PHYSICAL DATABASE DESIGN APPLIED IN THE RELATIONAL DATABASE MODEL

O.C. BAXTER B.Ing.

Mini-dissertation submitted in partial fulfilment of the requirements for the degree Masters in Engineering (Electronic) / Magister Scientiae (Engineering Sciences) at North-West University

Supervisor: Prof. W.C. Venter

2003

Potchefstroom

1 **INTRODUCTION** 5

2 **LITERATURE STUDY** 7

2.1 DEFINITIONS AND TERMINOLOGY 7

2.2 EMPIRICAL LAWS WITHIN COMPUTER SYSTEMS 12

2.3 DATABASE TYPES 13

2.3.1 Operational databases *13*

2.3.2 Analytical databases *13*

2.4 PRIMARY DATABASE MODELS 14

2.4.1 Hierarchical Model *14*

2.4.2 The network model *16*

2.4.3 The Relational Database Model *19*

2.4.4 The object-oriented model *25*

2.5 DATABASE DESIGN PRINCIPLES 28

2.5.1 Elements of database design *28*

2.5.2 Terminology *30*

2.5.3 Entity-relationship diagrams *34*

2.5.4 The selection of keys *36*

2.5.4.1 Candidate keys 37

2.5.4.2 Primary key 37

2.5.4.3 Alternate keys 38

2.5.4.4 Non-keys and indexes 38

2.5.5 Data integrity *39*

3 DESIGN **41**

3.1 DISK ACCESS AND STORAGE 41

3.1.1 Accessing data *43*

3.1.2 Control and configuration *45*

3.1.3 Reading data *47*

3.1.4 The writing operation *49*

3.2 USE OF MEMORY 54

3.2.1 Indexes into memory *54*

3.2.2 Data into memory *55*

3.3 CONCURRENT USAGE 57

3.3.1 Random file access *57*

3.3.2 Locking table *57*

3.3.3 Transactions *59*

3.3.4 Business process *60*

3.3.5 Other issues *60*

3.3.5.1 "Dirty reads" 60

3.3.5.2 Choice of programming language 61

3.3.5.3 Further possible additions or enhancements 62

4 IMPLEMENTATION **63**

4.1 DESCRIPTION OF SYSTEM 63

4.2 ASSUMPTIONS 64

4.3 THE NATURE OF THE INPUT 67

4.4 SYSTEM OVERVIEW 70

4.5 PERL 71

4.5.1 Primary functions and modules used 72

4.5.2 Custom functions in detail 73

4.5.2.1 Read 73

4.5.2.2 Insert 73

4.5.2.3 Update 74

4.5.2.4 Delete 74

4.5.2.5 Cascaded delete/deletion rules 74

4.5.2.6 Relation maintenance 74

5 RESULTS 75

5.1 DATABASE FUNCTIONS 75

5.2 SHORTCOMINGS 76

5.3 ENHANCEMENTS/IMPROVEMENTS 77

6 CONCLUSION 79

7 REFERENCES/BIBLIOGRAPHY 81

A. APPENDIX A: PROGRAM OUTPUT 84

A.1 LOGIN 84

A.2 READING RECORDS 85

A.3 INSERT A NEW RECORD 87

A.3 UPDATE 90

A.4 DELETE 92

B. APPENDIX B: PROGRAM CODE 96

1 Introduction

Databases are becoming a progressively integrated part of our information-driven society. The effective operation of almost all businesses depends on information systems that make use of database systems which provide data integrity, availability and security in varying degrees. The age old saying – "Scientiae potenta est" / "Knowledge is power" - has become increasingly applicable where timely and structured data (information) form the basis of business decision making.

Various database systems are available to the consumer (mainly in the relational database market since this is the model most applications use due to its popularity), such as Microsoft SQL Server, Oracle, DB2, MySQL, SAPDB to name but a few. With the database models these software systems use to control the logical structure of databases already standardized (for the purpose of integration), most of the finer, lower level operations are protected as intellectual property or are obscure. This includes the physical interaction of the database management software with physical computer resources. An investigation in this area and how it relates to a database model is the main focus of this study.

Three empirical "laws", Moore's Law, Gate's Law and Parkinson's Law, denote statistical trends rather than scientific theory, approximating real-world tendencies rather than theoretical abstraction that dictate these trends, and as such cannot be used to predict the long term development of technology in this area. However, as society is not only governed by the limits of science / our understanding of the natural world, but also human perceptions and goals coupled with this scientific knowledge, these laws should be considered in any future strategic alignment. Also, these laws should be considered in view of the increasing demands on data storage and the performance of systems connected with this data.

In effect, these laws state that any increase in processing or storage potential is immediately consumed by data and software requirements. Again, these are not natural laws – only observations of how industry has traditionally reacted to the advancement of semi-conductor technology in terms of software. This implies that with the advancement of technology users might be able to do more (in terms of the functionality software

offers), but they won't necessarily be able to do it faster. This is where one can further look at the specialized use of hardware to assist with the software functions in databases as discussed in the last chapter of this mini-dissertation.

In summary, the purpose of this mini-dissertation is as follows.

1. Firstly, to investigate the evolution of database models and concepts relating to them by citing a number of primary database models over the years, discussing the benefit and pitfalls in each.
2. Secondly, to define the simple system model building blocks of a database system and how they relate to the physical components, and then practically implementing these principles in a design of a relational database system.
3. Lastly, to use the design process to identify system nodes that allow potential for an increase in performance and briefly discuss the potential in these areas. For example, not only software but also hardware based storage design for database systems to counteract the relation between data resource consumption and processing power.

It must be noted that this study is not purporting to be an in-depth analysis of physical database design; it merely suggests possible ways in which to implement mechanisms to allow a database model to operate practically in reference to physical computer system resources. The subject of database design is multi-faceted with high degrees of complexity in various areas requiring both vast knowledge and experience to be understood completely. As such this mini-dissertation provides a fair overview which can serve as departure for specialised study in the respective areas of database design.

In view of this, the results indicate that by using some of the methods and mechanisms suggested, a usable relational database can be obtained which, to a degree, operates effectively with available system resources. Further modularisation (to allow greater flexibility) and especially memory functions can be addressed to evolve it into a productively usable database. Thus for the purpose of the design and with the assumptions made in relation to this design, the practical implementation was successful.

2 Literature study

2.1 Definitions and terminology

Please note that certain Internet sources are used, specifically referring to Techtarget.com, since the definitions provided typify the generally accepted and understood concepts, which describe the respective terms as they are used in the industry.

Data: Data is values stored within a database. These values are static until they are modified by some defined process (Hernandez, 2003:45). A singular unit or point of data is a ***datum***.

Information: Information is data that have been processed so that it is useful and meaningful to the end user. Information is dynamic due to its dependence on the data stored in the database, and due to the many possible ways in which it can be presented and processed (Hernandez, 2003:45). In other words, information is contextually sensitive depending on the perception of the interpreter (Bellinger, 2004). It is not simply a collection of data (Fleming, 1996), as without any relation between the values, the collection does not represent meaningful content.

Intelligence: The organizational intelligence/learning process is a continuous cycle of activities that include sensing the environment, developing perceptions and generating meaning through interpretation, using memory about past experience to help perception, and taking action based on the interpretations developed (Choo, 1995). Intelligence is employed to develop knowledge.

Knowledge: Beyond the relation of data, there are patterns that emerge from information that have the potential to represent knowledge. It only becomes knowledge when one is able to realise the patterns and their implications (Bellinger, 2004). An important attribute of knowledge is that, when the pattern is understood, it allows predictability (Bellinger, 2004).

Database: According to the Techtarget.com online technology dictionary (Techtarget, 2003), a database is a collection of data that is organized so that its contents can easily be accessed, managed and updated. The most prevalent type of database is the

relational database, a tabular database in which data is defined so that it can be reorganized and accessed in a number of different ways. A distributed database is one that can be dispersed or replicated among different points in a network. An object-oriented programming database is one that is congruent with the data defined in object classes and subclasses.

Databases contain aggregations of data records or files, such as sales transactions, product catalogues and inventories, and customer profiles. Typically, a database manager provides users the capabilities of controlling read/write access, specifying report generation and analysing usage. Databases and database managers are prevalent in large mainframe systems, but are also present in smaller distributed workstation and mid-range systems such as the AS/400 and on personal computers.

Structured Query Language: Structured Query Language (SQL) is a standard language for making interactive queries from and updating a database such as IBM's DB2, Microsoft's Access and SQL Server, Oracle, Sybase and Computer Associates.

Data model: A data model is mathematical formalism consisting of two parts (Ullman 1988:32):

- A notation for describing data, and
- A set of operations used to manipulate that data.

A data model is a way of organizing a collection of facts pertaining to a system under investigation.

Theoretically, data models provide a way of thinking about the world, a way of organizing the phenomena that interest people. They can be thought of as an abstract language, a collection of words along with a grammar by which one can describe a subject. By choosing a language, one pays the price of being constrained to form expressions whose words are limited to those in the language and whose sentence structure is governed by the language's grammar. One is not free to use random collections of symbols for words nor can one put the words together in any ad hoc fashion. Accordingly, data models provide set structures to allow clarity and meaning, while imposing some limitations in how it can represent or accommodate data.

A major benefit received by following a data model stems from the theoretical foundation of the model. From the theory emerges the power of analysis, the ability to extract inferences and to create deductions that emerge from the raw data.

Different models provide different conceptualisations of the world; they have different outlooks and different perspectives. There is no universally agreed upon best data model since the choice of model depends on the subject represented.

Relational database: According to Techtarget.com (Techtarget, 2003), a relational database is a collection of data items organized as a set of formally-described tables from which data can be accessed or reassembled in many different ways without having to reorganize the database tables. The relational database was introduced by E. F. Codd at IBM in 1970.

The standard user and application program interface to a relational database is SQL, as previously defined. SQL statements are used both for interactive queries from a relational database and for gathering data for reports. These queries can return either data or information, since the data can be manipulated by the query to have a particular meaning (information), or it can simply return the static value as it is contained in the database (data).

