sometime, and you might as well have the power to do it right from your client/server development tool. A database design facility lets developers and DBAs create entire databases directly from the same tool in which they build the database application. You have an easy-to-use design facility, as well as the capability to generate the native DDL.

Most database design facilities, such as those found in PowerBuilder and Symantec Enterprise Developer, provide graphical diagramming capabilities that resemble CASE tools. These tools let you create icons on the screen that represent tables in the database. You can set the relationships when you link the icons together by common keys, usually using a drag-and-drop facility. Once complete, the tool can generate the DDL automatically and create the objects in the physical database. Some tools let you roll back database changes.

Some tools support database design through crude SQL interfaces, which require you to write the proper SQL code, and some tools require you to use external facilities such as CASE tools to create and alter a database. These are less desirable characteristics.

8.12.4 Database Connectivity

Tools should support most popular databases, including Oracle, DB/2, Informix, and Sybase, using native 32-bit drivers, and should offer support for many other databases through ODBC. Database connectivity should be built directly into the tools, so you don't have to integrate third-party middleware. Some tools let you tune the application for a particular database. For example, setting cache size, buffer size, bucket size, and so forth can help meet the particular needs of the application using the database server.

In addition, the tool should be decoupled from the database server. In other words, you should be able to swap out databases quickly, making only minor modifications to the application. Tools should also support heterogeneous database connectivity, or the ability to connect to more than one brand of database server at a given time, or multiple servers running the same DBMS.

8.12.5 Application Design Facility

A less common tool feature is an application design facility that lets you model the processes. Generally speaking, most development tools leave this to CASE tools and methodologies (for example, Booch, Coad/Yourdon, OMT, and so on). However, most specialized client/server development tools provide simple mechanisms to do things such as browse an application's class hierarchy. This lets you view how each object interacts with the others and move objects around to structure the application better.

Application partitioning tools such as Forte and Dynasty provide high-level application design facilities, letting you define application objects in a single environment and then partition them to available application servers accessible by a network. Once the objects exist, you can move them to application servers for realtime processing by dragging and dropping them inside the application partitioning facility. Although this is a functional process, it is an example of an application design facility built into a development tool.

8.12.6 Correct Use of Objects

Most client/server development tools use an object-oriented development model. However, they do so in different ways. Object- oriented development, simply put, means that a development tool or language supports inheritance, encapsulation, and polymorphism.

Using an object-oriented model, a tool can separate an application into classes and objects (instances of a class), or self-contained modules that encapsulate both the data and the methods that interact with the data. Inheritance lets developers program from the generic to the specific, reusing as much code as possible. If designed correctly, object-oriented systems are easy to build and maintain.

Some tools support multiple inheritance, or the ability to inherit data and methods from two classes, combining them in a single class. Although this is sometimes nice to have, it is a dangerous practice. Combining methods and data can cause conflicts, and they are difficult to diagnose in an object-oriented environment because the problems actually exist in higher-level classes. Therefore, use multiple inheritance at your own risk.

8.12.7 Programming Language

Programming languages give you the ability to make the application manipulate data and set or alter properties of controls, forms, and dialogs, as well as access disk files, invoke the operating system's API, and many other low-level processing tasks. Keep in mind that the tool should emphasize graphical design and minimize the amount of hand-coding necessary.

Most client/server tools provide fourth generation languages (4GLs), or high-level languages (sometimes called scripts). These are often proprietary languages.

The programming language needs to provide typical operators such as if-then-else and looping logic. Programming languages should also let you manipulate the database directly. Moreover, the programming language should provide object-oriented features, meaning support for inheritance, encapsulation, and polymorphism, not procedural programming.

8.12.8 Application Deployment

Client/server development tools need to have an efficient application deployment mechanism, or the ability to distribute runtime versions of the application to end users. This should be royalty-free. Most client/server tools compile applications for distribution, but the differences lie in the compiled code.

Typically, client/server development tools provide p-code interpreters. This means that the tool creates a file that musst be interpreted at runtime into machine language for the processor. This provides an easier development environment because the code does not go directly to the processor as a native executable, but performance can suffer. A few client/server tools (such as Delphi), as well as most 3GLs, provide true compilers that create native executables, which are preferable.

8.12.9 Performance

No matter what deployment mechanism the tool uses, application performance varies greatly from tool to tool. Performance is critical because it is how the user perceives the application. Performance problems are difficult to solve with faster processors because it's inevitable that lower-powered clients will run your application.

There are two performance considerations: application performance and database performance. Client/server tools should provide consistent application performance when painting screens, displaying data, and jumping quickly from screen to screen. Doubling the number of objects in an application should not greatly affect the performance of an application.

Tools should access databases quickly, moving the data efficiently from the physical database over the network to local memory. Although much of this is dependent on the capabilities of the database server, application performance should not change when the size of the database increases. Measuring performance should not be a subjective process (for example, "seems fast"). To obtain valid results, you should compare tools in the same category, using similar configurations (for example, database, middleware, and so on).

8.12.10 Third-Party Component Integration

Today, the key to quick development is the ability to build as much of an application as possible from prebuilt component parts. Client/server development tools should support standard components such as VBXs and the newer ActiveX (formerly called OCX) controls. Tools that don't support components lock you out of a world that could potentially save you a lot of money — as much as 50 percent of the coding time, depending on the project.

Components are added in as objects to applications to solve particular problems, such as the need for a built-in calculator, access to middleware layers, or high-level financial calculations. An application can be made up of many components that were either developed in-house, or better yet, purchased from third-party vendors.

8.12.11 Cross-Platform Support

Cross-platform support promises that the client/server tool will support more than one operating system and GUI. This means that you can write an application one time, then port it without

modification to other platforms. Some tools allow porting to similar platforms such as Windows 3.11, Windows 95, and Windows NT (we call these second-tier tools). Others support very dissimilar platforms, including Unix, Macintosh, and OS/2 (we call these first-tier cross-platform tools).

There is a tradeoff to cross-platform support. When a tool supports many different platforms, it has a tendency to do no single platform well. Applications running on many of the supported platforms lack a native look and feel or have performance problems. Therefore, it's a good idea to use cross- platform tools only if you really need them.

When considering cross-platform tools, you need to consider development platforms vs. deployment platforms. Some cross- platform tools only let you build an application on a particular platform (for example, Windows 3.11). Once an application exists, you can deploy it to other platforms that the cross-platform tool supports. However, changes to and debugging of an application must occur on the originating platform.

8.12.12 Room to Grow

Scalability refers to a tool's ability to support a growing number of clients. Many specialized client/server tools don't scale well. Applications built with the tool should handle additional clients without a significant impact on performance or stability. Scalability is also a function of how efficiently the client uses the database server.

Products that scale well include tools that support application partitioning through the use of proprietary ORBs (for example, Dynasty and Forte), remote OLE automation or Distributed COM (PowerBuilder, Delphi, and Visual Basic), TP monitors (EncinaBuilder from Powersoft and JAM/TPI from JYACC), or CORBA- compliant distributed objects (usually accessible using DLLs). These tools can off-load some of the application processing to a standalone application server. In addition, an application server is able to funnel database requests, thus placing a lighter per- user load on the database server.

CHAPTER 9

DISTRIBUTED DATABASE

9.1 Overview of Disributed Database

Information technology (IT) systems are experiencing rapid growth in numbers of users supported and system complexity. The IT community must handle the requirements of mission-critical applications, capacity growth rates that exceed 50 percent annually, excessive downtime, and increasing business dependence on computer systems. To add a further layer of complexity, systems management is now an issue-how can a system administrator or network manager monitor and control all of the computing resources? Without centralized management tools, he or she cannot effectively manage this data.

