Chapter 3

Database Systems

This chapter focuses on the different types of database systems that have been developed. Firstly, a brief history on file systems is given, which includes the file system's shortcomings. Next, the different database implementation models are discussed. Finally, object-oriented databases and multimedia databases are discussed.

3.1 File systems

The discussion concerning databases would not be complete without first discussing the historical roots of the database: the File System Era.

3.1.1 Manual file systems

Before databases were used, the management of a small organisation was able to control and maintain its data by using a manual file system, which traditionally consisted of a collection of documented files that were stored in a filling cabinet. The data would be organised according to how it would be used i.e. the data in each file folder was logically related. A small law firm would therefore typically categorise its files so that each folder would contain a specific client's details.

The manual file system served the needs of most small organisations with few reporting requirements. However, when the organisation needed to expand and had more complex reporting requirements, problems would arise due to the difficulty in maintaining the data using the manual file system. Large and accurate reports often needed to be efficiently made, especially in government related organisations and large public companies. Due to the difficulty in locating the desired data in the growing collection of file folders, report generation from a manual file system would be slow and difficult, and often did not meet the set requirements. A computer-based system was needed that could be used to manage data more efficiently and easily generate the required reports (Livadas, 1990).

3.1.2 Computer-based file systems

In most cases, it was technically complex to convert from the manual file system to a computer-based system. The services of a data processing (DP) specialist was needed to create the essential computer file structures, write the software for managing the computer file structures, and develop the application programs that

would produce the required reports for the organisation. As a result, a large number of internal computer-based systems were made.

At first, the computer files resembled the format of the manual files. A *field* is defined as a group of characters that have an explicit meaning, such as a patient name or address. A *record* is defined as a group of one or more fields that are logically associated, which describe a person, animal, place or other object. Each file contains logically related data (raw facts), all organised into fields and records, with the smallest recognisable piece of data being a single character (i.e. a file is a collection of related records). The following is a straightforward example of a dentist's patient data file:

Full_Name	Address	Phone_Num	Owing	Last_Visit
Sipho Dlamini	12 Hillebrand St. BLVI	782-7919	0.00	12-07-01
Cedric Perry	144 Park Rd. KIMP	291-3596	130.00	23-02-02
Pat A. Lee	78 Denville St. FROS	928-3952	0.00	19-12-01
James R. Baker	122 Giraffe Rd. ANML	722-9265	170.00	12-09-02
Hilda Samuel	9 Ribbon Rd. PRNT	588-7932	0.00	02-07-01
Anne Parker	38 Kingsley St. PLCE	781-6925	52.00	09-04-02
Suraid Suliman	46 Jeremiah St. BLVI	782-3277	0.00	05-01-02
Denzil P. Moore	81 Mercury St. DOBB	131-7418	0.00	07-08-01
Mario A. Zampi	7 Sugar St. FROS	928-6562	-23.00	02-02-02
…	…	…	…	…

Table 3.1: Patient file example

The DP specialist would then use the above patient file (and any other necessary data files) to write programs that will generate the necessary reports for the dentistry practice. A more complex example would be a department of a large public company. The DP specialist would need to create a much larger number of data file structures than in the above patient file example, and a larger variety of reports would need to produced by developed application programs. Developing these programs took a reasonably large amount of time, but once completed, the computer-based file system saved the department's manager(s) a great deal in time and effort.

A large organisation consisting of a number of departments (e.g. Marketing, Sales, Human Resources, etc.) required a set of programs to be written for each department. The DP specialist not only needed to write new data file structures and reporting programs for the remaining departments, he/she also needed to write additional programs to produce new reports for the first set of departments already containing data files, as well as to maintain all the departments. Department data often needed to be shared, which required programs that could

interface and perform tasks between these departments. The time constraints placed on the DP specialist, due to the continual file system growth, forced the company's management to allow more programmers to aid the DP specialist, who now took the role of a DP manager. The DP manager would then supervise what became known as the DP department (Rob & Coronel, 1997).

3.1.3 Shortcomings of the file system

Data-retrieval tasks for the file system required extensive programming in a third generation programming language (even for the simple tasks), which is usually a high-skill and time-consuming activity. The above patient file example is represented in a logical manner which is easier for humans to interpret, but is different from the way in which the computer stores the data on disk. Using the file system, the programmer must understand the physical file structure, so that he/she can write the programs that make use of the correct data characteristics and access paths. The larger the file system, the more difficult it is to produce error-free programs that manage the necessary data. Since it is necessary to code third generation language programs to generate any report, it is almost impossible to perform ad hoc queries; programs to generate new reports may only be completed weeks or even months after their request.

Another hassle concerning file systems is that any modification made to a particular file's structure requires changes to be made to all programs that make use of that file. In the above patient file example, the addition of a field called Medical_Aid_Details requires us to make changes to all previously written programs that make use of the patient file. If access to a data file is dependent on the file's structure, the data file exhibits *structural dependence*. Similarly, any modifications made to a particular file's data types requires changes to be made to all programs that access that file. In my example above, if the Phone_Num field must be changed from type string to long integer, all previously written programs that access the patient file would need to be modified. If changes to a data file's data characteristics forces changes to be made to programs accessing that file, the data file is said to exhibit *data dependence.* The file system in general exhibits structural dependence and data dependence (as can be seen in the above example), which makes it difficult and time consuming to make modifications to a file system while ensuring that the data access programs function correctly.

Due to the complexity and design of a large file system, it is often difficult to gather data from a number of sources, which as a result causes the designer to

store the same data in a number of different files, thereby producing *data redundancy*. For example, a music store may contain two departments, a Sales department, containing information such as who bought their products (including delivery information), and an Accounts department, containing information such as how much each client owes the store. Each department contains client data stored in files Sales_Clients and Accounts_Clients respectively. To reduce complexity, the DP specialist designed the file system so that the client address is duplicated in both files, and can therefore easily be accessed by each department. The problem with this approach is that, due to the data redundancy, the data capturer could easily misspell the address in one of the files, or even completely forget to update the address in the one file. This causes *data inconsistency*, which can have devastating effects. The Sales department might then send the ordered product by post to the correct address, but send the account to an incorrect address; or even worse, a client could be billed for an ordered product that he/she won't receive!