In addition to being relatively easy to create and access, a relational database has the important advantage of being easy to extend. After the original database creation, a new data category can be added without requiring that all existing applications be modified.

A relational database is a set of tables containing data fitted into predefined categories. Each table (which is sometimes called a relation) contains one or more data categories in columns. Each row contains a unique instance of data for the categories defined by the columns. For example, a typical business order entry database would include a table that describes a customer with columns for name, address, phone number, and so forth. Another table would describe an order: product, customer, date, sales price, and so forth. A user of the database could obtain a view of the database that fits the user's needs. For example, a branch office manager might like a view or report on all customers that bought products after a certain date. A financial services manager in the

same company could, from the same tables, obtain a report on accounts that need to be paid.

When creating a relational database, you can define the domain of possible values in a data column and further constraints that may apply to that data value. For example, a domain of possible customers could allow up to ten possible customer names but be constrained in one table to allow only three of these customer names to be specifiable.

The definition of a relational database results in a table of metadata or formal descriptions of the tables, columns, domains, and constraints.

Repository: A repository is a logical structure that stores and protects data. Repositories provide the following functionality:

- add (insert) data to the repository
- retrieve (find, select) data in the repository
- delete data from the repository

Some repositories allow data to be changed, or in other words to be updated. This is not strictly necessary because an update can be accomplished by retrieving a copy of the datum from the repository, updating the copy, deleting the old datum from the repository, and inserting the updated datum into physical storage where it is retained on electronic memory for later retrieval.

The main functions of repositories are:

- Security: Repositories are typically password protected with many utilising elaborate security mechanisms such as encoded data.
- Robustness: Accidental data loss is safeguarded against via the transaction mechanism in the event of power failures and system crashes.

An example of a commercially available repository is Kala (Simmel and Godard 1991).

Transaction: A transaction is a sequence of related database manipulation operations that together form a unit from the perspective of the information that it describes. Transactions have the property that, if they are interrupted before they complete, the

database will be restored to a self-consistent state, usually the one before the transaction began, to ensure data integrity. This action is known as rollback. Rollback functionality usually involves capturing the complete transaction and recording the values that are about to be changed into a temporary non-volatile storage area, before attempting to physically alter the data in the database. If an interruption should occur during this change process, the system can refer to the temporary stored information to either complete the data changes, or revert to the values prior to transaction being executed. Otherwise the values are committed to the database.

Data integrity: The condition existing when data is unchanged from its source and has not been accidentally or maliciously modified, altered, or destroyed during any operation, such as transfer, storage, and retrieval (INFOSEC, 1999).

Database Management System (DBMS): A database management system is a data repository along with a user interface providing for the manipulation and administration of a database. Commonly a DBMS is understood to be a software system, a program (or suite of programs) that is run on a computer system. A few examples of commercially available DBMSs include Gemstone, O2, Versant, Mattise, Codasyl, Sybase, Oracle, DB2, Access, and dBase. A DBMS provides many features and services missing from the limited functional characteristics repository.

DBMSs are seen to be composed of three levels of abstraction:

- physical: this is the implementation of the database in a computer. It is concerned with matters such as storage structures and access method data structures.
- conceptual: this is the expression of the database designer's model of the real world in the language of the data model.
- view: different user groups can be given access to different portions of the database, known as views.

RDBMS: A relational database management system (RDBMS) is type of DBMS that is specifically involved in the administration of a relational database, which implies programmatically adhering to the rules imposed by the relational database model. This is explained in greater detail in the following chapter.

Most commercial RDBMSs use SQL to access the database, although SQL was invented after the development of the relational model and is not necessary for its use. The leading RDBMS products are Oracle, IBM's DB2 and Microsoft's SQL Server.

Despite repeated challenges by competing technologies, as well as the claim by some experts that no current RDBMS has fully implemented relational principles, the majority of new corporate databases are still being created and managed with an RDBMS (Techtarget, 2003).

I/O: Input/Output, as it refers to the transfer of data into and out of a computer.

2.2 Empirical laws within computer systems

Three empirical "laws" should be considered in view of the increasing demands on data storage and the performance of systems connected with this data.

First, Moore's law which was the observation that the logic density of silicon integrated circuits has closely followed the curve (bits per square inch) = $2^{(t - 1962)}$ where t is time in years; that is, the amount of information storable on a given amount of silicon has roughly doubled every year since the technology was invented. This relation, first uttered in 1964 by semiconductor engineer Gordon Moore (who co-founded Intel four years later) held until the late 1970s, at which point the doubling period slowed to 18 months (Moore, 1965). It must be noted however that this is not a physical/natural law but rather one that is the result of industry investment and research – in a sense, self-fulfilling.

The second is Gate's law, which states: "The speed of software halves every 18 months." This often cited law is an ironic comment on the tendency of software bloat to outpace the every-18-month doubling in processing capacity per dollar predicted by Moore's Law.

The third is Parkinson's law of data: "Work expands so as to fill the time available for its completion" (Parkinson, 1958). This has been converted to "Data expands to fill the space available for storage", interchanging time and data in relation to the availability of resources. It simplifies to the phenomenon of more memory encouraging the use of more memory-intensive techniques. It has been observed since the mid-1980s that the memory usage of evolving systems tends to double roughly once every 18 months –

thus following the expansion of capacity as predicted by Moore's law. The laws of physics (heat dissipation in particular) lead one to believe that the latter cannot continue indefinitely as manufacturing and development costs become more prohibitive. This statement of course neglects the impact of sudden materials advancement which could alter the time-frames mentioned drastically. However, the principle remains applicable in that whatever capacity is available will be consumed.

2.3 Database types

Two types of databases are found in database management:

- operational databases and
- analytical databases.

2.3.1 Operational databases

Operational databases form part of the most critical systems in most organisations today. This kind of database is primarily employed in on-line transaction processing (OLTP) where data is dynamically modified (added, updated, deleted) on a daily basis to reflect up-to-the-minute information. These types of databases are used in industries where the data changes constantly, such as the retail, manufacturing, healthcare and publishing industries.

Typically these types of databases can be used to track near 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 reorder stock.

2.3.2 Analytical databases

Analytical databases are primarily employed in on-line analytical processing (OLAP), where historical and time-dependent data are stored to track trends and hold statistical data ranging over long periods of time. This information then serves in tactical or strategic business decisions which have to extrapolate from available data to aid

businesses or organisations to plan and act pro-actively. Contrary to the highly dynamic operational databases, these types of databases store mainly static data which very rarely change, if ever.

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. Chemical labs, geological companies, and marketing analysis firms are examples of organizations that use analytical databases.

2.4 Primary database models

The following paragraphs describe some of the more commonly known and utilised database models, indicating the advantages and disadvantages of each respective model. An understanding of these models and their evolution is required to do a practical design/implementation.

2.4.1 Hierarchical Model

Hierarchical DBMSs were popular from the late 1960s, with the introduction of IBM's Information Management System (IMS) DBMS, through to the 1970s. It was developed to model the many types of hierarchical organisations that exist in the world, as it is a natural means of clear and understandable categorisation. There is no original document describing the hierarchical model, implying that it is a social model adapted to represent data in understandable formats.

Recent examples include:

- Control Data Corporation's Multi-Access Retrieval System (MARS VI),
- IBM's Information Management System (IMS), and
- System-2000, as distributed by the SAS Institute.

The hierarchical data model organizes data in an inverted tree structure where there is a hierarchy of tables flowing from a single table/grouping, or "root". The relationship between tables within the structure is described by the term *parent/child.* The association of parent to child is a 1:M (one to many) relation, whereby a parent table can be associated with many child tables, while one child table can only be associated with

one parent. The linking of child and parent tables takes place expressly by use of a pointer or physical arrangement of records in the table.

Together with the parent/child relationship (PCR), another element in the description on the hierarchical model, is that of a *record*. A record can be defined as a collection of field values that provide information on an entity or a relationship instance. In turn, records of the same type are grouped into record types. A record type is given a name, and its structure is defined by a collection of named field or data items, where each field has a certain type, e.g. integer, character, real or string. As described in the previous paragraph, an occurrence (or instance) of the PCR type consists of one record of the parent type and many instances of the child record type.

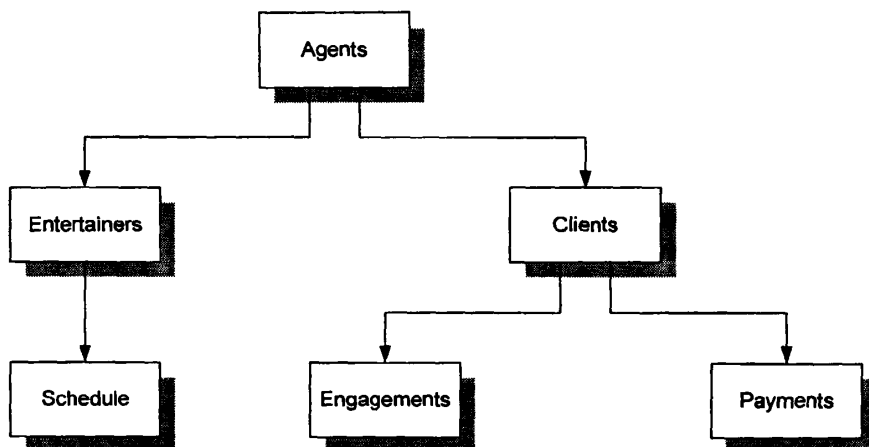


Figure 1: Diagram of a typical hierarchical database

To illustrate the hierarchical model, a simple example is presented. In figure 1 (Hernandez, 2003, p6), an agent manages several entertainers, and each entertainer has his own schedule. The Agent also maintains a number of clients whose entertainment needs have to be met by the agent. The client in turn books engagements through the agent, and makes payments to the Agent for these engagements.

Advantages:

- One noted advantage is quick data retrieval because of the structure of the tables and the explicit links that exist between these table structures.
- Another advantage with this model is that there exists implicit referential integrity, i.e. a record in a child table must be linked to an existing record in a parent table, and when a record is removed from the parent table, all colligated child table entries are removed accordingly.