The quantity of data being stored on distributed systems has increased exponentially over the last decade. From 1996 to 1998, Windows NT-based server data has grown from 11 to 39 petabytes worldwide. This data explosion shows no signs of slowing down; typical server capacities that exceed 100 gigabytes are not far in the future. By 2002, data stored on the Windows Server operating system is projected to exceed 260 petabytes worldwide. Much of the data stored on Windows-based servers is business- critical. In a recent Strategic Research study of over 200 sites, 31 percent of the servers were running Windows NT version 4.0 or earlier to host mission-critical databases.

The migration of mission-critical systems to distributed environments, increases in the number of Web-based applications, and general growth in the enterprise end-user community all contribute to this rapid growth. As the number of client/server systems increases in an organization, so does the number of storage subsystems. Unfortunately, tools for remote management and management standards have only recently become mainstream in the distributed systems market-place.

The type of data being stored on client/server systems is changing as well. Growth in the Internet/intranet space and 32- bit and 64-bit architectures gives impetus to the changes in data seen in the distributed network. First, the increase in popularity in Web-based applications has resulted in an increase of multimedia data types (including large video streams) that have significant storage requirements. Moreover, the ease of use and low cost of ownership of Web-based applications are sparking a trend toward publishing data that is content-oriented, rather than application-oriented, as was previously the case. Content- style data ranges from static (for example, publishing a book online) to dynamic (such as publishing a daily newspaper). Finally, 32-bit and 64-bit platform support results in the development of more powerful, graphically intense applications, which in turn create large volumes of data that present significant storage management challenges.

As the popularity of the client/server infrastructure increases, the cost and complexity of managing distributed storage increases as well. The LAN environment has seen a 60-percent yearly growth rate of storage management expenditures since 1993.

Storage management and storage recovery is a concern for many IT managers. While most recent IT magazines and articles address concerns with desktop application management and software distribution, a large portion of enterprise computing budgets is spent on storage issues. As much as 25 percent of a typical computing budget is dedicated to storage, storage management, and other storage-related activity.

Storage limitations can also constrain other areas in enterprise computing. For example, the ultimate scalability of application implementations is often limited by the effectiveness of the storage and storage-recovery mechanisms in place. Without centralized control and management of distributed systems, applications experience excessive downtime, much of which is caused by storage-related failures. Server outages and inconsistent access to data directly affect productivity. Through interviews with hundreds of IT managers, Strategic Research has determined that centralized sites (those using central control and management tools to manage distributed and host storage) typically experience less than half of the downtime of sites without centralized control and management. A representative centralized site experiences 26 hours of downtime annually, compared to a decentralized site that has 54 annual hours of downtime. Conservative estimates place downtime costs at $80,000 per hour, which means that centralized sites save over $2.2 million dollars in downtime-related costs each year.

The cost of managing a megabyte of storage on distributed systems has decreased from $8 per megabyte in 1994 to $3 per megabyte in 1998, largely as a result of storage management tools that support centralized storage management in distributed systems.

Innovations are emerging in the storage industry to meet the growing demands of client/server computing. These innovations include new storage devices, media types, data transfer protocols, and management standards. Storage concepts such as hierarchical storage management (HSM), bulk media changers/libraries, data vaulting managers, and fault-tolerant storage subsystems are being introduced by a variety of vendors. Storage requirements will continue to increase because the trends described above are just the beginning of a new wave in the need for increased storage and storage management. As enterprise storage becomes more complex, it

becomes essential for IT managers to be able to effectively manage it in order to be successful in meeting their short-term and long-term computing goals.

9.1 Another Basis of Classifying Databases

Two common methodologies for database management are centralized data processing and client/server processing. With centralized processing, one computer controls all data functions, including storage, processing, applications, and programming. In a client/server processing environment, the server controls the database and its storage, while clients process applications to access the server base and retrieve information.

Databases are classified based on physical configuration:

- Centarilised
- Clinet server
- Distributed

9.1.1 Centralised Control

The concept of centralised control implies that there will be some identifiable person or persons who have a central responsibility for the operational data in the DBMS. That person is the Database Administrator (DBA).

The following are some advantages that accrue from this notion of centralised control:

- Redundancy eliminated
- Inconsistency avoided
- Data shared
- Standards enforced
- Security applied
- Integrity maintained
- Requiremants balanced
- Centralized Databases

A centralized database (e.g., hospital information system) can be very big and complex, but offers the following advantages:

- Shared data, reduced redundancy
- Fewer inconsistencies in data
- Enforcement of standards
- Security restrictions
- Balancing of conflicting requirements

9.1.2 Distributed DBMS

The DB is stored across many computers (servers or hosts) data is combined from several servers many computers may help in the processing of one query machines may be both a client and a server.

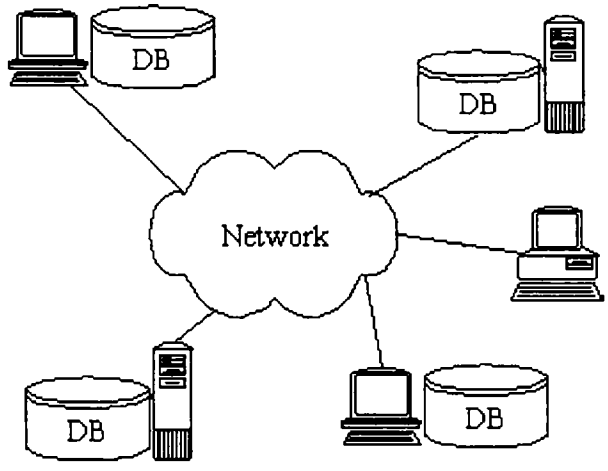


Fig. 9.1

Each type of processing has its own advantages and disadvantages; therefore, the tradeoffs must be considered. While both methods allow sharing of data by users and have communications capabilities, client/server processing reduces communications costs and distributes the processing burden between the main server machine and the supported clients, providing more power on both ends. On the other hand, centralized data processing offers more database control. With processing in one location, it is easier to enforce system and procedural standards, to maintain security and integrity, and to balance conflicting processing requirements.

9.1.3 Client/Server Databases

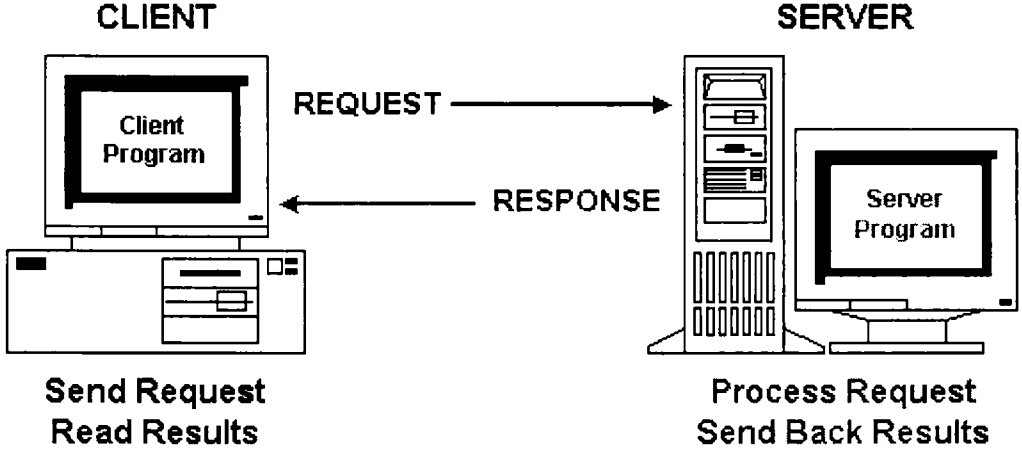


Fig 9.1

As we discussed earlier, most databases that you will come across these days will be relational databases. However, there are many types of relational databases and not all of them will be useful for web applications.