The ideal design for any data storage system is if a data capturer need only make changes to one place in the system, thereby eliminating any data redundancies which cause *data anomalies*. This is because in a system containing data redundancies, any necessary change made to these fields must be identical in all corresponding fields, in order to maintain data consistency. Three types of data anomalies are defined. *Modification anomalies* are caused when a single change in a particular field value necessitates the same data to be updated (i.e. repeated) in other parts of the system. *Insertion anomalies* are caused when certain field values for an existing record must be inserted repetitively for a set of new records. Lastly, *deletion anomalies* occur when a deleted record causes changes to be made in a number of places in the system that was previously referencing that data (Livadas, 1990).

3.2 Databases

A database can be defined as a shared, integrated computer structure that contains a collection of end user data and metadata. The data characteristics and the relationships that exist between the end user data are described by the metadata. The database therefore contains logically related data stored in a single data repository, as apposed to the individual and unrelated files that are characteristic of the file system. Due to this different approach in storing, managing, and accessing the end user data, a well designed database eliminates most of the previously discussed problems that are resident in file systems,

specifically the data inconsistencies, data anomalies, and the data structural dependency problems.

A *database management system* (DBMS) is special software designed for databases that is used to manage the database structure, and control access to the end user data and metadata stored in the database. The DBMS allows multiple applications and users to access the database concurrently, and also hides much of the internal details of the database from the application/user, making the database environment more secure and easier to use. As can be seen in the below diagram, the DBMS interfaces between the user/application and the database, facilitating all communication between the two by translating user or application requests into the code required to retrieve the information from the database (Rob & Coronel, 1997).

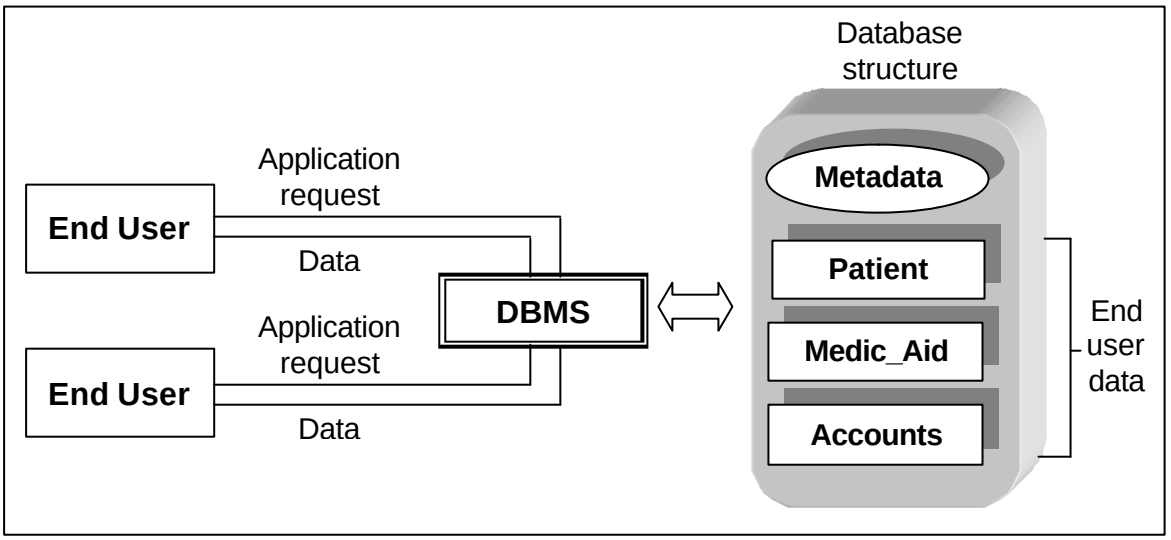

Figure 3.1: DBMS environment (Adapted from Rob & Coronel, 1997)

3.2.1 Advantages of a DBMS

A DBMS has many advantages over previous data management and storage systems such as the file system. The main advantages of a DBMS are:

- ***Efficient access to data***: A DBMS is specifically designed to optimise the storage and retrieval of data, which is especially needed in a multi-user network where the database must be accessed remotely over the network.
- ***Data independence to applications/users***: The DBMS allows an application/user to access the data in an abstract manner. Unlike the file system, this advantage allows the application/user to access the data without

the need to manage access paths, and minimises the amount of effort needed to change data characteristics.

- ***Centralisation of data***: The administration of data when accessed by a number of users is simplified when the data is centralised. A well designed database should minimise data redundancies and data anomalies.
- ***Data security***: Now that the data is always accessed via the DBMS, authentication, access control, and other security measures can be enforced by the DBMS.
- ***Data integrity***: The DBMS can also ensure specified data integrity constraints on the data, such as to enforce the uniqueness of a particular attribute.
- ***Concurrent access***: The DBMS handles concurrency in a manner that will allow its users to think that they are accessing the database exclusively. The user/application therefore does not need to directly deal with concurrency issues relating to the database.
- ***Reduced application development time***: The DBMS not only allows the user/application to access the database via a high-level interface, it also contains many important functions that can be used on the data which will help improve development time. These applications should also be more robust due to the fact that the DBMS handles the tasks that would otherwise need to be programmed in each application.

It is clear from the above DBMS advantages that the database resolves many of the shortcomings that are typical of the file system. There are, however, certain cases where a database would not be adequate. These cases include certain specialised applications, such as those with real-time constraints, which require more efficient data access than that which a DBMS can offer. The DBMS is nevertheless suited to most application domains that involve large-scale data management (Ramakrishnan & Gehrke, 2000).