Disadvantages:

- Since the hierarchical model is a tree like organisation of its data objects, it limits the type of relation that can be represented in the schema/logical structure.
- Navigation through the tree structure requires the database user to have prior knowledge of the database structure.
- A record cannot be stored in a child table if no associated table entry exists in the parent table for it. For example, in the model shown in figure 1, one cannot add a new entertainer if no agent exists for him/her.
- A M:N relationship can be handled by allowing duplication of child record instances – but this is not inherently supported by the hierarchical model. For example, if you wanted to determine which entertainer is booked by which client, data from the Clients table will have to be added to the Schedule child table, and similarly data from the Entertainers table will have to be added to the Engagements child table. However, for this to be viable, the software designer is required to ensure consistency amongst the different instances of the same record by incorporating a mechanism to synchronise the data between these instances. Otherwise, a user may enter data inconsistently, which contravenes the consistency aspect of the *database* definition.
- If a certain relationship was not noted in the initial stages of the design, or the requirement for added child tables/relationships are only identified after the design, it will result in having to redesign the whole database.

2.4.2 The network model

The popularity of the network data model coincided with the popularity of the hierarchical data model, mainly because it was developed to overcome some of the shortcomings presented by the hierarchical model. Some data was more naturally modelled with more

than one parent per child, as these types of relationships featured in daily life. This requirement is of course an inherent limit of the hierarchical model. Thus the network model was also used, which permitted the modelling of many-to-many relationships in data.

In the late 1960s, several commercial database systems emerged that relied on the network model. The most influential of these systems were the Integrated Data Store (IDS) system, which was developed in General Electric under the guidance of Charles Bachman (Bachman and Williams, 1964), and Associate PL/I (APL) (Dodd 1969). These and other systems were studied extensively by the Database Task Group (DBTG) within the Conference on Data Systems Languages (CODASYL) group that earlier set the standard for COBOL. This study resulted in the first database standard specification, called the CODASYL DBTG 1971 report (CODASYL 1971). Since then, a number of changes have been suggested to that report, including (CODASYL 1978).

The CODASYL formally defined network model is based on mathematical set theory. The basic data modelling is accomplished by using *set structures* and *nodes*, where a *node* represents a collection of records and a *set structure* provides and displays a relationship within the model. A set consists of an owner node/record type, a set structure name, and a member node/record type.

A member record type can play the role of a member node in more than one set, i.e. a record in the owner node can be related to multiple records in the member node. However, a record in the member node is only related to one record in the owner node. Due to this fact the multi-parent concept is supported within the model. An owner record type can also be a member or owner in another set. Furthermore, a record from the member node cannot exist without it being related to a record in the owner node. A set structure as defined in the Agent database as previously used, is shown in figure 2 (Hernandez, p11), with the Agents representing the owner node, and the Clients representing the Member node (a 1:M relationship).

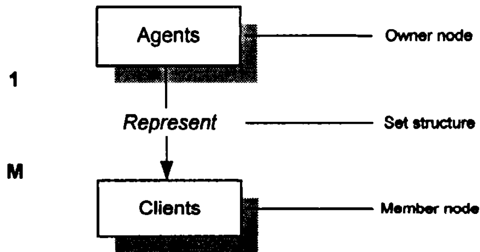


Figure 2: A simplified set structure

The data model is a simple network, and link and intersection record types (called junction records by IDMS) may exist, as well as sets between them. Thus, the complete network of relationships is represented by several pair-wise sets; in each set some (one) record type is the owner (at the tail of the network arrow) and one or more record types are members (at the head of the relationship arrow). Usually, a set defines a 1:M relationship, although 1:1 is permitted.

As an example, refer to figure 3, showing the Clients node related to the Payments node using the Make set structure. The Clients node is also related to the Engagements node through the Schedule set structure. In turn the Engagements node is connected with the Entertainers node via the Perform set structure relation.

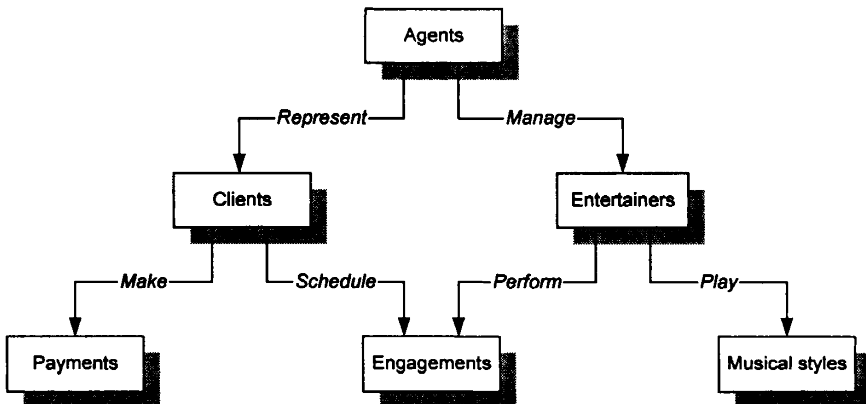


Figure 3: An example of a typical network database

(From Hernandez, 2003, p10, figure 1.3)

Advantages:

- A database user can access data from within the network database, working backwards or forwards through the sets, unlike the case with the hierarchical database where one had to start from the root table.
- Data access is fast with the possibility of more complex queries than with the hierarchical model.

Disadvantages:

- The database user has to be familiar with the set structures that define the relationships between the record collections/types, to be able to navigate the database efficiently.
- It is difficult to change the database structure without affecting the programs that interact with the database, as the set structures/relationships between data are explicitly defined.

2.4.3 The Relational Database Model

The Relational Model was defined by E. F. Codd in June of 1970 in his work "A Relational Model of Data for Large Shared Databanks" (Codd, 1970:377-387). Codd, being a mathematician, founded his work on mathematical principles in an effort to provide critical elements such as data integrity, data redundancy and a more limited dependence on the physical implementation of a database. He based this model on two areas within mathematics; that of set theory and of first-order predicate logic.

In 1979, Codd described the basics of the Relational Model, Version 2. Then in 1985, Codd published a series of articles in ComputerWorld in which he outlined the basic requirements of a relational system. He also provided a scorecard to measure the relational compliance of a DBMS. In 1989, he completed definition of the second version of the Relational Model.

This model forms the basis of most modern DBMSs that are based on the concept of a relation which is a set of tuples. A tuple is a set of facts that are related to each other in some way, not necessarily because of a natural association – they can simply be related

just by being grouped together in a set. Each fact in a tuple is a datum or data point whose value comes from a specified domain (e.g., the domain of all integers, the domain of all character strings of length 255 or less, etc.)

Mathematically the model can be represented as follows:

Given n (not necessarily distinct) sets $D_1, D_2, \ldots, D_n$ – referred to as the domains of the relation) – the Cartesian product

$$\mathcal{D} = D_1 \times D_2 \times \ldots D_n$$

of these domains is defined as the set of all (ordered) n-tuples $<d_1, d_2, \ldots d_n>$ such that

$$d_1 \in D_1, d_2 \in D_2, \ldots, d_n \in D_n;$$

Then, a mathematical relation on $D_1, D_2, \ldots, D_n$ is a subset of the Cartesian product $\mathcal{D}$:

$$\mathcal{R} \subseteq \mathcal{D} = D_1 \times D_2 \times \ldots \times D_n$$

In other words, a relation is a subset (or even a complete set) of all the possible tuples formed by the Cartesian product of the domains. Since tuples are sets (of values) and a relation is also a set (of tuples), relations are sets of sets.

Relations are naturally represented as tables, but conversely tables are not relations because relations cannot have duplicate tuples and there is no such limitation on tables. Most (if not all) commercial "relational" DBMSs violate this principle: they allow duplicate tuples. This is described further in a following paragraph.

In practice the terminology listed below in table 1 form part of the technical vernacular used when speaking in terms of relational databases.

Table 1: Terminology in relational databases

Term	**Meaning**
Relation: Two dimensional table	A relation is a collection of tuples, each of which contains values for a fixed number of attributes. Relations are sometimes referred to as flat files, because of their resemblance to an unstructured

	to as flat files, because of their resemblance to an unstructured sequence of records. Each tuple in a relation must be unique - that is, there can be no duplicates. For that purpose a so-called primary key value is used – i.e. one specific attribute value that uniquely identifies that tuple throughout the whole database.
Attribute: Table column	Other commonly used terms for attribute are 'property' and 'field.' The set of permissible values for each attribute is called the domain for that attribute. This is the smallest structure within the database and represents some or another characteristic of the subject of the table.
Tuple: Table row	A tuple is an instance of an entity or relationship or whatever is represented by the relation. This is also called a record. It is comprised of a set of fields in a table, regardless of whether these fields contain values.
Views:	Views are virtual structures that provide great flexibility in the retrieval of data, using the relations established between tuples to read certain fields/attributes from two or more tables. These views provide you with the ability to work with data from multiple tables simultaneously, restrict access to certain data and provided data integrity through what is termed as validation views.
Key	A single attribute or combination of attributes (composite key) whose values uniquely identify the tuples of the relation. That is, each row has a different value for the key attribute(s). The relational model requires that every relation have a key and that: • no two tuples may have the same key value • every tuple must have a value for the key attribute (the key fields have non-null values). The two most important types of keys are the primary and foreign keys. Primary keys are comprised of a single field or combination of fields within a record that uniquely identifies the record within the table. Foreign keys are used to establish relationships

	between tables, where the foreign key (in say, table B) is merely a copy of the primary key of the table the relationship is being established with (say, table A). I.e. the foreign key is used to relate a tuple/record from table A to one (or more) records in table B. Thus the foreign key is not required to be unique. Keys are further dealt with in section 5 of this chapter.

There are two restrictions on the relational model that are sometimes circumvented in practice:

- Duplicate tuples are not permitted. If two tuples are entered with the same value for each and every attribute, they are considered to be the same tuple. In practice this restriction is sometimes overcome by assigning unique line or tuple numbers to each entry, thus assuring that it is unique.
- No ordering of tuples within a relation is assumed. In practice, however, one method or another of ordering tuples is often used.