In particular, it will be the client/server databases rather than the stand-alone packages that you will use for the web.

A client/server database works like this: A database server is left running 24 hours a day, and 7 days a week. Thus, the server can handle database requests at any hour. Database requests come in from "clients" who access the database through its command line interface or by connecting to a database socket. Requests are handled as they come in and multiple requests can be handled at one time.

For web applications which must be available for world wide time zone usage, it is essential to build upon a client/server database which can run all the time.

For the most part, these are the only types of databases that Internet Service Providers will even provide. However, if you are serving web pages yourself, you should consider many of the excellent freeware, shareware or commercial products around. You might like postgres for UNIX since we prefer a UNIX-based web server. However, there are plenty of good applications for PC and Mac as well.

- DBMS is housed within the server
- Clients access the server via some network
- Application programs are stored on the client machines

9.2 Distributed Database

A distributed database can be defined as consisting of a collection of data with different parts under the control of separate DBMSs running on independent computer systems. All the computers are interconnected and each system has autonomous processing capability serving local applications. Each system participates, as well, in the execution of one or more global applications. Such applications require data from more than one site. The distributed nature of the database is hidden from users and this transperency manifests itself in a number of ways.

Although there are a number of advantages to using a distributed DBMS, there are also a number of problems and implementation issues. Finally, data in a distributed DBMS can be partitioned or replicated or both.

The DB is stored across many computers (servers or hosts) data is combined from several servers many computers may help in the processing of one query machines may be both a client and a server

9.2.1 Major Features of a DDB

The major features of a DDB are the following:

- Data stored at a number of sites, each site logically single processor
- Sites are interconnected by a network rather than a multiprocessor configuration
- DDB is logically a single database (although each site is a database site)

- DDBMS has full functionality of a DBMS

To the user, the distributed database system should appear exactly like a non-distributed database system.

9.2.2 Advantages of Distributed Database

Advantages of distributed database systems are as follows:

- Improved reliability/availability
- Capacity and growth
- Distributed database sharing
- Efficiency and Flexibility
- Local autonomy (in enterprises that are distributed already)
- Improved performance (since data is stored close to where needed and a query may be split over several sites and executed in parallel)
- Economics

(1) Reliability and Availability

An advantage of distributed databases is that even when a portion of a system (i.e. a local site) is down, the overall system remains available. With replicated data, the failure of one site still allows access to the replicated copy of the data from another site. The remaining sites continue to function. The greater accessibility enhances the reliability of the system.

(2) Capacity and Growth

An advantage of distributed databases is that as the organisation grows, new sites can be added with little or no upheaval to the DBMS. Compare this to the situation in a centralised system, where growth entails upgrading with changes in hardware and software that effect the entire database.

(3) Distributed Database Sharing

An advantage of distributed databases is that users at a given site are able to access data stored at other sites and at the same time retain control over the data at their own site.

(4) Efficiency and Flexibility

An advantage of distributed databases is that data is physically stored close to the anticipated point of use. Hence if the usage patterns change then data can be dynamically moved or replicated to where it is most needed.

9.3 Disadvantages of Distributed Database Systems

Disadvantages of distributed database systems are as follows:

- Complexity (greater potential for bugs in software)
- Cost (software development can be much more complex and therefore costly. Also, exchange of messages and additional computations involve increased overheads)
- Distribution of control (no single database administrator controls the DDB)
- Security (since the system is distributed the chances of security lapses are greater)

- Difficult to change (since all sites have control of the their own sites)
- Lack of experience (enough experience is not available in developing distributed systems)

9.4 Distributed Database Problems

The disadvantages of the distributed approach to DBMS implementation are its cost and complexity. A distributed system, which hides its distributed nature from the end user, is more complex than the centralised system. Increased complexity means that the aquisition and maintainence costs of the system are higher than those for a centralised DBMS. The parallel nature of the system means that errors are harder to avoid and those in the applications are difficult to pinpoint. In addition, the distributed system, by its very nature, entails a large communication overhead in coordinating messages between the different sites.

9.5 Distributed Database Issues

There are a number of issues or problems which are peculiar to a distributed database and these require novel solutions. These include the following:

- Distributed query optimisation
- Distributed update propagation
- Distributed concurrency control
- Distributed catalog management

9.6 Global Query Optimization

Global query optimization is complex because of:

- Cost models
- Fragmentation and replication
- Large solution space from which to choose
- Computing cost itself can be complex since the cost is a weighted combination of the I/O, CPU and communications costs. Often one of the two cost models are used; one may wish to minimize the total cost (time) or the response time. Fragmentation and replication add another complexity to finding an optimum query plan.

9.7 Distributed Update Propagation

Update propagation in a distributed database is problematic because of the fact that there may be more than one copy of a piece of data because of replication, and data may be split up because of partitioning. Any updates to data performed by any user must be propagated to all copies throughout the database. The use of snapshots is one technique for implementing this.

9.8 Concurrency Overview

Most DBMS's are multiple users systems; that is, they are systems that allow any number of transactions to access the same database at the same time. In such a system some kind of concurrency control mechanism is needed to ensure that concurrent transactions do not

interfere with each other's operation. There are essentially three ways in which things can go wrong - three ways in which a transaction, though correct in itself, can nevertheless produce the wrong answer because of interference on the part of some other transaction. The interfering transaction may also be correct in itself as it is the interleaving of operations from the two correct transactions that produces the overall incorrect result. The three problems are:

- The lost update problem.
- The uncommitted dependency problem.
- The inconsistent analysis problem.

The concurrency control techniques most commonly used are:

- Locking.
- Timestamping.

9.9 Distributed Concurrency Control

Concurrency control in distributed databases can be done in several ways. Locking and timestamping are two techniques which can be used, but timestamping is generally preferred.

The problems of concurrency control in a distributed DBMS are more severe than in a centralised DBMS because of the fact that data may be replicated and partitioned. If a user wants unique access to a piece of data, for example to perform an update or a read, the DBMS must be able to guarantee unique access to that data, which is difficult if there are copies throughout the sites in the distributed database.

9.10 Distributed Catalog Management

The distributed database catalog entries must specify site(s) at which data is being stored in addition to data in a system catalog in a centralised DBMS. Because of data partitioning and replication, this extra information is needed. There are a number of approaches to implementing a distributed database catalog. There are as follows:

- Centralised - Keep one master copy of the catalog;
- Fully replicated - Keep one copy of the catalog at each site;
- Partitioned - Partition and replicate the catalog as usage patterns demand; and
- Centralised/partitioned - Combination of the above.