3.2.2 Database Models

A *database model* can be defined as a theoretical manner of representing the data structures and relationships found within a database by using a collection of logical constructs. A number of different database models were formulated in order to solve the previously discussed file system shortcomings. These database models can be grouped into two categories: *implementation* models and *conceptual* models. Implementation models focus on how the data is represented in the database, and include the hierarchical database model, the network

database model, and the relational database model. Conceptual models focus on what is represented in the database, and include the object oriented model and the entity relationship model.

3.2.2.1 The Hierarchical Database Model

A company called North American Rockwell was the prime contractor for the Apollo moon project in the nineteen sixties. The company started to develop its own database system in order to manage all the project related data, but after an audit of its computer tapes, found that over sixty percent of the data was redundant. This forced North American Rockwell to develop a new strategy for managing the large amount of data.

Using parts of some existing database concepts, the company developed software known as Generalized Update Access Method (GUAM). This concept was based on the fact that the smallest components in the structure would form larger components, while these larger components would assemble into even larger components, and this process is repeated until the parts eventually make up the final component. The GUAM thus formed a hierarchical structure illustrated in the following diagram:

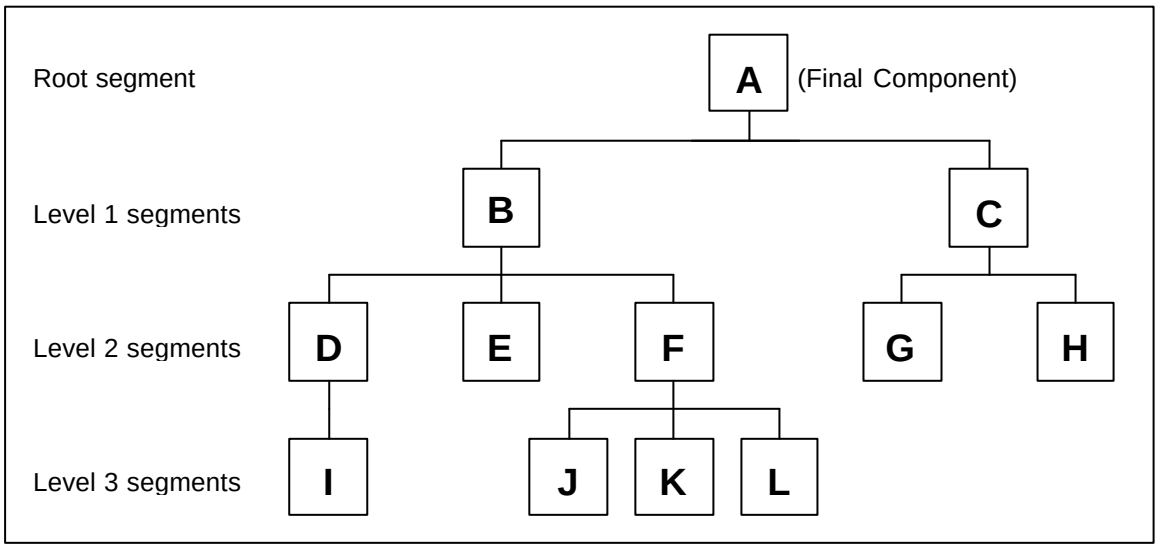

Figure 3.2: A hierarchical structure (Adapted from Rob & Coronel, 1997)

IBM then joined North American Rockwell to expand GUAM's capabilities, resulting in the replacement of the computer tape medium with more up-to-date computer disk storage that allowed specific data within the storage medium to be referenced via pointers. The joint effort resulted in what was known as the

Information Management System (IMS). This model was the first major commercial implementation of an increasing number of database ideas developed to oppose the computer file system's shortcomings.

As can be seen in the above diagram, the user or application program views and accesses the hierarchical database as a hierarchy of segments. A *segment* corresponds to a file system's record type, and therefore the hierarchical database is a collection of records represented as the above hierarchical tree. As with the standard tree terminology, segment A is called the *root* of the tree, and is the *parent* of segments B and C, which are the parents of segments D, E, F and segments G, H respectively. Inversely, segments J, K, L are the *children* of segment F, and segments B, C are the children of root segment A.

The hierarchical model allows its designers, users, and applications to view and access its structure logically, instead of physically. The DBMS manages the physical structures needed to implement the hierarchical structure. The users/designers therefore only need to concentrate on the logical side of the problem at hand. The DBMS physically stores and accesses the hierarchical structure by defining a path that traces the parent segments to the child segments, starting from the root, in a preorder traversal. The path followed to segment H in the above diagram would therefore be: A, B, D, I, E, F, J, K, L, C, G, H. To improve efficiency, the database designer must ensure that the most frequently accessed segments are on the left side of the tree (since these segments are accessed first). The hierarchical structure represents a number of one (the parent) to many (the children) relationships.

There are three main problems concerning this model. Firstly, a major change in the database design (such as the deletion of a parent segment) usually still requires a number of changes to be made to the application programs. Secondly, relationships such as many-to-many, or a child with many parents, are difficult to implement in this model which is mainly suited to one-to-many relationships. Thirdly, extensive programming is required to develop a user-friendly system based on this model. Although the hierarchical database model is no longer considered a modern database implementation model, its design formed an important part of our database history, whose limitations led to a different view of designing databases (Ullman, 1982).

3.2.2.2 The Network Database Model

The network model is in many ways similar to the hierarchical database model. There were a few reasons why the network model was created. It was specifically designed to represent complex data relationships more effectively than the hierarchical model. As in the hierarchical model, the relationships between the records may be perceived as one-to-many relationships. Unlike the hierarchical model however, the network model can have records that have more than one parent. Other reasons for the development of this newer model was to improve database performance, and to develop a set of database standards.