The physical order of these tuples (records) or attributes (fields) in a table has no effect and each record is uniquely identified by a specific field with a unique value. These characteristics allow for data to exist independently of how it is physically stored – this provides the advantage that a user does not have to know the physical location (within the structure of the database) to be able to access it, as is the case with the hierarchical and network databases.

Again referring to the simplified scenario shown by Hernandez (2003, p 14), figure 4 displays the relations between two tables, which is established implicitly by matching values in a shared field. The Clients and Agents tables have a relation created by matching Agent ID fields. Similarly, for the Engagements and Entertainers tables the relation results from the shared field Entertainer ID, thus allowing you to associate entertainers with certain engagements.

When navigating through the database the user/programmer only needs to know the relations that exist between tables to be able to navigate to a specific record. These relations can either be directly established via shared fields, or indirectly by relations constituted through fields that are shared fields in other tables. For example, one can

determine the entertainers that have performed for a specific client. This is due to the direct relation between the Clients table and the Engagements table, and the direct relation between the Entertainers and the Engagements tables. Naturally the extent of this navigation depends on the structure of the tables.

Agents

Agent ID	Agent first name	Agent last name	Date of hire	Agent phone
100	Mike	Hernandez	15/05/1995	555-1234
101	Greg	Piercy	15/10/1995	555-4321
102	Katherine	Ehrlich	01/03/1996	555-5555

Clients

Client ID	Agent ID	Client first name	Client last name	Client phone	...
9001	100	Stewart	Jameson	555-1234	...
9002	101	Shannon	McLain	555-4321	...
9003	102	Estela	Pundt	555-5555	...

Entertainers

Entertainer ID	Agent ID	Entertainer first name	Entertainer last name	...
3000	100	John	Slade	...
3001	101	Mark	Jebavy	...
3002	102	Teresa	Weiss	...

Engagements

Client ID	Entertainer ID	Engagement date	Start time	End time
9001	3000	01/04/1996	1:00 PM	3:30 PM
9002	3001	13/04/1996	9:00 PM	1:30 AM
9003	3002	02/05/1996	3:00 PM	6:00 PM

Figure 4: Examples of tables in a relational database

The relations in relational database models can exist as one-to-one, one-to-many or as many-to-many. Data is retrieved using Structured Query Language or SQL, which has been established as the standard for creating, modifying, maintaining and querying relational databases.

Advantages:

The relation database model exhibits the following advantages over the aforementioned hierarchical and network models as stipulated by Hernandez (2003, pp. 17-18) and Codd (1990, p431-439).

- Built-in integrity: Database integrity is built into the model at field, table and relationship level due to its mathematical foundation. For fields the accuracy of data is ensured, for tables it is assured that duplicate entries and missing primary keys (fields that identify tuples uniquely – explained in more detail later) don't exist, and at relationship level the validity of relationships between tables is verified.
- Data independence from database applications/Adaptability: Making a change in the structure of the tables in the network model requires programmatic making changes to all the database's queries. As a result, the network model is inflexible in the extreme. The relational model cleanly separates the logical from the physical model and this decoupling mitigates or eliminates these problems. Also, the relational model's integrity constraints are very helpful in ensuring that structural changes did not adversely affect the meaning of the database. Neither logical changes by users/designers nor physical changes to the database implementation by the database software provider will negatively affect the way in which applications operating on the application layer function. This is mainly due to the relationships that exist and the use of SQL to interact with the data.
- Concurrency/Parallelism (thus also scalability): Due to its foundation on mathematical theory and its independence from database applications, the model can be split mathematically into separate components to allow distributed processing.
- Easy data retrieval: Data can be retrieved from a certain table or groups of tables that are related directly or indirectly. Multiple views can be created on the same database.

Disadvantage:

- Speed: It has been the perception that relational database software operate slower. This was due to limitations in software implementations of this model as

well as processing power of hardware available at the inception of this particular model. Over the years the advancement of both software and hardware have allowed more efficient implementations by meeting the physical requirements of the relational model.

- Complexity: The main disadvantage of relational databases is the increase of complexity as compared with simpler flat-filed based systems and the costs involved (training and licensing) in maintaining such complex systems.
- Large objects: The relational model presents some practical problems when confronted with the storage of large objects, such as documents or picture files. Usually this is compensated for by segmenting such objects into related records. This has the pitfall that each time one wishes to view the object these segments have to be recombined to be displayed, adding processing time. Refer to BLOBs under the object-oriented model.

2.4.4 The object-oriented model

Even though relational databases have been accepted as the common business standard, it still lacks in areas serving applications for computer-aided design (CAD), geographic information systems (GIS) and multimedia storage systems. The object-oriented database, also referred to as the 'post-relational' database model, addresses this limitation of the relational model of dealing with Binary Large Objects (BLOBs). Further examples include document managing systems, email messages and directory structures.

At machine level data is represented in binary format, residing in a storage structure that is addressed or controlled by the database management software. With the previous models, the databases are designed to support small bit streams representing values expressed as numeric or small character strings. However, with large blocks of data that need to be stored, which cannot be divided or grouped into smaller objects – not logically at least – these models are limited.

This atomised data (which cannot be reduced any further), cannot easily be accommodated within the relational database. The best one can do is to store pointers to the physical locations (on some or another storage device) of these BLOBs, outside the database. The pointers allow the relational database to be searched for BLOBs, but

the BLOB itself must be manipulated by conventional file I/O methods, which impedes the performance of the database system.

Object-oriented databases provide native support for BLOBs, but there is no clear model or framework for the object-oriented database like the one Codd provided for the relational database (Codd, 1970). Under the general concept of an object-oriented database, everything is treated as an object that can be manipulated, where each object inherits characteristics of their class and have a set of behaviours (methods) and properties that can be manipulated. The hierarchical notion of classes and subclasses in the object-oriented database model replaces the relational concept of atomic data types.

Like the other models, the object model assumes that objects can conceptually be collected together into meaningful groups. These groups are called classes within the model. An object grouping is meaningful because objects of the same class must have common attributes, behaviours, and relationships with other objects.

Unlike entity sets and relations, classes do not actually hold the objects of that class and they are only conceptual entities. There is nothing in the object model that is equivalent to either an entity set or a relation. Similar to the network model, the relationships among objects are specified via a "physical" link (pointer) between objects. According to Rumbaugh et al. (1991), "The object model describes the structure of objects in a system – their identity, their relationships to other objects, their attributes, and their operations."

Object DBMSs add database functionality to object programming languages. They bring much more than persistent storage of programming language objects. Object DBMSs extend the semantics of the C++, Smalltalk and Java object programming languages to provide full-featured database programming capability, while retaining native language compatibility. A major benefit of this approach is the unification of the application and database development into a seamless data model and language environment. As a result, applications require less code, use more natural data modelling, and code bases are easier to maintain.

According to Rao (1994), "The object-oriented database (OODB) paradigm is the combination of object-oriented programming language (OOPL) systems and persistent

systems. The power of the OODB comes from the seamless treatment of both persistent data, as found in databases, and transient data, as found in executing programs."

In contrast to a relational DBMS where a complex data structure must be flattened out to fit into tables or joined together from those tables to form the in-memory structure, object DBMSs have no performance overhead to store or retrieve a web or hierarchy of interrelated objects. This one-to-one mapping of object programming language objects to database objects has two benefits over other storage approaches: it provides higher performance management of objects, and it enables better management of the complex interrelationships between objects.

This makes object DBMSs better suited to support applications such as financial portfolio risk analysis systems, telecommunications service applications, World Wide Web document structures, design and manufacturing systems, and hospital patient record systems, which have complex relationships between data. In summary, some advantages and disadvantages of the object oriented model as compared with the relational model, are highlighted below.

Advantages:

- The object model allows complex objects to be attribute domains; this is prohibited in the relational model.
- As previously stated, a higher degree of integration with object-oriented programming languages is achieved. Thus it's far more developer friendly, but consequently, less user friendly.
- There is no performance overhead with its handling of persistent & transient data.
- The object model restricts all system entities to be objects which is a more general concept than a relation (relations can be objects but not all objects are relations).
- BLOBs are inherently supported.
- The model allows better management of complex interrelationships between objects.

Disadvantages:

- There are many higher-order, non-programming query languages for the relational model. There are few equivalents for the object model (UniSQL is an example).
- There is no generally accepted formal object model. (There is however a model proposed by the Object Management Group (OMG) that is being used as the de facto standard for object-oriented DBMS).
- The object model is aimed more at programmers than at end users since the relations between objects are more obscure that with the relational model. This could impair user-friendliness in some cases.

The object-relational model, also known as the extended relational data model, extends the relational model by including a number of object-oriented elements and characteristics. As there is still much controversy amongst the proponents of the object-oriented and relational models, this model is still being refined. It has however been practically applied in the industry - specifically in the IBM Informix Dynamic Server 9.30 (Hernandez, 2003, p. 22).

2.5 Database design principles

2.5.1 Elements of database design

Database design through the creation of an entity-relationship diagram (also known as an "ERD" or data model) is an important yet sometimes overlooked part of the application development lifecycle. An accurate and up-to-date data model can serve as an important reference tool for database administrators, developers, and other members of a joint application development team. The process of creating a data model helps the team uncover additional questions to ask of end users – those who actually determine the value of the information they extract from the database.

According to Hernandez (2003, p. 33), the objectives of an effective database design are that:

- the database can support the retrieval of both required or ad hoc information (as the need arises);

- the tables within the database have suitable structure in terms of the relative fields, unique field identifiers (or so called keys) and limited redundant data;
- data integrity at a field, table and relationship level;
- the database provides relevant information within its business context;
- and finally, that it allows future growth in terms of altering or expanding the structure.

The traditionally employed method of database design consists of three steps: requirements analysis, data modelling and normalisation. The requirements analysis is conducted by studying the current business being modelled – what is needed currently and provisionally in the future. This is accomplished mainly through interviewing those people in the company that understand the business processes within it and assessing the information from these interviews and analysing current systems (if any) that serve the business process at that point in time.