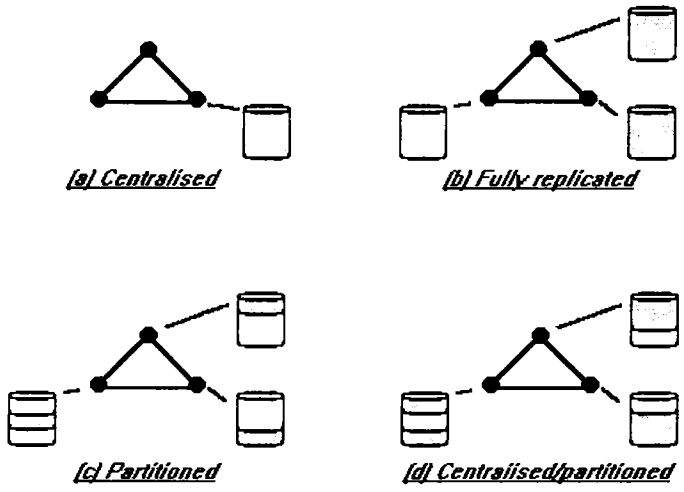


Fig 9.4

9.11 Snapshots

A snapshot, like a view, is a derived relation (however, snapshot are physically stored in the database), and are used for implementing update propogation in distributed databases. Snapshots allow freezing without preventing transactions which are suitable for applications which tolerate earlier versions of data. Supports one master copy and x snapshots. Note that snapshots are read only.

9.12 Transparency and Autonomy

Transparency involves the user not having to know how a relation is stored in the DDB; it is the system capability to hide the details of data distribution from the user.

Autonomy is the degree to which a designer or administrator of one site may be independent of the remainder of the distributed system.

It is clearly undesirable for the users to have to know which fragment of the relation they require to process the query that they are posing. Similarly, the users should not need to know which copy of a replicated relation or fragment they need to use. It should be upto the system to figure out which fragment or fragments of a relation a query requires and which copy of a fragment the system will use to process the query. This is called replication and fragmentation transparency.

A user should also not need to know where the data is located and should be able to refer to a relation by name which could then be translated by the system into full name that includes the location of the relation. This is location transparency.

9.13 Data Replication

In a distributed DBMS a relational table or a partition may be replicated or copied, and copies may be distributed throughout the database. This feature can cause problems for propagating updates and concurrency control and this is transparent to users in distributed databases.

9.14 Replication Transparency

By replication transparency in a distributed database we mean that partitioned/replicated updates must be propagated through to all copies in existence. Replication is desirable and transparent.

9.15 Distributed Database Transparency

A distributed DBMS should provide a number of features which make the distributed nature of the DBMS transparent to the user. These include the following:

- Location transparency
- Replication transparency
- Performance transparency
- Transaction transparency
- Catalog transparency

The distributed database should look like a centralised system to the users. Problems of the distributed database are at the internal level.

9.16 Location Transparency

By location transparency in a distributed DBMS we mean that users/programs should not need to know where data is stored. This is a system catalog concern only. This simplifies application program logic and allows data movement within the distributed database as usage patterns emerge.

9.17 Replication or Fragmentation of Data

The unit of distribution may be a relation or a fragment; often fragment is a more suitable unit since it allows parallel processing of a query.

Replication improves availability since the system would continue to be fully functional even if a site goes down. Replication also allows increased parallelism since several sites could be operating on the same relations at the same time. Replication does result in increased overheads on update.

Fragmentation may be horizontal, vertical or hybrid (or mixed). Horizontal fragmentation splits a relation by assigning each tuple of the relation to a fragment of the relation. Often horizontal fragmentation is based on predicates defined on that relation. Vertical fragmentation splits the relation by decomposing a relation into several subsets of the attributes. Relation R produces fragments each of which contains a subset of attributes of R as well as the primary key of R. Aim of vertical fragmentation is to put together those attributes that are accessed together.

Mixed fragmentation uses both vertical and horizontal fragmentation. To obtain a sensible fragmentation design, it is necessary to know some information about the database as well as about applications. It is useful to know the predicates used in the application queries – at least the 'important' ones.

Aim is to have applications using only one fragment.

Fragmentation must provide completeness (all information in a relation must be available in the fragments), reconstruction (the original relation should be able to be reconstructed from the fragments) and disjointedness (no information should be stored twice unless absolutely essential, for example, the key needs to be duplicated in vertical fragmentation).

9.18 Data Partitioning

In a distributed DBMS a relational table may be broken up into two or more non-overlapping partitions or slices. A table may be broken up horizontally, vertically, or a combination of both. Paritions may in turn be replicated. This feature causes problems for concurrency control and catalog management in distributed databases. This is transparent to users.

9.19 Client/Server Software Architectures—An Overview

9.19.1 Purpose and Origin

The term client/server was first used in the 1980s in reference to personal computers (PCs) on a network. The actual client/server model started gaining acceptance in the late 1980s. The client/server software architecture is a versatile, message- based and modular infrastructure that is intended to improve usability, flexibility, interoperability, and scalability as compared to centralized, mainframe, time sharing computing.

A client is defined as a requester of services and a server is defined as the provider of services. A single machine can be both a client and a server depending on the software configuration. For details on client/server software architectures.

9.19.2 Why Client/Server ?

The business environment of the 1990's is rapidly changing. Corporate organizations are flattening their organization structures to be more responsive to customers and the marketplace. Traditional competitors are now cooperating with one another while competing at the same time in different markets. Key competitive advantages are characteristics like time-to- market, customer service and simplicity. Being first to a market with a quality product or service which delights customers can be the difference between market success or failure. Time to market is now measured in weeks not years, and being first may be the only advantage one company has over its competitors. Customer service means delighting the customer with both a product or service as well as being responsive to changing customer requirements. Organizational structures are flattening in order to empower employees to make decisions closer to the customer, to be more responsive to the customer. Personal computers provide a whole new level of computing flexibility, with simple easy to use programs. These programs are transportable, allowing businesses to react instantly to customers needs. Simplicity, to the end user, is critical to the success of client/server

computing. End users need information access as easy as getting cash from an automated teller machine (ATM). Customers can get money from their bank account at any ATM across the country with just an ATM card and a password. Client/server computing means changing our information systems to support the changes of the business environment. Lets look at an analogy.

9.19.3 Client/Server Analogy

Restaurant service is an analogy to help explain client/server computing. The customer (client) makes a series of requests for a specific set of services that may include an appetizer, beverage, main course and a dessert. These requests are all typically made to one person, the waiter (server). The services may actually be provided by a number of other people in the restaurant including the bus boy, bartender, and a variety of chefs. However, to the customer, these services are all provided by one person, the waiter. The customer doesn't want to know who performs what service. He would just like to have a high quality meal delivered in a timely fashion. The client, in client/server computing is much like the customer in a restaurant. The client requests a service, like running an application or accessing some information from a data base. The server becomes responsible for performing the service and returning the information to the client in a timely manner. The server is like the waiter in a restaurant responsible for handling the client's requests and delivering the finished product to the client.

9.19.4 Client/Server Definition

Client/server computing provides the seamless integration of personal computers with host systems. This style of computing allows organizations to be responsive to their customers while still maintaining the security and integrity to manage their business effectively. Client/server computing generally refers to a computing model where two or more computers interact in such a way that one provides services to the other. This model allows customers to access information resources and services located anywhere within the customers information network. Customers are very interested in client/server computing because it allows them to be more responsive, as well as to effectively utilize all computing resources within their network.

As the term implies, client/server computing has two basic components, a client and a server. The client requests a service to be performed. This service might be to run an application, query a data base, print a document, or even perform a backup or recovery procedure. The server is the resource that handles the client's request. Clients are typically thought of as personal computers but a client can be a midrange system or even a mainframe. Servers are typically thought of as a midrange or mainframe system, however a server can be another personal computer on the network. Client/server networks are like our restaurant example where specific computers provide one or more services to other computers within a network. Today's networks have computers for file serving, data base serving, application serving, and communications serving. Each of these servers are dedicated devices which provide a specific service to all authorized users within a network. These servers also allow some of the processing to be handled on each users PC and some on a centralized server.