It was becoming increasingly important to develop a set of database standards; problems due to the lack of these standards were especially felt amongst programmers because the database designs and applications were less portable. Lack of these standards also slowed down progress in the search for improved database models. In 1971, the Conference on Data Systems Languages (CODASYL) group put together the Database Task Group (DBTG), which had as its aim to define standard specifications for an environment that would allow database creation and management.

The finalised DBTG report included specifications for three integral database components. The first component is the network schema, which is an abstract organisation of the whole database as seen by the database administrator. Included in this is a record type for each record, the components of each record, and the definition of the database name. The second component, the subschema, defines the part of the database as viewed by the users/applications that make use of the data. The third component is a data management language that is used to define the data structure and data characteristics, and to manipulate the data.

The DBTG specified three distinct management language components to produce the desired standardisation for the integral database components. For the first, a schema Data Definition Language (DDL) is used to allow the administrator to define the schema components. Secondly, a subschema DDL is used to allow the user/application to define the necessary database components. Thirdly, a Data Manipulation Language (DML) is used to manipulate the data. All network database software designed for mainframes conformed to the above DBTG standards. This allowed designers and users to change commercial applications without any significant setbacks while operating at the conceptual or schema level.

Some terminology is needed before giving an example of a network model. Firstly, a *set* is equivalent to a relationship or an association in other model terminology. Every set is made up of at least two records, an *owner* record and a *member* record. An owner record is equivalent to the hierarchical model's parent definition, while a member record is equivalent to the hierarchical model's child definition. The main difference between the two models (hierarchical and network) is that a member in the network model can belong to more than one set as a member (i.e. a member can have more than one owner). The following diagram is a simple example of network database model.

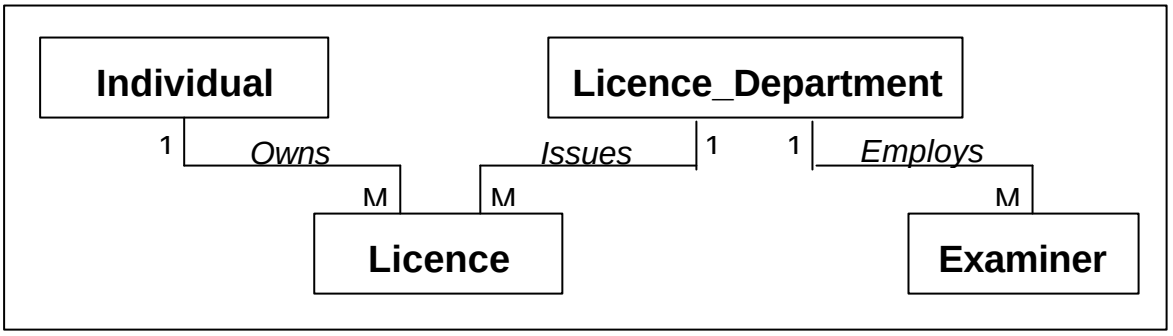

Figure 3.3: An example of a network database model

In the above example, it is clear that 'Licence' is a member of the set 'Owns' *and* of the set 'Issues'. In this network model, three one-to-many relationships exist. An individual may own a number of licences (e.g. Motorbike, Code B, and Heavy-duty vehicle licences). A particular licence is however owned by only one individual. Similarly, a licence department issues many licences, but a particular licence is issued by only one licence department. In the last set, a licence department (usually) employs several examiners, while a particular examiner is (usually) employed by only one licence department.

The network database model's main advantages over the hierarchical model is that it can implement many-to-many relationships far more easily than the former model, and that it achieves enough data independence to at least partially shield the application programs from its physical storage details. A big disadvantage of the network model is that the designer/programmer must still be familiar with the database structure in order to fully benefit from its efficiency (i.e. knowledge of some physical details is still necessary). Another disadvantage is that changes made to the database structure usually require subschema definitions to be revalidated before application programs can use the database (data independence is achieved, but not structural independence). As with the hierarchical database model, it is difficult to make applications that access the database user-friendly. A number of front-ends have been made to make it easier

to use the network model, most of which are based on the relational model (Rob & Coronel, 1997).

3.2.2.3 The Relational Database Model

Although the hierarchical and network database models greatly improved data storage and access, the ability to create a well designed database was very difficult due to the database's structural complexity, and the need to be familiar with the database structure in order to maximise efficiency. The time taken to perform queries on the database needed to be improved, and the problems caused by structural changes made it difficult to efficiently develop robust application programs that required data from the database.

In 1970, E.F. Codd of IBM proposed a new model known as the relational database model, which revolutionised the database field. Although impractical at first, exponential improvements in computer power soon allowed the model to be implemented. The relational model has advanced to become the current database standard. It now forms the basis for leading DBMS products on the market, including Microsoft's Access and SQLServer, Oracle, IBM's DB2, Sybase, Informix, Paradox, and FoxBase.

The relational model is logically simple and powerful to use. The database is a collection of one or more *relations*, with each relation being the equivalent of a matrix (or table) of rows and columns. This matrix representation allows inexperienced users to understand the contents of a relational database, and also allows queries to be easily performed through high-level (fourth generation) languages. The relational database management system (RDBMS) carries out the same basic functions as those performed by the hierarchical and network models, as well as a multitude of other functions that make database design and access easier to accomplish. The RDBMS lets the designer or user/application be concerned with only the logical view of the database, the physical issues are managed entirely by the RDBMS.