Within the data-modelling stage, the database structure is modelled or mapped out using data-modelling methods such as entity-relationship diagrams, semantic object modelling or object-role modelling. Entity-relationship data modelling goes back to a paper by Peter Chen in 1976. Chen proposed a data model-diagramming scheme which would transcend thinking about physical records, by focusing on the entities and their interrelationships in the users' real world domain of interest being modelled in a database. When RDBMSs came on the scene, ERDs formed a good and natural scheme for designing relational databases. Today, ERDs and RDBMSs are widely used throughout the world in the development of computer-based information systems. The use of ERD is discussed in a subsequent paragraph.

Normalization consists of breaking down large tables into smaller ones. The aim of this process in database design is the following:

- minimize data redundancy,
- minimize data restructuring,
- minimize I/O by reduction of transaction sizes, and
- enforce referential integrity.

Within the normalization process, a number of normal forms are used to test tables against to detect duplication. The most commonly used normal forms are: First Normal Form; Second Normal Form; Third Normal Form; Fourth Normal Form; Fifth Normal Form; Boyce-Codd Normal Form; and the Domain/Key Normal form – each of which are used to detect a certain set of problems.

A normal form represents the degree to which an entity is normalized. An entity is in First Normal Form if every field contains only atomic values. An entity is in Second Normal Form if it is in First Normal Form and if non-identifying attributes are dependent on the entity's unique identifier. An entity is in Third Normal Form if it is in second normal form and there are no functional dependencies among non-key attributes. An entity is in Boyce-Codd Normal Form if it is in Third Normal Form and every determinant is a candidate key. A candidate key is a set of columns that could be chosen to be the entity's unique identifier. Keys are discussed further in subsequent paragraphs.

Hernandez incorporates normalization implicitly into his design methodology, thus circumventing the tedious re-work this usually involves using the normal form evaluation through multiple iterations.

2.5.2 Terminology

Within the entity-relationship diagrams, some terminologies become important. Firstly, an entity is a logical collection of things that are relevant to one's database, while the physical counterpart of an entity is a database table. Subsequently, an attribute is a descriptive or quantitative characteristic of an entity. The physical counterpart of an attribute is a database column (or field).

A relationship is a logical link between two entities. A relationship represents a business rule and can be expressed as a verb phrase. Most relationships between entities are of the "one-to-many" type in which one instance of the parent entity relates to many instances of the child entity. For example, the relationship between EMPLOYEE and FACTORY_LOCATION would be represented as: one FACTORY_LOCATION (parent entity) employs many EMPLOYEEs (child entity).

The second type of relationship is the "many-to-many" relationship. In a "many-to-many" relationship, many instances of one entity relate to many instances of the other entity. "Many-to-many" relationships need to be resolved in order to avoid data redundancy/duplication. "Many-to-many" relationships may be resolved by creating an intermediate entity known as a cross-reference (or XREF) entity, also called an associative table. This cross-referencing entity is made up of the primary keys from both of the two original entities/tables. Both of the two original entities become parent entities of the XREF entity. Thus, the "many-to-many" relationship becomes resolved as two "one-to-many" relationships.

For example, the "many-to-many" relationship of (many) EMPLOYEEs are assigned (many) TASKs can be resolved by creating a new entity named EMPLOYEE_TASK. This resolves the "many-to-many" relationship by creating two separate "one-to-many" relationships. The two "one-to-many" relationships are EMPLOYEE (parent entity) is assigned EMPLOYEE_TASK (child entity) and TASK (parent entity) is assigned to EMPLOYEE_TASK (child entity).

Thus, a relationship between entities exist when you can in some way associate records from the one table with the other, either through the use of keys or a linking table (referred to as the XREF entity above – whereby a third table is used to match fields between the other two). The importance of the relationship lies in its ability to bind multiple tables together, thus forming a multi-table view depending on the user's requirements. Relationships also ensure data integrity through its links with other attributes since they reduce redundant data and eliminate the possibility of duplication.

Relationships between two entities may be classified as being either "identifying" or "non-identifying". Identifying relationships exist when the primary key of the parent entity is included in the primary key of the child entity. On the other hand, a non-identifying relationship exists when the primary key of the parent entity is included in the child entity but not as part of the child entity's primary key.

In addition, non-identifying relationships may be further classified as being either "mandatory" or "optional". A mandatory non-identifying relationship exists when the relating value in the child table cannot be null. On the other hand, an optional non-

identifying relationship exists when the value in the child table can be null (thus no parent records are required for entering a record in the child table).

Mandatory or optional classifications are also termed "types of participation". Further the "degree of participation" can place the requirement on the number (minimum and maximum) of occurrences of records within a child table. For example, if you can relate at least one, but no more than five entries in a child table to the parent table for the relation, then the degree of participation is 1,5 (minimum on the left, maximum on the right).

The type of relationship, otherwise known as cardinality, helps determine the nature of the relationship between the child entity and the parent entity. The cardinality of a relationship can be ascertained by determining the number of instances of the child entity that relate to each instance in the parent entity. There are four types of cardinality:

1. one to zero or more (common cardinality),
2. one to one or more (P cardinality),
3. one to zero or one (Z cardinality),
4. one to exactly N (N cardinality), and
5. many to many (through a combination of the above).

Keys are of great importance in establishing relationships and maintaining integrity within the database. The primary key is an attribute (or combination of attributes) that uniquely identify each instance of an entity. A primary key cannot be null and the value assigned to a primary key should not change over time. A primary key also needs to be efficient in terms of storage and processing capacity.

For example, a primary key that is associated with an INTEGER data type will be more efficient than one that is associated with a CHAR data type. Primary keys should also be non-intelligent; that is, their values should be assigned arbitrarily without any hidden meaning. Sometimes none of the attributes of an entity are sufficient to meet the criteria of an effective primary key. In this case the database designer is best served by creating an "artificial" primary key.

A "foreign key" exists when the primary key of a parent entity exists in a child entity. A foreign key requires that values must be present in the parent entity before like values

may be inserted in the child entity. The concept of maintaining foreign keys is known as "referential integrity".

Fields or attributes as per relational database theory, are the smallest structures in the database. A field specification, called a "domain", represents the elements of a field, which can be broken up into three types: general, physical and logical (Hernandez, 2003, p.70).

- General elements contain fundamental information about a field, e.g. the field name, the description and the parent table.
- Physical elements indicate how a field is constructed and displayed, e.g. the data type, length of the values, and the display format (decimals and so forth).
- Logical elements identify the values stored within a field, such as whether it is a required value, the range of allowed values and a default value.

An effective/well designed database contains only one value in a field, and the field name is sufficiently descriptive of the value it holds. Three typical poor field value types are listed below (Hernandez, 2003, p. 55).

1. Multipart fields are fields that contain two or more distinct items in their value. For example, in a table that contains employee information, combining a street and town name/area code into one field called ADDRESS makes it difficult to retrieve information on employees that reside in a certain area. This impedes the flexibility of the database in that the actual stored data limits the information that can be retrieved from it.
2. Multi-valued fields are fields which contain multiple instances of the same type of value. For example, in a table that describes an organisational structure, one has a field for a manager's name, say MANAGER, and another field listing *all* the employee numbers (integer values) that report to this manager. This also greatly impedes the extraction of valuable information, requiring some form of post-data-retrieval processing.
3. Calculated fields are those fields that comprise of the result of some form of character manipulation (e.g. concatenation) or mathematical calculation. For example, in a table that contains stock information – one field the cost per item (UNITCOST) and another the amount of items in stock (AMOUNT). Then a

calculated field would be a field called VALUE, containing the value of AMOUNT multiplied by the UNITCOST. Especially where field values are volatile, a change in either the unit cost or amount of items in stock would necessitate updating the calculated field as well, thus at least doubling the time and resources that would've been required otherwise (two updates instead of one + processing time for calculation).

A null represents an unknown or missing value. This does not equate to the value zero (0), an empty string or one or more spaces, since all of these can have contextual meaning when evaluated by a language such as SQL. In situations of human error (with missing input) or where certain field values are unknown, nulls can be utilised as place holders until the data can be entered correctly. This does however present problems when doing mathematical calculations such as summation or aggregation, if the use of nulls are not carefully considered or implemented.

2.5.3 Entity-relationship diagrams

Peter Chen's original paper, "The Entity-Relationship Model - toward a unified view of data", published in 1976, laid the groundwork for entity-relationship modelling. This was reinforced by a subsequent paper published in 1977, "The Entity-Relationship Approach to logical Database Design". In this mini-dissertation, the diagrammatic representation used by Kroenke (2000) and Hernandez (2003) will be used which differ slightly from the original format, but seems to be most evident in current material on the subject. However, there are some deviations that depend on the authors' preferences.

Simply put, an entity-relationship diagram is a graphical representation of an organisation's data storage requirements, created as abstractions of the real world which simplify the problem to be solved while retaining its essential features.

Entity-relationship diagrams are used to:

- identify the data that must be captured, stored and retrieved in order to support the business activities performed by an organisation; and
- identify the data required to derive and report on the performance measures that an organisation should be monitoring.

Entity-relationship diagrams have three different components:

- ENTITIES
- ATTRIBUTES
- RELATIONSHIPS

The following definitions follow from Goodland and Slater (1995). "An entity is something of significance to the system about which information is to be held". An entity/entity class (table) is diagrammatically represented by a rectangle, with the entity name in the rectangle and the entity key name below.

"An attribute is the smallest discrete component of the system information that is meaningful". Attributes are associated to entities – in other words, they have meaning in context of the entity they are related to. Attributes are listed within the entity shape, with key designations beside the relevant fields (COK – composite, AK – alternate, CAK – candidate, FK – foreign and PK - primary keys). Key definitions follow in the next section. Refer to figure 5 for an example of a typical diagram.