For example, a data base server uses the PC for the display (user interface) and processing (application logic) portions of an application, while the server provides data management portion of the application. On the other hand, an application server uses the PC for the display portion of an application, while using the server for both the processing and data management portions.

Client/Server systems allow many people to share centralized data. This requires a network of personal computers (PCs) which are the "clients," to be linked to a central computer, known as the "server." Any kind of data (addresses, values, codes, phone numbers and so forth) can be stored in a database on the server. The database is accessed with a software application that runs on client PCs. Sometimes the work of the application is done on the server, and sometimes it's done on the PC - this functionality is more efficient than traditional computer systems because work can be taken from the server, computed locally with a PC, then moved back to the server. This process is not so taxing on the PCs or the server because they share data, leading to more efficient use of computing resources.

Applications used in client/server systems are usually of the Graphical User Interface type, or "GUI" (pronounced "gooey"). This type of application is usually in a Windows(tm) format, so you can either use a mouse to point-and-click through the application or navigate with keystrokes. GUI applications are much easier to use than traditional mainframe-based applications, which leads to increased productivity and quality, as well as a reduced training investment.

Most of the time businesses have very specific types of data in their databases (whether Oracle or Sybase), so the application used to access this data has to be customized. This can be done with many development tools: Java, SilverStream, PowerBuilder, Visual Basic, Delphi, C++ or Visual C++, MS Access, HTML, JavaScript, and WebPB

9.20 Client/server on Internet

Most transactions that occur on the Internet are client/server based. Some examples include:

- *FTP (file transfer protocol)* – An FTP client program contacts an FTP server and requests the transfer of a file; the FTP server responds by transferring the file to the client.
- *WWW (World Wide Web)* – In this case the client program is a browser. A browser requests the contents of a web page and displays the results on the user's computer.
- *E-MAIL* – A mail client program enables the user to interact with a server in order to access, read and send electronic mail messages.

Client/server computing is a common networking model which enables many users to access information in an efficient manner. Generally, the user's computer is called the client and the machine that contains the information being accessed is called the server.

The client computer runs an application called a client program. A client program enables a user to send a request for information to the server and read the results that the server sends back. The server computer runs a server program which processes requests and sends results back to the client. Most Internet transactions, such as FTP, e-mail and accessing web pages are based on client/server networking.

9.21 Client-Server Applications

Client/Server technology is the fastest growing segment of systems development and deployment in the financial services industry.

The viability of the technology has increased to the point that client/server implementations have expanded well beyond the original domain of this technology (e.g., Decision Support Systems (DSS), and departmental, highly specialized transaction processing). Several mission critical, enterprise wide client/server systems have been deployed throughout the financial services industry. Large scale implementations (200+ clients) have grown tremendously, as well.

The increased reliance on client/server technology has resulted in far heavier utilization of local and wide area connectivity. Network systems management, control and security remain the greatest concern in managing the ever increasing distributed computing environment.

Client/server technology is not being utilized in the "downsizing" and "rightsizing" methodology that many industry experts had anticipated it would be. The large majority (94%) of client/server applications being developed in US banks are new applications. Only 6% of client/server development represent downsized applications (e.g., retooling pre-existing host applications).

Client/server spending is the fastest growing segment of US bank IT spending. The client/server spending is approximately about 25.0% CAGR.

This technology description provides a summary of some common client/server architectures and, for completeness, also summarizes mainframe and file sharing architectures. Detailed descriptions for many of the individual architectures are provided elsewhere in the document.

9.21.1 Mainframe Architecture (not a client/server architecture)

With mainframe software architectures all intelligence is within the central host computer. Users interact with the host through a terminal that captures keystrokes and sends that information to the host. Mainframe software architectures are not tied to a hardware platform. User interaction can be done using PCs and UNIX workstations. A limitation of mainframe software architectures is that they do not easily support graphical user interfaces or access to multiple databases from geographically dispersed sites. In the last few years, mainframes have found a new use as a server in distributed client/server architectures.

9.21.2 File Sharing Architecture (not a client/server architecture)

The original PC networks were based on file sharing architectures, where the server downloads files from the shared location to the desktop environment. The requested user job is then run (including logic and data) in the desktop environment. File sharing architectures work if shared usage is low, update contention is low, and the volume of data to be transferred is low. In the 1990s, PC LAN (local area network) computing changed because the capacity of the file sharing was strained as the number of online user grew (it can only satisfy about 12 users simultaneously) and graphical user interfaces (GUIs) became popular (making main-frame and terminal displays appear out of date). PCs are now being used in client/server architectures.

As a result of the limitations of file sharing architectures, the client/server architecture emerged. This approach introduced a database server to replace the file server. Using a relational database management system (DBMS), user queries could be answered directly. The client/server

architecture reduced network traffic by providing a query response rather than total file transfer. It improves multi-user updating through a GUI front end to a shared database. In client/server architectures, Remote Procedure Calls (RPCs) or standard query language (SQL) statements are typically used to communicate between the client and server.

9.22 Client-Server Basics

There are four forms of distributed processes that are widely used in distributed computing: client, server, peer, and filter.

- Clients are usually computational entities that request service resources.
- Servers are usually resource entities that respond to service requests.
- Peers are usually identical to one another and may make or service requests.
- Filters are usually request or response relays that consistently modify what they relay.

Clients are consumers of services and resources. Servers are providers of services and resources. Peers may consume and provide services and resources but are not generally specialized. Filters are sometimes both client and server and frequently act as an intermediary in client-server systems. Specialization tends to reduce the amount of common code that appears in multiple places. In other words, such specialization has a tendency to keep application code in a single place and not spread it around an enterprise.

Task division between clients and servers is a difficult problem. As in any distributed application the sharing of services and their availability must be considered. For instance, in the traditional C-S we have a GUI on the client and a database on the server. In such a case, the computation is usually best moved onto the client. This is a typical two-tier client-server environment.

9.22.1 Client-Server Architecture

Client-server architecture is a particular type of distributed computing networks which is very popular and among he most widely used today for business computing networks. Client-server architecture concerns how processing activity is distributed over the network, regardless of the topological arrangement of the network's components. A client node in a network is the computer that requests and receives services from there computers on the network.

It is possible for larger and more powerful computers to ometimes play the role of "client" in requesting processing services from other computers. In a distributed computing environment, any client computer that has hardware and software capability to share can also act as a server. Typically, clients in a local area network are user PC workstations. Such computers, are called end-user clients, never play the role of server. ervers are usually "high-end" microcomputers that are especially quipped to handle special processing and communication tasks. They are provided with special server application software and he hardware resources to run the software efficiently.

Additionally, the server is normally provided with special network connections that provide rapid efficient delivery of data to the clients.

Under client-server architecture, processing work is distributed between the client and the server; both are required to meet the entire need of the application. Client software running on the client platform makes requests to servers, which are usually dedicated to that function alone.

9.22.2 File Server

These simply provide entire data files to the client. One special ype, used to provide files over the public internet, is known as an ftp server database server —these extract data from a database according to the request of the client communication erver —provide gateway and other related communication services servers—permit the queuing and management of printing services mail server —hold user e-mail messages and forward them two particular types of mail servers found on the internet are OP servers (receive and hold email for clients) SMTP servers(accepts email from clients and forward it over the network) print server — provides "print spooling" services to help hold print jobs scheduled for shared printers until the printer is free.