A relation is the main building block of a relational model, and consists of a *relation schema* and a *relation instance*. A relation schema describes the column titles for a table, and is made up of the relation's name, the name of each column/field/attribute in the table, and the domain of each field. A relation instance is a set of records/tuples, in which each record has the same number of fields as

the defined relation schema. I will use the following 'Bookings' example to illustrate:

	Ticket_Num	Venue_Num	Booked_Date	Time_Slot	Seat	Show_Code
	0018901	23	12/01/2002	2	178	2
	0018902	7	13/02/2003	3	45	1
	0018903	18	14/12/2002	4	210	3
	0018904	2	09/03/2003	1	13	1
	0018905	7	22/11/2002	3	39	3

BOOKINGS table

Link between the two tables

	Show_Code	Show_Name	Start_Date	End_Date	Theatre_Num
	1	Terminator 3	12/02/2003	13/05/2003	5
	2	Rat Race	01/01/2002	17/03/2002	2
	3	Star Trek Nem.	12/11/2002	15/02/2003	7

CURRENT_SHOWS table

Table 3.2: 'Bookings' example of relational tables

In the Bookings example, the table CURRENT_SHOWS contains the following five fields (in the format *field name*: *domain name*) – Show_Code: integer; Show_Name: string; Start_Date: date; End_Date: date; and Theatre_Num: integer. These fields, domain names, and a set of associated field values (e.g. the maximum number of characters in a string), together with the relation's name CURRENT_SHOWS, make up the relation schema for this relation. A relation instance for the BOOKINGS table can be a subset of the above table (such as all records with the value Show_Code = 3), or even the entire table (as it is above), at this point in time. The order of the records is not important, and neither is the order of the columns if the query language used accesses the data by column name (as opposed to column number) (Ramakrishnan & Gehrke, 2000).

As in the Bookings example, a user or designer perceives the entire relational database to be a collection of tables containing data, each usually related to one or more tables by a common attribute(s) (although it is not mandatory for every table to be related to at least one other table). An important property of relational databases is that although the tables are physically independent of each other, a logical link between the tables allows us to associate the data between them. The above illustration shows that a logical link occurs between the BOOKINGS table and the CURRENT_SHOWS table. The attribute Show_Code is common to both

tables, and it is this field that forms the logical relationship between them. It can be determined from this relationship that ticket number “0018903”, booked for the 14 December 2002 (seat number 210), is associated with Show_Code number 3, which according to the table CURRENT_SHOWS represents the movie “Star Trek Nemesis”.

This relationship will correctly determine the associated ticket data on the condition that each value in the Show_Code field of table CURRENT_SHOWS is unique. If two entries exists with value 3 for example (and different show names), the retrieved associated data may be incorrect. A key is a device that helps define the relationships between the tables. The Show_Code field of table CURRENT_SHOWS is called a *primary key*, since this field is used to uniquely identify each record in that table, and does not contain any null values. A *foreign key* is a field that identifies records in a different table i.e. it forms the link between that table and the primary key (which is the common field) of another table. The Show_Code field of the BOOKINGS table is a foreign key of that table which identifies the corresponding movie information in the CURRENT_SHOWS table. Note that the BOOKINGS table contains potential foreign keys to other tables not shown in the above example. The Venue_Num and Time_Slot fields contain numbers that should be primary keys of a VENUE table and TIME table respectively, with each table listing more detailed information such as venue descriptions and time schedules.

A relational database can be thought of as a collection of relations/tables with unique relation names. It is easier to understand than the previous two database models because it resembles a file from a conceptual point of view. Although a relational database table looks very much like a file, there is a crucial difference. A table in the relational model is simply a logical structure, and therefore exhibits complete data *and* structural independence, unlike the file system (which exhibits data and structural *dependence*). The relational model also has a great advantage over the hierarchical and network models, which exhibit data independence but not structural independence. This means that any structure changes made to a relational database do not require changes to be made to all application programs that access the database.

The relational database has more than just the structural independence advantage over previous models. Another very important advantage is its powerful and very flexible query capability. Ad hoc queries can easily be made by using a fourth-generation language (4GL), which requires a user to specify only *what* must be done, without needing to specify *how* it must be done. The most

common 4GL used for relational database software is Structured Query Language (SQL), which will be discussed after a brief explanation of the *entity relationship model* (Brathwaite, 1991).

(a) The Entity Relationship Model

An *entity* can be simply defined as a person, event, place, or object for which we intend to gather data e.g. a car manufacturer may have 'models', 'parts', and 'suppliers' as defined entities. An entity has a number of characteristics/attributes that are used to distinguish it from other similar entities. Related entities when grouped together form an *entity set*. A table contains related records with the same attributes; we can therefore say that a table has the equivalent meaning as an entity set.

An entity relationship (ER) model is a conceptual model used to describe the data relating to a real-world activity in terms objects, their properties, and the relationships between them. It was originally described in a research paper in 1976, after which certain alterations were made to make it the model it is today. The ER model is largely used for initial database designs, and is particularly suited to the relational database model because of its logical structure. It allows a database designer to transform informal user requirements into a more detailed and accurate description that can be implemented by a DBMS.

The first three steps in the database design process is made easier with the aid of an ER model. The first step, *requirements analysis*, involves understanding what data needs to be stored, how applications will access the database, and the determination of any specific performance requirements. The second step is *conceptual database design*, and involves the development of a high-level description of the data that will be stored in the database, based on the requirements analysis outputs. This step is carried out by using the ER model. The third step, *logical database design*, involves choosing a DBMS, and in the case of an RDBMS, converting the ER model into a relational database schema.