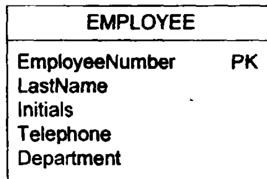


Figure 5: Entity and attribute diagram

"A relationship is an association between two entities that is important to the system." A relationship is represented by a diamond connected by a line to each of the entities involved. The maximum cardinality of the relationship is indicated by a number ("1") or letter ("n", "m") within the relationship diamond, with the "action/activity" that relates the two entities, indicated below. If the minimum cardinality is zero (indicated by a 0 or empty circle in the connecting line), occurrences of the other entity type are not obligatory. Conversely, if the minimum cardinality is not zero (indicated by a vertical line on the connecting line), occurrences of the other entity type are obligatory. This degree of participation can also be used to indicate the minimum and maximum related records required in the relation.

For example, if one refers to the ERD in figure 6, it indicates that every course requires at least one faculty member to teach it.

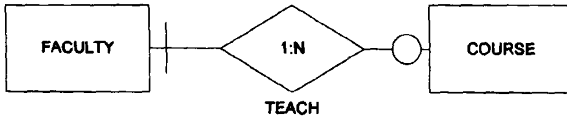


Figure 6: Example of ERD

Hernandez (2003) represents the various types of relationships diagrammatically as follows:

	Dual tables	Single tables[1]
One-to-one		
One-to-many		
Many-to-many[2]		

1. Relationships can exist between different occurrences of the same type of entity. For example, one EMPLOYEE is managed by another EMPLOYEE (both within the same entity).
2. Many-to-many relationships in an entity-relationship diagram tend to conceal areas of poor understanding. Almost always, a many-to-many relationship conceals a hidden entity. For this reason many-to-many relationships are eliminated by identifying and adding the hidden entity to the model and relating it using the one-to-many relationship type.

2.5.4 The selection of keys

Keys provide identification to each record, provide various kinds of integrity (table and relationship level) and they designate relationships between tables. Thus the selection of keys is crucial to the design of one's relational database. The key type determines the function this key fulfils in the table (candidate, primary, foreign or non-key).

2.5.4.1 *Candidate keys*

A candidate key is a field or set of fields that uniquely identifies a specific instance/occurrence as described by the table subject. Each table must contain at least one candidate key, as from these candidate keys a primary key is designated. There are certain "rules" that should be applied to fields to determine whether these fields are candidate keys, termed "Elements of candidate key" by Hernandez (2003, p. 253). According to these rules, for a field or a set of fields to qualify as a candidate field, a field or set of fields must have the following elements:

- it cannot be a multipart field;
- it must contain unique values to prevent duplication;
- it cannot contain null values;
- it cannot be used where the value of the field/set causes a breach of security or privacy;
- it cannot be used if its value is in any way optional (as it can be left null);
- it should comprise of the minimum number of fields required to provide uniqueness if it is a combination of fields to avoid redundancy;
- it must exclusively identify the value of each field in a record; and
- its values can only be changed in exceptional cases.

When no candidate key can be naturally established within the currently existing data, an artificial candidate key is created, conforming to the rules above. An artificial key can also be used when it might prove more efficient than any other candidate keys. Usually a sequential number of some sort is used in such instances – sometimes a simple incremental "counter", while at others this field has particular meaning – such as a part number or employee ID. This can then in turn be utilised as a good primary key.

2.5.4.2 *Primary key*

The primary key of any table is any candidate key of that table which the database designer arbitrarily designates as "primary". It is however suggested by Hernandez (2003, p. 262) to use single-field candidate keys instead of composite keys – constructed through a combination of fields. This key field exclusively identifies the table throughout the database structure and creates relationships between tables. The key

value, in turn, uniquely identifies a certain record throughout the entire database, preventing any duplicates.

A primary key is referred to as a "surrogate key" if the column contains no real data other than a uniqueness identifier – i.e. it's an artificial candidate key. If real data can be used as a primary key (e.g., a social security number – with cognisance of any security limitations), then it is referred to as an "intelligent key" since the actual value of the key holds meaning (Frick, 2000).

A foreign key is a set of one or more columns in any table (not necessarily a candidate key, let alone the primary key, of that table) which may hold the value(s) found in the primary key column(s) of some other table. So a primary key must exist to match the foreign key, and so establish a relationship.

2.5.4.3 Alternate keys

The alternate keys of any table are simply those candidate keys which are not currently selected as the primary key. According to Date (1995:115), "... exactly one of those candidate keys [is] chosen as the primary key [and] the remainder, if any, are then called alternate keys."

Alternate keys provide the ability to uniquely identify a certain record within a table as an alternative to using the primary key. If designated, these alternate keys can be utilized when it is determined that the originally selected primary key proves to be inefficient in terms of performance, for example. Then the database designer can make use of the designated alternate keys and select a new primary key without having to identify these fields and their requirements again. If this was not the case, entity level integrity could not be assured.

2.5.4.4 Non-keys and Indexes

Non-key fields are those fields that do not fulfil the function of candidate, primary, foreign or alternate keys. These fields merely express a characteristic of whichever subject is addressed by a specific table, and its value is a function of the primary key value (since this is a unique identifier).

A key is about uniqueness, not access, while an index is about access, and not necessarily uniqueness. In concept, an index is any set of one or more columns from a given table, sorted in any arbitrary sequence for the purpose of speeding up physical operations. Thus an index is a sorted sub-set of the data in the table to which it refers, with address pointers to the actual data row. The columns included in an index are not "keys". That is why indexes are often separated from the actual database within RDBMSs like MS SQL Server and Oracle.

2.5.5 Data Integrity

The integrity of data, which is defined as the validity, consistency and accuracy of data, is crucial in the relational (as in any) database. Integrity is the whole point of storing data within a database after all – if data cannot be trusted then the information gathered from it is rendered utterly useless. It must be noted that data integrity does not involve physical security, fault tolerance, or data preservation (backups) (Frick, 2000). Within the database design process, four types of data integrity are prevalent:

1. table-level/entity integrity;
2. field-level/domain integrity;
3. relationship-level/referential integrity; and
4. business rules.

Entity integrity ensures that each row in the table is uniquely identified. Entity integrity is most often enforced by placing a primary key constraint on a specific column (or set of columns) which imposes certain restrictions on a record in itself, thus ensuring this type of integrity. In short, these restrictions prevent duplicate records, with the primary key identifying each record uniquely, while the primary key may not be a null value.

Domain integrity defines the permissible entries for a given column by restricting the data type, format, or range of possible values (Frick, 2000). Default values can also be specified when no field values are supplied. Integrity at this level also ensures that all fields of the same type are consistent throughout the entire database, allowing clearly identified operations on these values that will not result in programmatic errors.

Relationship-level/referential integrity guarantees that the relationship between two tables is verified and that records in parent and child tables/entities are synchronized whenever data is altered or removed in either table. This is to prevent the occurrence of so-called "orphans" – records that cannot be related from the child to the parent table. Such orphans would usually result where parent entries are removed without altering or removing the child relations for it in the child entity / entities.

Business rules or user-defined integrity are limits imposed from the viewpoint of the organisation that makes use of the data. This type of integrity can include allowable field values and ranges, types and degrees of participation and the level and extent of synchronisation of data between related tables (in referential integrity).

3 Design

3.1 Disk access and storage

It is all well and good to have a database model with a well structured ERD layout to accompany it, but how does one go about physically storing this to allow effective access and operation? This is where the DBMS bridges the gap between the conceptual world of the model and real-world applicability as it structures data on a file system so that the model can be physically implemented.

One way to accomplish this, as it is employed in this study, is to make use of two methods that will define all operations within the database. Firstly, the use of direct random disk access to read or write a certain byte in a file on the storage medium. Secondly, direct disk access is combined with set record sizes, which provides one with the ability to read a specific record from the storage medium.

The functionality of accessing a specific byte is only catered for in a few programming languages, under which feature the very powerful C/C++. It is also well known that some of the leaders in RDBMS design, such as Oracle, make used of C/C++ to design their systems (while it was initially assembly language based) because it is a fairly high-level language that allows one to manipulate resources with sufficient control on a variety of operating system platforms.

The function that gives one the ability to access a specific byte in a file, is *fseek* (*seek* in Perl). As it is defined by The Open Group Base Specifications (IEEE Std 1003.1:2003):

```
#include <stdio.h>
int fseek(FILE *stream, long offset, int whence)
```

"... the *fseek*() function shall set the file-position indicator for the stream pointed to by *stream*. The new position, measured from the beginning of the file, shall be obtained by adding *offset* to the position specified by *whence*. The specified point is the beginning of the file for SEEK_SET, the current value of the file-postion indicated for SEEK_CUR (obtainable with the *ftell*() C function), or end-of-file for SEEK_END."

For this to be useful in reading or writing data in a structured manner to disk, the record structure must be defined in terms of bytes (or even bits when looked at an encoded solution). This is accomplished simply by using field definitions in reference to commonly known variable types – such as boolean, integer, char, string and float – as these variable definitions all have byte limited representations, and can thus be defined with set field length (in bytes). This in turn allows defined record lengths within a data file, allowing easy retrieval.

For example, assume an employee record consists of four fields, the first being the primary key containing the employee number, the second the employee's surname, the third his forenames, and the last field containing the employee number of his manager (which plays the role of a foreign key in establishing an organisational structure relation, for example).

We can define the employee number, which is a numeric only value, as a non-negative integer that consists of 3 bytes, allowing a field value range of 0 to 16,777,215 (2^{24} -1) – more than sufficient in a fair-sized organization for a number of years. This field definition is also used for the last field containing employee number of the employee's manager.

The second and third fields can contain alphanumeric values, so it is suitable to assign field lengths of the number of bytes equal to the number of characters these fields contain. For the sake of the example, we say both fields can have 15 characters each. Thus, diagrammatically we have what is shown in figure 7 below, with the whole record length resulting from the field definition equalling 36 bytes.

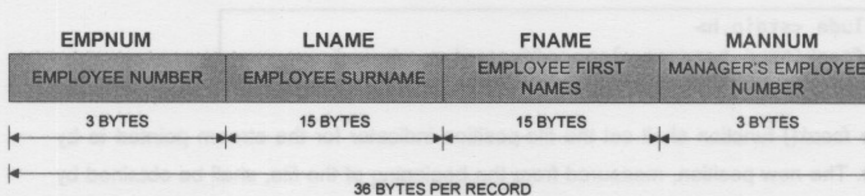


Figure 7: Field defined record length

It is possible to reduce the bytes used in text fields by defining a data type according to binary representation using look-up table methodology. This provides the benefit of a

storage saving (and also time to read/write the data), but with an increase on processing / calculation required to decode and encode the data as it is read. This is likely only to be beneficial and noticeable on large systems.