9.22.3 Web Server

Provides web pages for client browsers using the "HTTP" protocol in organization may have many servers; for example, Penn State niversity has hundreds, including some whose current status is available for you to see on the Internet.

9.22.4 Two Tier Architectures

With two tier client/server architectures, the user system interface is usually located in the user's desktop environment and the database management services are usually in a server that is a more powerful machine that services many clients. Processing management is split between the user system interface environment and the database management server environment. The database management server provides stored procedures and triggers. There are a number of software vendors that provide tools to simplify development of applications for the two-tier client/ server architecture.

The two-tier client/server architecture is a good solution for distributed computing when work groups are defined as a dozen to 100 people interacting on a LAN simultaneously. It does have a number of limitations. When the number of users exceeds 100, performance begins to deteriorate. This limitation is a result of the server maintaining a connection via "keep-alive" messages with each client, even when no work is being done. A second limitation of the two tier architecture is that implementation of processing management services using vendor proprietary database procedures restricts flexibility and choice of DBMS for applications. Finally, current implementations of the two tier architecture provide limited flexibility in moving (repartitioning) program functionality from one server to another without manually regenerating procedural code.

9.22.5 Three Tier Architectures

The three tier architecture (see Three-Tier Software Architectures) (also referred to as the multi-tier architecture) emerged to overcome the limitations of the two tier architecture. In the three tier architecture, a middle tier was added between the user system interface client environment and the database management server environment. There are a variety of ways of implementing this middle tier, such as transaction processing monitors, message servers, or application servers. The middle-tier can perform queuing, application execution, and database staging. For example, if the middle-tier provides queuing, the client can deliver its request to the middle layer and disengage because the middle-tier will access the data and return the

answer to the client. In addition the middle layer adds scheduling and prioritization for work in progress. The three tier client/server architecture has been shown to improve performance for groups with a large number of users (in the thousands) and improves flexibility when compared to the two tier approach. Flexibility in partitioning can be a simple as "dragging and dropping" application code modules onto different computers in some three tier architectures. A limitation with three tier architectures is that the development environment is reportedly more difficult to use than the visually-oriented development of two tier applications. Recently, mainframes have found a new use as servers in three tier architectures (see Mainframe Server Software Architectures).

Three tier architecture with transaction processing monitor technology. The most basic type of three tier architecture has a middle layer consisting of Transaction Processing (TP) monitor technology (see Transaction Processing Monitor Technology). The TP monitor technology is a type of message queuing, transaction scheduling, and prioritization service where the client connects to the TP monitor (middle tier) instead of the database server. The transaction is accepted by the monitor, which queues it and then takes responsibility for managing it to completion, thus freeing up the client. When the capability is provided by third party middleware vendors it is referred to as "TP Heavy" because it can service thousands of users. When it is embedded in the DBMS (and could be considered a two tier architecture), it is referred to as "TP Lite" because experience has shown performance degradation when over 100 clients are connected. TP monitor technology also provides:

- The ability to update multiple different DBMSs in a single transaction;
- Connectivity to a variety of data sources including flat files, non-relational DBMS, and the mainframe;
- The ability to attach priorities to transactions; and
- Robust security.

Using a three tier client/server architecture with TP monitor technology results in an environment that is considerably more scalable than a two tier architecture with direct client to server connection. For systems with thousands of users, TP monitor technology (not embedded in the DBMS) has been reported as one of the most effective solutions. A limitation to TP monitor technology is that the implementation code is usually written in a lower level language (such as COBOL), and not yet widely available in the popular visual toolsets [Schussel 96].

Three tier with message server. Messaging is another way to implement three tier architectures. Messages are prioritized and processed asynchronously. Messages consist of headers that contain priority information, and the address and identification number. The message server connects to the relational DBMS and other data sources. The difference between TP monitor technology and message server is that the message server architecture focuses on intelligent messages, whereas the TP Monitor environment has the intelligence in the monitor, and treats transactions as dumb data packets. Messaging systems are good solutions for wireless infrastructures.

9.22.6 Three-Tier with an Application Server

The three-tier application server architecture allocates the main body of an application to run on a shared host rather than in the user system interface client environment. The application

server does not drive the GUIs; rather it shares business logic, computations, and a data retrieval engine. Advantages are that with less software on the client there is less security to worry about, applications are more scalable, and support and installation costs are less on a single server than maintaining each on a desktop client [Schussel 96]. The application server design should be used when security, scalability, and cost are major considerations [Schussel 96].

9.22.7 Three- Tier with an ORB Architecture

Currently industry is working on developing standards to improve interoperability and determine what the common Object Request Broker (ORB) will be. Developing client/server systems using technologies that support distributed objects holds great pomise, as these technologies support interoperability across languages and platforms, as well as enhancing maintainability and adaptability of the system. There are currently two prominent distributed object technolgoies:

- Common Object Request Broker Architecture (CORBA); and
- COM/DCOM (see Component Object Model (COM), DCOM, and Related Capabilities).

Industry is working on standards to improve interoperability between CORBA and COM/DCOM. The Object Management Group (OMG) has developed a mapping between CORBA and COM/DCOM that is supported by several products.

9.22.8 Distributed/Collaborative Enterprise Architecture

The distributed/collaborative enterprise architecture emerged in 1993 (see Distributed/Collaborative Enterprise Architectures). This software architecture is based on Object Request Broker (ORB) technology, but goes further than the Common Object Request Broker Architecture (CORBA) by using shared, reusable business models (not just objects) on an enterprise-wide scale. The benefit of this architectural approach is that standardized business object models and distributed object computing are combined to give an organization flexibility to improve effectiveness organizationally, operationally, and technologically. An enterprise is defined here as a system comprised of multiple business systems or subsystems. Distributed/collaborative enterprise architectures are limited by a lack of commercially-available object orientation analysis and design method tools that focus on applications.

9.23 Usage Considerations

Client/server architectures are being used throughout industry and the military. They provide a versatile infrastructure that supports insertion of new technology more readily than earlier software designs.

- Maturity

Client/server software architectures have been in use since the late 1980s.

- Costs and Limitations

There a number of tradeoffs that must be made to select the appropriate client/server architecture. These include business strategic planning, and potential growth on the number of users, cost, and the homogeneity of the current and future computational environment.

- Dependencies

If a distributed object approach is employed, then the CORBA and/or COM/DCOM technologies should be considered (see Common Object Request Broker Architecture and Component Object Model (COM), DCOM, and Related Capabilities).

- Alternatives

Alternatives to client/server architectures would be main-frame or file sharing architectures.

- Complementary Technologies

Complementary technologies for client/server architectures are computer-aided software engineering (CASE) tools because they facilitate client/server architectural development, and open systems (see COTS and Open Systems—An Overview) because they facilitate the development of architectures that improve scalability and flexibility.

9.24 Client-Server Applications

Client-Server database applications shift the burden of data torage and manipulation to a dedicated application. In such pplications Access provides the interface to the server data. Some methods for retrieving data from a server are:

- File-server - Database is moved to a network server's hard drive. nefficient due to the data moved across the network to serve ueries.
- Linked ODBC tables - User's SQL is translated to a universal SQL ialect and is passed to the ODBC manager running on the client orkstation.
- Direct connection using Data Access Objects - The OpenDatabase e thod is used. The ODBC connection is given information to open server database directly.
- SQL pass-through (SPT) queries - Used when some advanced feature of the database server is required that ODBC SQL can't understand.
- ODBCDirect - Instead of the query being translated, it is sent directly to the database server.