The ER model is made up of 3 major components: entities, attributes, and relationships. As already mentioned, an entity set is equivalent to a table, and the attributes of a table form the common properties of the entity set. A relationship is the link or association between two or more entities. Three types of relationships exist: *one-to-one*, *one-to-many*, and *many-to-many*. Briefly, the main process to be followed when creating an ER diagram is to convert data requirements (from the requirements analysis) into entities, including each entity's attributes that may

be needed by application programs. Primary keys and foreign keys are defined for each entity (if possible), and the relationships between the entities are formed. The following is an ER diagram for the bookings example:

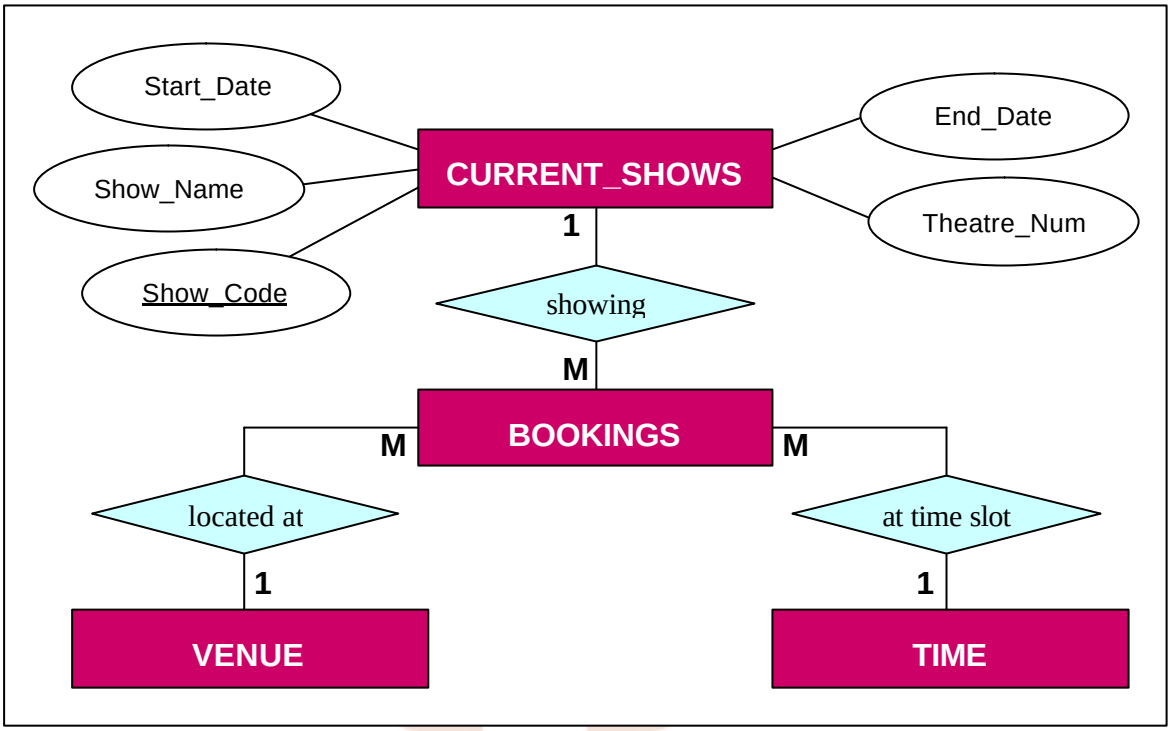

Figure 3.4: ER diagram for the Bookings example

The above notation conforms to the ER model notation, where rectangles represent entities, ovals represent attributes, and diamonds represent relationships. Note that the example ER model shows the attributes for the CURRENT_SHOWS table only (a complete ER model displays the attributes of each entity). The VENUE and TIME tables have been included and also form one-to-many relationships with the BOOKINGS table. The Show_Code attribute of table CURRENT_SHOWS is a primary key and is depicted by underlining the attribute (Ramakrishnan & Gehrke, 2000).

(b) Structured Query Language (SQL)

SQL is the standard 4GL (nonprocedural language) used for relational databases. The RDBMS uses SQL in order to translate an application/user query into the detailed code needed to retrieve the required data. SQL has a limited vocabulary, and is easy to learn. The basic SQL command set consists of about thirty commands, although many software developers have produced extensions to increase the number of available commands. To make a truly user-friendly database application, however, a programmer should incorporate a GUI into the

application that works in combination with SQL to retrieve, add, delete, or change any data in the relational database.

SQL is able to perform all the essential functions on a database via its easy-to-use query capabilities. These functions are divided into three categories. The first category of functions, *data definition*, allows for the creation of the database and its tables. The second category, *data management*, includes commands that allows a user/application to insert, update, and delete data in the tables. The third category, *data query*, includes commands that retrieve data according to properties required by the user/application in the SQL query.

An important attribute for any database 4GL is that it must conform to a basic standard in order to ensure portability across DBMS's. The American National Standards Institute (ANSI) prescribed a standard SQL, called SQL-92 (the most recent version) to ensure this. SQL's basic vocabulary is made up of less than 100 words, which together with its simple query structure makes it an easy language to learn. Many RDBMS developers make special enhancements to the SQL-92 standard, resulting in several SQL dialects. These differences are nevertheless minor, and should not greatly affect portability (Abiteboul, Hull & Vianu, 1995).

SQL has a logical query format. The SELECT command, for example, is used to retrieve table data according to specified attributes. It has the following format:

SELECT {*attribute(s)*}
FROM {*table(s)*}
WHERE {*condition(s)*};

For example, if we run the following query on the above BOOKINGS example,

SELECT BOOKINGS.Ticket_Num
FROM BOOKINGS, CURRENT_SHOWS
WHERE (BOOKINGS.Show_Code = CURRENT_SHOWS.Show_Code) AND
(CURRENT_SHOWS.Show_Name = 'Star Trek Nemesis');

we will retrieve the list of all ticket numbers associated with the movie "Star Trek Nemesis" i.e. ticket numbers '0018903' and '0018905''.

3.2.2.4 Object-Oriented Databases

An *object* can be described as a computer structure or entity that encapsulates an information element, which includes text, images, audio, video, and other data.

Using this description, *object-orientation* can be defined as a collection of design and development principles based on objects. It was traditional to separate the data from the algorithms used to process that data. This was sufficient in almost all application areas up to the late 1980's, since data up to that time was only *passive* i.e. its purpose was solely for storage and retrieval of potential information that needed to be processed by application programs or outside procedures.

The functional requirements of database applications have changed in the last two decades from simple applications that retrieve a small quantity of textual records, to complex applications that require the retrieval of multimedia data and compound data such as arrays and complex structures. Database designers and programmers soon faced difficult requirements due to the increasing data volume and constantly changing databases that need to manage *active data* – data that can alter the presentation or characteristics of passive data, and can even be part of a decision making process (e.g. by executing encapsulated code).