It is not necessary for the programming language used to develop the RDBMS to employ the same variable byte definitions as utilised within the RDBMS program, but it is advisable to maintain congruency to these standardised definitions to allow for ease of use and compatibility. Another benefit of using standard variable types is that the internal representations on machine level have already been optimised for processing and storage means.

Low level database operations can be categorised as primarily three activities:

- reading from the database;
- writing to the database, consisting of:
    - updating existing records and
    - inserting new records;
- and deleting data from the database.

Per definition, a program which coordinates these activities in such a fashion that data integrity is maintained is a database management system.

3.1.1 Accessing data

The main issue to address is in actual fact that of location or the address for a requested chunk of data or record, otherwise there is almost no way to gain access to it directly/by reference. One would only be able to read this record by sequentially reading the file until the primary key field for that table is found. Even then it would be a laborious (if not altogether impossible) process to extract the field values if they are not delimited by some or other reserved character, or if they are not of fixed size in terms of byte allocation.

However, remembering that within the relational model the primary key must be unique throughout the entire database for each record, one can start formulating the means to access a specific record based on this characteristic. If one could have just the key field

listed together with some manner of address indicator, a far more streamline method of accessing a record is available.

Now, to what do these address indicators point? To answer this, one must define a data block – a container of fields from the same table (thus with a fixed record size) with a fixed size as well. From documentation available it is apparent that Microsoft SQL Server and Oracle databases use pages and data block allocations respectively. Oracle offers a few data block sizes – 2, 4, 8, 16 or 32 kilobytes. MS SQL Server's pages are set at 8 KB each.

The choice of block sizes and what should be stored within them are contentious matters in themselves. There are for instance many debates surrounding the choice of larger block sizes for OLAP type systems (high volume, few users), while for the smaller volumetric activity of OLTP (relative small volume per user, many users), smaller block sizes should be used.

For the sake of simplicity, a block size of 8 KB is used in this design. The original choice of 8 KB apparently resulted from the 8 KB file system buffer in Unix, which can cause unnecessary activity of clearing and reloading this buffer if block sizes other than 8 KB are used. Since Windows uses so-called "direct I/O", this is no longer an issue.

In this design, the structure of a data block (shown in figure 8) consists of the following:

- a block number;
- the record size within the block; and
- the next block for the same record type (i.e. the same table).

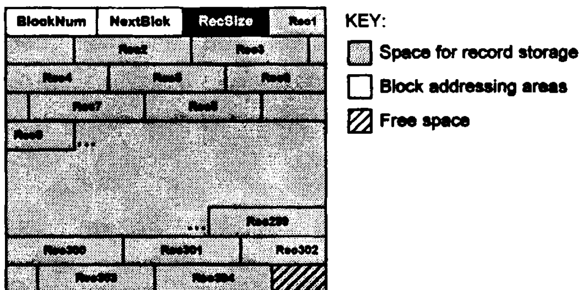


Figure 8: Data block definition

The first two blocks are important when sequential scans are conducted. The block number is the unique designation for this specific physical data block – effectively its address – with the next block indicating the record of the next data block in the sequence for this table. If one allocates about 4 bytes ($2^{32} - 1 = 4294967295$) to this attribute's value definition, a maximum of 32 terabytes of data can be addressed with an 8 KB block size. In fairness, because disk space is effectively reserved for records of the same table in a block, this amounts to areas of free space within the database, which is thus wasted from the file system point of view. This is however an acceptable overhead in relation to the function gained by this block-definition, except when a large number of tables are concerned. One can combine different tables/entities in the same blocks (this is done by Microsoft SQL Server's – so-called mixed pages – i.e. it can contain data of different table entities), but this adds to the complexity of the addressing scheme.

The record size attribute contains the record size and can in reality be omitted. It is included only in this case as a checking mechanism when reading the data records. Of course the actual size of records can be derived from their field definitions, but this will require a computation each time a record is read. The next block attribute contains the next block related to this table/entity, which can be used when reading all records from this table. After these attributes in the data block, the data records follow. Once space in this data block is depleted, a new data block is assigned and linked to the previous data block.

To accomplish the above, one requires some means of control and configuration, i.e. control over what is stored where, and in the sense of configuration, how it is stored.

3.1.2 Control and configuration

It follows as a requirement from data block allocation and field definitions (relating to tables), that the database software has some sort of system table(s)/configuration file(s) used to store the data block allocation in relation to primary keys, and the table definitions.

For the means of configuration, a central repository of all table definitions (field sizes and names together with table relations according to primary and foreign keys) are to be stored in a file or table that can be maintained by a developer who needs to create or

change the definition of a table or its relative fields. Due to the limited number of users that are connected during actual development, limited support for concurrent usage is required.

Together with this dictionary of table definitions, an index of data block allocations per table and primary keys is kept (called the primary index), which indicates the data block number and the record number within this data block enabling one to access the physical address of a record directly. This index is stored in tabular form and is preferably sorted according to primary key values (ascending/descending) to allow for faster searching through the use of algorithms as the one discussed in a subsequent paragraph. This of course places a requirement on the programming language used to build the DBMS to be able to sort according to numeric as well as text values. If this was not done it would lengthen the process of finding a specific key and related data block address. It is important to note that even though these values are stored and sorted, one still has to read these values somehow.

However, when one accesses an existing record for the first time, you might only possess a certain field value. Say in an employees table where the employee number (field EMPLOYEENUM) is the primary key, with fields like LNAME (last name) and FNAME (first name), one wishes to read a record of someone with the last name of "Smith". To physically read/search for the record relating to this field value, would require a sequential scan of all records or fields in those records contained in the table entity – thus its associated data blocks. Noted, not an entire record or all record field values are read – only one byte at a time from the field in question (which can be accessed thanks to the table definition), so that neither time nor resources are wasted to process records that have differing field values.

To alleviate this to an extent, secondary indexes have to be created as well, consisting of table names, candidate keys (to maintain addressing uniqueness) and parts of other non-unique fields, defined by choice. This choice will be dependent on the nature of the data. In our example, it is very likely that a non-unique field value for LNAME (last name) will be the search criteria for a read and due to its frequency of use it should be added as part of the secondary index. Multiple secondary indexes can be created for a specific table.

These secondary indexes have to be structurally defined and of a fixed size as is the case for the normal data records, so that one is able to accommodate all indexes together and without having to create an index to read the index. Sorting also becomes a requirement to allow for speedy access through similar algorithms described in the "Reading data" section below. Otherwise one is still able to use sequential reads with the obvious pitfall of increased disk or processing activity, which results in a definite loss of performance.

To simplify the use of indexes we can store the indexes of different tables separately, and not in one central index file. This removes the need to include the table in the index fields, but it does fragment the system architecture to a degree and makes placing the constraint of unique primary keys throughout the database, more difficult to enforce.

3.1.3 Reading data

In practice, to go about reading a specific record in reference to the primary and secondary indexes has to be accomplished by using sorting and some sort of reading algorithm. Again, depending on the nature of the data, some reading algorithms prove more efficient than others. The efficiency of these algorithms is usually depicted using the big O-notation (Cormen et al, 1990). For the purpose of this study, we employ a simple range splitting algorithm.

First let us consider the case where no reading algorithm is used. For example, let's say there are 10,000 records in a database – thus 10,000 primary keys are stored with their respective data block allocations within the primary index. In the best case for sequential reading/scanning the DBMS only needs a single read to find the primary key record in the index, at worst the DBMS needs 10,000 reads. Furthermore, the number of reads is dependent on the first selection (if the record required is at position 133, starting the sequential read at position 1 will reach the record far quicker than one starting at 10,000 going downwards). By using a splitting algorithm such as range division, a result, independent of the starting position, can be reached far more quickly – provided that the index is ordered.

Looking at the instance where the primary key value is known (if it is a record that has already been read once or one of which the primary key is directly known – for instance

an employee number): say the key value is located at position 133 in the index of a collection of 10,000 primary keys. Use the algorithm

(TopPosition – BottomPosition)/2 + BottomPosition = SplitPosition

with initial values of TopPosition = maximum number of records (in this case 10,000), and BottomPosition = minimum number of records (in this case 1). With the key values sorted in a descending order (thus the primary key value for the record at position 10,000 < than that of the record at position 1), if the primary key value at SplitPosition < than the primary key value required, TopPosition = SplitPosition, otherwise BottomPosition = Splitposition. Once TopPosition – BottomPosition = 2, the required position is equal to (TopPosition – 1) or (BottomPosition + 1). To arrive at the primary key value at position 133, 13 iterations (and thus 13 reads) are required. Note that the exceptions are position 1 and 10,000 – the initial Top and Bottom positions – which have to be checked initially before starting the iteration process.

Similarly, for an initial read of a non-primary key field (thus the primary key is not known), we can use the arranged secondary index to search for values with the same algorithm as above. However, a subset of the main set has to be created by searching through the index and demarcating the applicable fields. This subset is first formed from the table name (which is known by way of the application used to retrieve data, through its programming). This will simply affect the starting range values for the algorithm and not the algorithm itself.

Where multiple records are returned from a read query on a table – for example for the surname "Smith" in the employees table, the data is read into a temporary table (effectively an array) with the same structure as the original for the fields concerned. This data is then manipulated as required for viewing purposes.

A simple process flow for the read activity of indexed values is shown in figure 9. This is described as: field value/search term entered → read table definition for fields & relations → read index structure for table in question (i.e. which fields make up the index) → scan index in intervals determined by index structure → read block address → move to block address → read data → return data to front end.

The process only differs between the primary and secondary indexes in that the ordering of values is handled differently. To reiterate, the primary index has the actual primary keys sorted in ascending or descending values, whilst the secondary index contains the ordered values of the field in which a value was entered by the user. Where the field name is not an indexed value, a sequential scan through all fields as referenced by the primary index is done.