9.25 The Technical and Business Advantages of Client-Server Computing

Client-Server computing has several major advantages over entralized, "host-based" processing strategies. It is scalable, earning that network administrators can easily add or remove storage, processing and communication resources depending upon the organization's need. Another advantage is that client-server networks are relatively open systems, meaning that a variety of different types of computers, running under various operating systems, can be attached to the same network. In addition, an important advantage is that client-server architecture is more robust" and less likely to fail. This is because processing responsibility is spread over many computers; if one fails, although the services it provides are not available, other components of the client-server network continue to function. Unlike host based systems, there is normally no central component hose failure will incapacitate the entire network.

Migrating from older style systems to client-server architecture as been an expensive and difficult task for many business rganizations. One problem is that the design of software and the etrofitting of older legacy systems to work in a client-server rchitecture is a complex difficult task. Another

problem is that lient-server networks rely on the smooth interaction between lient and server partitions of the applications. This requires hat both ends must be upgraded and maintained when new versions f the application software are released. Finally, client-server ased applications often require greater user knowledge concerning the software and data storage procedures; and migrating to client-server architecture requires considerable retraining of the workers.

In spite of these difficulties, client-server networks provide a definite business advantage in designing many types of application systems. Always keep in mind that a network provides he capability for running centralized, decentralized or distributed applications, and that all three types may run concurrently on computers attached to the same network. It is up to business decision makers to determine how this capacity should e used in regard to a particular business application.

The network of streets and roads that connect our towns and cities permit businesses to distribute their operations if it is desirable, or to keep them centralized. Similarly, client/server networks permit businesses to choose the best strategy to distribute functions depending upon the requirements of any articular application system. An application system should be viewed as the machine (hardware and software), data, and human people and procedures) elements that function together to meet ome particular needs related to a business process.

For a three-tier client-server environment, we move the computation off the client as much as possible, leaving it as just a thin veneer of application necessary to run the GUI. All of the auxiliary additional logic (business rules) and computation are moved onto another platform (not the server). Moving the business rules off to a shared server centralizes all rules making them easier to change and modify. This reduces the complexity of maintaining and sustaining work.

Distribution may be calculated as a factor of frequency and size of messages between entities. The higher the frequency or size of the messages, the greater the need for nearby placement of the two communicating entities. Factoring such figures is done through affinity analysis. Additional considerations must be given to the quality of service and failure models. If a rapid response for certain requests is a requirement (low latency), then the two entities need to be close. If a network partition occurring between entities could be critically damaging, the likelihood of such a possibility must be reduced.

9.26 Pros and Cons of Client/Server

Client/server was originally developed to allow more users to share access to database applications. Compared to the mainframe approach, client/server offers improved scalability because connections can be made as needed rather than being hard-wired. The client/server model also supports modular applications. In the so-called "two-tier" and "three-tier" types of client/server systems, a software application is separated into modular pieces, and each piece is installed on hardware specialized for that sub-system.

One area of special concern in client/server networking is system management. With applications distributed across the network, it can be challenging to keep configuration information up-to-date and consistent among all of the devices. Likewise, upgrades to a newer version of a client/server application can be difficult to synchronize or stage appropriately. Finally, client/server systems rely heavily on the network's reliability; redundancy or fail-over features can be expensive to implement.

9.27 Conclusion

Network clients request information or a service from a server, and that server responds to the client by acting on that request and returning results. This approach to networking has proven to be a cost-effective way to share data between tens or hundreds of clients. Usually the client and server are two separate devices on a LAN, but client/server systems work equally well on long-distance WANs (including the Internet).

Client/server is just one approach to distributed computing. The client/server model has been popular for a long time, but recently peer-to-peer networking has re-emerged as a viable alternative. Other approaches like clustering also have benefits in specific situations.

9.28 Compiling

Compiling our own version of Apache is a 4-step process:

- Downloading the archive and uncompressing it
- Editing the src/Configuration file
- Running src/Configure
- Running make

Begin by downloading the archive from www.apache.org <http://www.apache.org/>. If you are adventurous, then download the latest beta version. Otherwise grab the version without any b's listed in its name. (Notice how small the archives are. They could easily fit on a single floppy disk!)

Next, uncompress and untar the downloaded file with gunzip and tar. This should result in a directory named apache_1.2.5 or something similar. Change directories to the src directory within the apache_1.2.5 directory.

Use your favorite text editor to edit the file named Configuration. The Configuration file lists the modules ("patches") you would like to incorporate into your httpd binary. All of these modules are described in the online documentation. Consider uncommenting the module named status_module.

This module will allow you to monitor the status of your server from a WWW browser. To uncomment a module remove the pound sign (#) preceeding its definition. Next, enter the Configure command. This command very quickly edits a few files in your distribution and returns. Now, enter make and wait. If everything goes well, you should see a lot of text scrolling up your terminal but finally, you will get your prompt back and your Apache HTTP server is compiled.

9.29 Configuration

Configuring the Apache server requires the following steps: copying the compiled binary up one directory level editing conf/httpd.conf editing conf/srm.conf editing conf/access.conf Copy your newly created Apache server application up one directory out of the src directory. To make your life easier, this directory should also contain the htdocs, logs, conf, and cgi-bin directories.

Change to the conf directory and cp httpd.conf-dist httpd.conf. This will make sure you have an original version of the httpd.conf file. Using your text editor again, edit httpd.conf. In this

file you will want to change the following definitions: Port Enter an integer. The default port of HTTP servers is 80. If you are the superuser of your Unix computer, then you will be able to run the server on this port.

If you are just experimenting, then use a port like 8000. Example: 8000 User Enter a username. The user variable is used to tell the server who owns the httpd process once it is running. Since the httpd application will have to read and write files on your computer, it will be necessary to select a username that has just the right number of priveleges. It is best to create a bogus user named "nobody", give them few priveleges, and have the httpd application run under that user. Example: nobody Group Enter an integer. Every username should be assigned to at least one usergroup. Like usernames, groups help define priveleges on the computer and you should have a group defined that has limited authority.

Enter a pound sign followed by an integer denoting the group the httpd application should run under.

Example: #-1

- ServerAdmin

  Enter here the e-mail address of the person who should get messages when things go wrong. (Obviously this will never be needed.) Example: webmaster@lib.univ.edu
- ServerRoot

Enter the full path to the httpd application. Example: /usr/local/apache ServerName Enter here the IP address or fully qualified domain name of your Unix computer. This address or name will be the name returned to any clients connecting to your server. Do not make up a name. Example: dewey.lib.univ.edu

Next, cp srm.conf-dist srm.conf so you have a backup of the file.

- DocumentRoot

Enter the full path to the location where you will be saving your HTML documents. Example: /usr/local/apache/htdocs

- ScriptAlias

If you plan on using CGI scripts, then you will want to change this configuration to map a virtual directory to a real directory containing the scripts. Example: /cgi-bin/ /usr/local/apache/ cgi- bin/ AddHandler Again, if you want to run CGI scripts from your server, it is convienent to create a filetype here with the .cgi extension.

Example: cgi-script .cgi

You are more than half way there. Keep editing! The final configuration requires you to edit access.conf. First cp access.conf-dist access.conf. The access.conf file is made up of <Directory></Directory> pairs. By default the file defines access configurations for your HTML document root and your cgi-bin directory. You will want to change the values already in the file to the same values you specified for the DocumentRoot and ScriptAlias above.

Examples: <Directory /usr/local/apache/htdocs> and <Directory /usr/local/apache/cgi-bin>.