The simpler applications that accessed a relatively small amount of data were made using procedural languages, which were required to be initiated in order to produce information from the (passive) data. To solve the complexity problems, not only should the application program change, but the whole environment including the data. An object-oriented programming (OOP) environment was developed, where the programmer invokes objects to perform actions on themselves. This change in environment allowed data and procedures to be encapsulated to form objects, which made possible the conversion from passive data to active data (Chorafas & Steinmann, 1993).

The hierarchical, network, and relational DBMS were struggling to handle this new complex data. To make things worse, these DBMS's did not support distributed data, which caused developers to produce applications that included the functionality to access remote data. These problems resulted in the creation of an object-oriented database management system (OODBMS). OODBMS combine OO features such as encapsulation, class inheritance, and polymorphism with standard database features to produce a DBMS environment that can support active data.

Two generations of OODBs have been developed over this time. The first generation OODB is known as persistent storage managers for object-oriented programming languages (OOPLs). This type of OODB is a file system that automatically stores all objects developed in an OOPL, and at program termination automatically retrieves all stored objects that are requested by another

program. This OODB includes some standard database features such as a simple query language, concurrency control, and transaction management. The second generation OODB are OODBs with full support for ANSI SQL. This is the relational model community's response to OO, which resulted in the extended relational model. Unlike the first generation OODBs, this OODB is fully compatible with relational database systems, and include standard relational database features together with core OO concepts.

The advantages of OODBs are inherited by OO concepts, such as support for complex objects, reusability of classes, robustness and faster application development time. OO approaches are however an evolution, not a revolution. OODBMS face strong opposition from the established DBMSs (mainly the RDBMS), and even though OODBMSs support complex structures, many environments (especially smaller companies) do not require these added advantages, and are content with the RDBMS (Kim, 1995).

3.2.2.5 Multimedia Databases

The concept of a multimedia database system is fairly new, and most work has been performed on the theoretical level. A number of multimedia systems have been developed in the commercial market, but these systems have been developed mainly on a case-by-case basis, and would not be suitable for large-scale development.

(a) Terminology and multimedia database design issues

Subrahmanian & Jajodia (1996) define the basic concepts used to characterise a multimedia system. Firstly, a *media-instance* is defined. A media-instance consists of a collection of information that is represented by using a specific storage technique in a storage repository, together with functions and/or relations that express certain aspects, features and/or properties of the media-instance. For example, a media-instance could consist of a set of video segments represented using a number of consecutive bitmaps that are stored on a CD, including functions for parsing through the video frames.

Using the above definition of a media-instance, we can now define a *multimedia system* as a set of media-instances. A multimedia system at any particular time *t* therefore consists of the union of the states of all the different media-instances (belonging to that multimedia system) at time *t*. For example, a multimedia system

at time *t* could consist of a snapshot of an English soccer game on video, the corresponding Italian audio translation of the commentating, and textual game information (e.g. the current score), all at time *t*.

A *logical query language* is also developed for the multimedia database system, which is used to retrieve multimedia data such as feature or scene-specific information. A distinct *indexing structure* is used to store multimedia systems, which should allow queries to be efficiently executed (e.g. using object-tables and frames which will be discussed later).

A *media-event* is used to define the concept of a *media presentation*. A media-event is a representation of the global state of the different media at a certain point in time. As an example, a news broadcast at time *t* may include a video segment, audio segment and textual information describing a NATO news conference that took place live at time *t*. This therefore is a media-event at time *t* with the video state being “NATO News conference”, the audio state being “NATO news conference speech”, and the textual state being “NATO news conference textual info”. A media presentation is defined as a sequence of media-events, and therefore shows how a set of media-instances change over time.

A media-instance can be formally defined as a 7-tuple,

$$\mathbf{mi} = (\mathbf{ST}, \mathbf{fe}, \lambda, \Re, F, \mathbf{Var}_1, \mathbf{Var}_2)$$

where:

ST is a set of states that take on a certain form used to store information. This set can for example be the set of all bitmaps with similar dimensions.

fe is a set of features, which are parts of information that can be useful and/or unique within a given state. The features of interest within a given state may for example be specific people or even more generally, military officers of different rank.

λ is a map that describes which features belong to a given state. To illustrate, a given state *s* that contains the features general and sergeant which will be recorded as $\lambda(s)$ = {general, sergeant}.

$\Re$ is a set of inter-state relations, which describe differences between states, such as the differences in the features between two given states. For example, the inter-state relation add_general(s_1,s_2) could mean that the only difference (in features) between states s_1 and s_2 is that s_2 contains one more feature, namely a general.

F is a set of feature-state relations, which represent the relationships between features within a given state, such as the difference in the positions between specified features.

> **Var**$_1$ is a set of state variables ranging over states, and **Var**$_2$ is a set of feature variables ranging over features. A state-term of a media-instance **mi** is any element of (**ST** $\cup$ **Var**$_1$), and a feature-term of media-instance **mi** is any element of (**fe** $\cup$ **Var**$_2$) (Subrahmanian & Jajodia, 1996).

Subrahmanian & Jajodia (1996) developed a theoretical frame-based query language which can be used to perform queries on a multimedia system of the form **MMS** = {M_1, ..., M_n} where:

$$M_i = (\mathbf{ST}^i, \mathbf{fe}^i, \lambda^i, \Re^i, F^i, \mathbf{Var}_1{}^i, \mathbf{Var}_2{}^i) \text{ for } 1 \leq i \leq n$$

The query language developed consists of constant symbols, function symbols, variable symbols, and predicate symbols. Using this query language, we could for example perform the following query:

(∃X,P,S)outranks(sergeantdoe,X,P,S)&with(sergeantdoe,X,S)&frametype(image),

that asks whether there exists an image containing Sergeant Doe and an officer who outranks Sergeant Doe (in the same image).

Subrahmanian & Jajodia (1996) explain two structures that can be used to store state and feature information in a multimedia database. These structures allow queries to be performed efficiently on the multimedia database. The first structure is called an o*bject-table*, and is used to relate state information to feature information. It contains references and other information concerning which states contain a specified feature. Each feature therefore contains an entry in the object-table with a reference to a list of states containing that particular feature. If the features are uniquely named and stored in some logical order (e.g. alphabetically), efficient searches can be made. The second structure is the *frame* data structure*,* which is closely related to the object-table. The frame data structure contains a list of all the states in the multimedia system, with each state referencing a list of nodes pointing to each of its features (contained within that state) in the object table. Using these structures, it is simple and efficient to add, delete, and update state and feature information in the multimedia database.

As defined earlier, a media-event represents the global state of the different media at a given point in time. A media presentation has as its aim to collect a requested sequence of media-events, with each media-event ensuring synchronisation of the different media states. The media-events necessary to realise a media presentation are generated through query processing, while constraint-solving is used to ensure synchronisation.

Using the above definitions, in a multimedia system:

MMS = {$M_1, M_2, \ldots, M_n$} where $M_i = (\mathbf{ST}^i, \mathbf{fe}^i, \lambda^i, \Re^i, F^i, \mathbf{Var}_1{}^i, \mathbf{Var}_2{}^i)$ for $1 \leq i \leq n$,

we can specify a media-event as an n-tuple (s_1, s_2, ... , s_n) where $s_i \in$ **ST**$_i$. In other words, we can assemble a media-event by choosing a state s_i from each medium M_i. Further, we define a *multimedia specification* as being a sequence of queries Q_1, Q_2, ... , performed on the multimedia system **MMS**, such that each query results in a set of "acceptable" media-events i.e. the media-events that best suite the query. Using the definition of a multimedia specification, we can now define a multimedia presentation as a sequence of media-events **mev**$_1$, ... **mev**$_i$, ... such that media-event **mev**$_i$ satisfies the query Q_i. The queries can therefore be specified to generate the media-events necessary to make up a multimedia presentation.

One important aspect concerning multimedia databases that must be considered is the synchronisation of the media-events. This is achieved by synchronising the queries that result in the correct sequence of media-events. We cannot assume that the output of query Q_1 will be ready at time t_1, query Q_2 at time t_2, ... , unless time constraints are set in place. A deadline for the entire sequence of media-events, as well as individual query lower-bound and upper-bound times must be estimated. Synchronisation is then ensured by scheduling the actual start and end times of each media-event (using constraint satisfaction), and ensuring that these times are satisfied by the lower and upper-bound estimates (Subrahmanian & Jajodia, 1996).

Commonly, the concept of a *Binary Large Object* (BLOB) is used to combine multimedia data with traditional data. A BLOB is the representation of multimedia data as an untyped, typically long and variably sized attribute in an existing DBMS, such as the relational DBMS. Using this method, the multimedia data is managed as though it were traditional data. When requested, the BLOB is transferred from the DBMS to the requesting application program as blocks of data. It is the application program's responsibility to assemble the BLOB data into the correct format (and perform any further processing). A relational database can be used to associate the BLOB attribute with other data, allowing traditional data to co-exist with the multimedia data, and thus enjoying the benefits of the RDBMS. The main disadvantages with this technique is that the DBMS cannot modify portions of the BLOB (because it is treated as a single untyped attribute), and it is difficult to apply optimisation techniques for the delivery of the data (Nwosu, Thuraisingham & Berra, 1996).

(b) The pro's and con's of a multimedia DBMS

The issues concerning the development of a multimedia DBMS have been debated at the 1995 International Workshop on Multimedia DBMSs. The researchers that are opposed to the development of a multimedia DBMS argue that there is no real need to define or develop it. This group of researchers believe that current DBMSs are able to manage computational and storage issues concerning traditional data *and* multimedia data. These researchers agree that, instead of developing a complete multimedia DBMS, a strong front-end or back-end presentation manager combined with a current DBMS, would suffice. Their main argument is that presentation always has been and should remain external to the DBMS. These researchers do however agree that it is becoming necessary for the DBMS to handle multimedia meta-data.

The opposing group of researchers that agree with the development of a multimedia DBMS state that presentation is a very important part of multimedia data processing. In their opinion, early designers of DBMSs found no need for presentation with respect to traditional data processing because there wasn't very much to present. The introduction of multimedia data processing brought on the need for a DBMS that could handle multimedia data from input to output. This group continued their argument by saying that presentation is an integral part of multimedia data processing, and therefore better efficiency will be achieved if the presentation manager forms part of the multimedia DBMS. It was also suggested that formal development of a data model for multimedia data be made, which would include storage and transfer properties of multimedia data (which should improve the development of a multimedia DBMS) (Nwosu, Thuraisingham & Berra, 1996).

3.3 Conclusion

My work on secure multimedia databases described later in this dissertation is not dependent on a specific DBMS. My aim is to present models that mainly ensure the authentication and authorisation of multimedia data stored in a repository, depending on the content of the multimedia data (not just the complete image, video, etc.). The role of the DBMS used in my models is not to store the raw multimedia data (which is stored externally e.g. on a secure media server), but to store data relating to the security of the multimedia content. The database therefore does not need to manage the entire storage and distribution of requested multimedia content; the DBMS does however handle multimedia meta-

data. The BLOB technique is also not used to store an entire multimedia file – it's concept is however used (to a smaller extent) to store security information associated with the multimedia data e.g. region information. Front-end and back-end managers are used to manage the requests, assembling, and transfer of multimedia data. The relational DBMS was chosen to explain the models provided in this dissertation. Other DBMSs can however be used in place of the RDBMS without requiring major changes to the given models.