If you compiled your httpd binary with status_module, then uncomment all the lines in the <Location /server-status> directive and edit the allow from line to include your domain. If you do this, then once your server is running you will be able to enter a URL like http:// www.lib.univ.edu/server- status and see how your server is running.

9.30 Windows Configuration

Installing and configuring the Windows95/NT version of Apache includes two options. First, you can download the source code and compile it. This requires a Microsoft C++ compiler. Alternatively, you can download a pre-compiled version of the application. If you own the compiler, then you don't need any instructions. Otherwise, download the pre-compiled version. Whether you have downloaded the source-code or the pre-compiled version, you will want to edit the *.conf files just as above.

9.31 Starting Up

It is now time to actually start up your server. Move up one directory and run the httpd application with ./httpd -f path, where path is the fullpath name to your httpd.conf file. For example, httpd -f /usr/local/apache/conf/httpd.conf. (If you are running the Windows version, then you will want to run the application from the command line and specify the fullpath of the httpd.conf file as an argument: c:\apache\apache.exe -f c:\apache\conf\httpd.conf.) If an error is returned, then read it carefully and try to resolve the problem. If no errors occured, then use your browser to open a connection to your server.

Remember to specify the Port in your URL as you defined above. If for some reason you do not get the "It worked!" message, then check the contents for your logs/error_log file for clues to what went wrong. At this point your are either overjoyed or overwhelmed. Those of you who are overjoyed should now go back to your configurations to make more specific modifications. Those of you who are overwhelmed are encouraged to visit <news:comp.infosystems.www.servers.unix> and try to get some assistance there. Carefully worded questions will get responses.

9.32 APACHE

As you may expect, restricting access to your Apache server is more complicated when compared to the desktop servers. The simplest and most direct way to restrict access requires you to create .htaccess files within the restricted directories. These .htaccess files are exactly like the instructions in your access.conf file except they do not require the

<Directory></Directory> nor <Location></Location> directive tags.

IP and domain name restrictions Below is a simple sample .htaccess file restricting access based on addresses. The first line specifies that all GET and POST queries will be restricted. Next, the restrictions will be processed in the deny-allow order since later configurations override earlier ones. Then the hosts and/or IP address that are denied and allowed are specified. Finally, the file is closed. # this is an .htaccess file for IP addresses # limit the types of access <Limit GET POST> # define how restriction will be processed order deny,allow # define who to deny and allow deny from all allow from .ncsu.edu # close the directive </Limit> Usernames and passwords Restricting access based on usernames and passwords first requires you to define users and sets of users called groups. Then you create .htaccess files specifying these users and/or groups. If it hasn't been done before, you will have to compile the htpasswd program. It is located in the support directory of your original distribution. To compile the program change directories to the support directory and enter make. This should result in the creation of a

number of utilities one of which is htpasswd. For the .htaccess technique to work, you must edit your access.conf file and specify parent directories of your restricted directories with the AllowOverride Limit option. Next, using htpasswd you will create new users. The command takes the following form: htpasswd [-c] filename username where: -c This is used only the first time htpasswd is used. Its purpose is to create the password file filename This is the full path name to your password file. It can be any path or name, but make sure it is not in your server's document directory structure. Example: /usr/local/apache/conf/passwd username This is the name of the user you are creating. Example: eric After this, you have to create groups. Groups are ASCII text files with the following form: group: member member member where: group This is the name of a group. Example: users member This is the name of somebody previously defined by the htpasswd program. Example: eric Now you are finally ready to create your .htaccess file. Specify the realm using AuthName. The value of AuthName will appear in the message asking for username and password. Specify the authorization type with AuthType. (The majority of the time this will be Basic.) Echo the location of your password and group files next. Define the restricted methods of access. Specify what groups and users have access to this directory. Last, close the directive. Here is a example .htaccess file limiting users to passwords. # this is an .htaccess file for passwords # define the realm AuthName The Super Secret Space # define the authentication type AuthType Basic # where are the password and group file AuthUserFile /usr/local/users/Eric/apache/conf/passwd AuthGroupFile /usr/local/users/Eric/apache/conf/group # limit the types of access <Limit GET POST> # say exactly who can access require group users # close the directive </Limit> Once this sort of .htaccess file is saved in a directory, the first time a user tries to access the directory they will be asked to enter their username and password.

9.33 Quid Pro Quo

IP and domain name restrictions Using Quid Pro Quo, restrictions based on IP addresses or domain names are called Allow/Deny. This server only permits you to "allow or deny" sets of IP address or domain names for your entire server, not parts of it. It is unfortunately an all or nothing deal. To set up this sort of restriction, select Server Settings... from the Control menu. Then select Allow/Deny from the resulting Configure Quid Pro Quo dialog box. Next you will want to add a new item and enter either IP addresses or domain names. Like the other servers, you do not have to specify the entire address or name, just enough to make it meaningful. Your configuration takes place as soon as you close the configuration dialog box. Usernames and passwords Restrictions to parts of your server using usernames and passwords are called realms. This method is more secure than the first method. Implementing it is a two step process. First, using the Configure dialog box, select Realms. Now you can enter any name you want for a realm and then a string of characters that will be used to match parts of client requests. Any URL containing the string of characters will be restricted by usernames and passwords. The second step is to choose Passwords from the Configuration dialog box. From here you can create new user names and passwords, and then associate them with realms created in the first step. Like IP/domain name restrictions, these configurations take place as soon as you close the Configuration dialog box.

9.34 Website

Like Quid Pro Quo, WebSite provides the means of access control through a series of dialog boxes. Its features are more robust than Quid Pro Quo's and at least on par with Apache's. Its only a technical drawback is it inability to edit settings once they have been created. Many times, to make changes you must delete the old settings and recreate them with your new edits. IP and domain name restrictions The first step to limiting access by IP addresses and domain names is to open the WebSite Server Properties window and select Users. Next, create an authentication realm by clicking the New... button. Any label for your realm will do, just make it meaningful to yourself. Third, select Access Control and create a new URL Path.

Enter the full path from your server's root to the directory you want to restrict. Select a realm too. Finally, specify what IP addresses and/or domain names will be allowed access to your realm using the Class Restrictions panel. If you want to restrict access to particular hosts, then select the Deny, then Allow button and enter the IP addresses or domain names you are allowing access. Conversely, if you want to deny access to particular hosts (people who might be malicious for example), then select the Allow, then Deny button and enter the hosts to deny.

Close the WebSite Server Properties window and when you hear a system beep you will know the configuration has taken place. Usernames and passwords To limit access to some or all of your site, begin again with the WebSite Server Propertites window and the Users section. This time select or create a new Authentication Realm and create a new user using the New... button of the User panel. Second, go to the Access Control section and select or create a new URL Path for password protection. Remember to specify the full path name of the password protected directory begining with the root of your WebSite server. Finally, select the users allowed to access the directory by using the Add... button of the Authorized Users and Groups panel. Using the Add... button shoul result in a list of users you previously created. Again, changes won't take effect until you close the WebSite Server Properties window and you hear the system beep.

9.35 Three-Tier

A special type of client/server architecture consisting of three well-defined and separate processes, each running on a different platform:

1. The user interface, which runs on the user's computer (the client);
2. The functional modules that actually process data. This middle tier runs on a server and is often called the application server; and
3. A database management system (DBMS) that stores the data required by the middle tier. This tier runs on a second server called the database server.

The three-tier design has many advantages over traditional two- tier or single-tier designs, the chief ones being:

- The added modularity makes it easier to modify or replace one * Tier without affecting the other tiers; and
- Separating the application functions from the database functions makes it easier to implement load balancing.