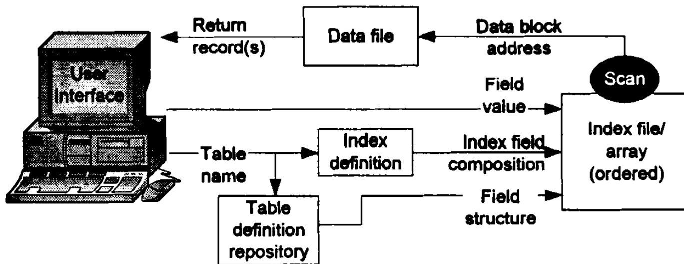


Figure 9: Basic read for indexed values

3.1.4 The writing operation

The action of writing data to the database can be divided into basically two activities – 1) updating existing data and 2) inserting new data. It is not necessarily required to define updating as a separate operation, as it can be accomplished by reading the data into memory, changing the necessary fields, deleting the record as it exists in the database and then inserting the record from memory into the database as a "new" record.

For the purpose of this study, updates are made directly to records since the field lengths are specified – and can accordingly be physically overwritten on the disk storage level. Within this study as well, deletion takes the form of an update whereby a special character is written into all record fields/simply the key field to indicate it as a deleted record. When a new record is added, it will be pointed to this deleted record instead of free space within a data block, so that the minimum amount of storage space is wasted. In the instance where no deleted records exist, the record will simply be added to a new data block. The reason behind this is to prevent a reshuffling of records whenever a deletion takes place, which would otherwise amount in unnecessary disk I/O.

When simply updating a record (not in the case where deletion occurs), there is no need to update the primary index table, which is set to point to the primary key and address block association. In the event that the primary key is or needs to be changed, a new record has to be created and the old, redundant record should be deleted. This is simply a design requirement that should be enforced through the DBMS.

However, updates of field values that might impact secondary indexes will result in an adjustment of secondary indexes, necessitating the re-ordering of these indexes with each update to keep them sequentially ordered. Similarly, in the event of a deletion, primary indexes are affected. Thus in the case of a deletion, both primary and secondary indexes have to be reorganised.

When new data is inserted, the current primary index and relational dependencies are read from the data dictionary (containing all table definitions), and the records created accordingly. Since it's reasonable to assume that artificial primary keys to be assigned to new records will be of a lower (or higher depending on the choice of the developer – consistency is essential) order/value than those prior to it, its primary index reference simply has to be appended to the primary index, with no re-ordering needed. For the secondary index(es), some reordering could be required.

To utilise space occupied by deleted records, new records are written to deleted areas in data blocks by keeping a deleted record reference. Figure 10 diagrammatically represents the process as described above, with the "Report" being the status of the insert – successful or not.

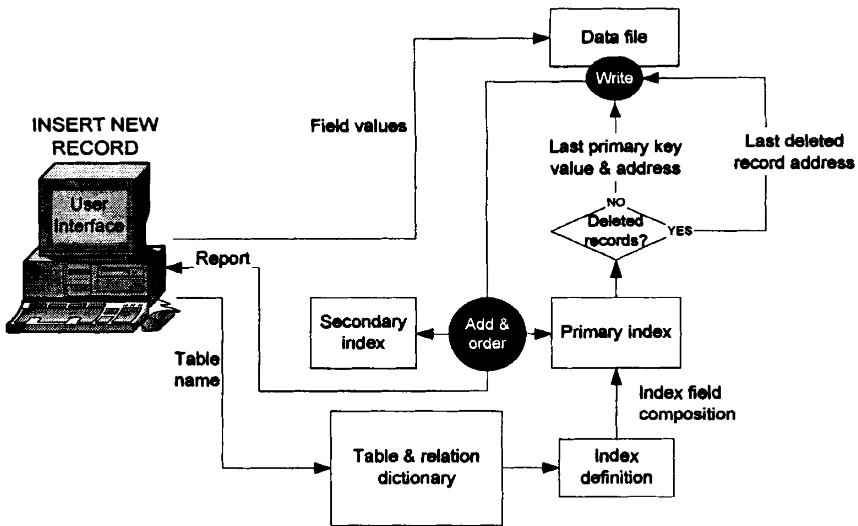


Figure 10: Inserting a new record

In reference to figure 10, the process of reading the table and field definition actually takes place, from a programmatic point of view, prior to the process of the data being inserted into the database as a new record. This is because the programmatic interface must posses the record structure so that it can present a suitable input screen for this new record. The same applies to the process of updating a record where the record in question is first read (thus the DBMS is "aware" of how data is structured within the record in terms of bytes) and displayed to the user, after which the values are changed and subsequently updated. It is shown simply for the sake of completeness even though it is not strictly part of the process.

The update process itself is very similar to the insert described in the foregoing figure. The only difference is the requirement for reordering the secondary indexes, as well as not taking into account previously deleted records. However, with the deletion process a loop is added to this process to keep track of deleted records, as shown in figure 11. One can either add another index specifically for keeping track of deleted records, or simply have a pointer in each deleted record pointing to the next data block/record address of a deleted record.

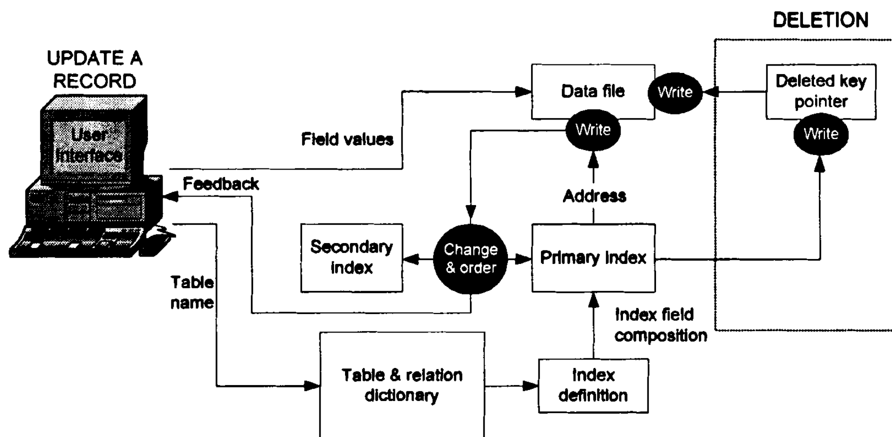


Figure 11: Update & delete

Good practice and relational consistency when inserting or updating data impose certain restrictions to prevent incorrect data entry. This also matters when defining default values in case no values are provided for certain fields when creating new records. It might be a requirement to enter data initially with some field values optional, allowing these values to be edited later on. This depends entirely on the business rules that apply to this data and as such the restrictions need to be structured around this. There may also be a restriction on whether a field value is editable after it has been created, or whether it is only editable at the time of creation.

Even though deletion in this case is formed through an update method, there are a few deletion rules to take into account, which require certain actions in the RDBMS (Hernandez, 2003:382):

1. Deny – the RDBMS does not delete data in the parent table but just marks it as an inactive record.
2. Restrict – the RDBMS will not delete a record in a parent table if any child records, still related to this table, exist.
3. Cascaded – the RDBMS will delete the parent record, as well as all related child records. More on this follows in the next paragraph.
4. Nullify – the RDBMS deletes the parent record while nullifying the child foreign key relationships.
5. Set default – this update rule requires the deletion of the parent table while setting all the foreign key values of related records in child tables to a default value – provided a default field value is defined.

Cascaded deletions take into account any "parent" data record that might have "child" tables formed by primary and foreign key relationships that will require the existence of a record in the parent table for a record to exist in the child table. For example, say for an employee data is entered of his/her dependants in a dependants table. If the employee is dismissed or quits and his/her record is deleted from the employee data table (EMPLOYEE), it would be required that any relevant data such as his/her information of dependants in the DEPEND table, also be deleted. It can then happen that the child table serves as a parent table for another child table. If there was not such a restriction, one could end with so-called "orphaned" records – losing referential integrity within the database. These dependencies are maintained within the table dictionary which contains all the relevant information of how relationships exist between tables within the database.

To thus accomplish a cascaded delete to prevent any orphan records from remaining (note, not all tables require this – it depends on the logical relations to other tables), one starts from the "bottom" of the relation, working upwards to the parent table. This would in effect require a loop in the deletion process for each child record related to the parent record. Cascaded deletes are not necessarily implied with mandatory table participations, e.g. if you have an ORDERS table in which there is mandatory participation with the EMPLOYEE table, you might not want to delete all orders related to the specific employee if the employee is removed from the database.

3.2 Use of memory

Due to the physical differences between storage on disk and storage in memory (RAM), access to memory is much faster than access to disk – e.g. access time for memory is measured in nanoseconds while disk access time is measured in milliseconds. However, due to the physical attributes of these storage mediums, faster access time from RAM comes at the price of volatility and capacity. Subsequently, for the sake of data integrity, a balance between these mediums should be maintained within a database management system.

For this reason, only certain kinds of data are really suitable for temporary storage in memory. These are mainly data that require extensive manipulation which would result in high disk activity or data that remain largely unchanged that can remain in memory because it is consistent with the state of the data on disk. In the case of the latter, some mechanisms need to be in place to "refresh" the memory buffer which is discussed in subsequent paragraphs.

3.2.1 Indexes into memory

In the scope of this design, a large amount of data is moved when indexes are re-ordered. Doing this re-ordering on disk would result in intensive disk activity, affecting general system performance considerably, especially when scaled for larger applications. The reasoning behind this can best be considered when looking at the contrary method – that of re-ordering the data on disk.

Re-ordering a secondary index on disk when a new record is added would require reading the entire index table into an array (i.e. memory), or doing so record by record, or a combination of the two. The combination can consist of first scanning through the index record by record to the point where the new index record should be inserted. Then, taking the subsequent set of index records that should then follow the newly inserted index, and writing this to memory, one can overwrite the record following immediately after the point of insertion (as this record is in memory with all following). Then the program writes the data back from memory to disk, following the inserted record. Figure 12 below illustrates the process.

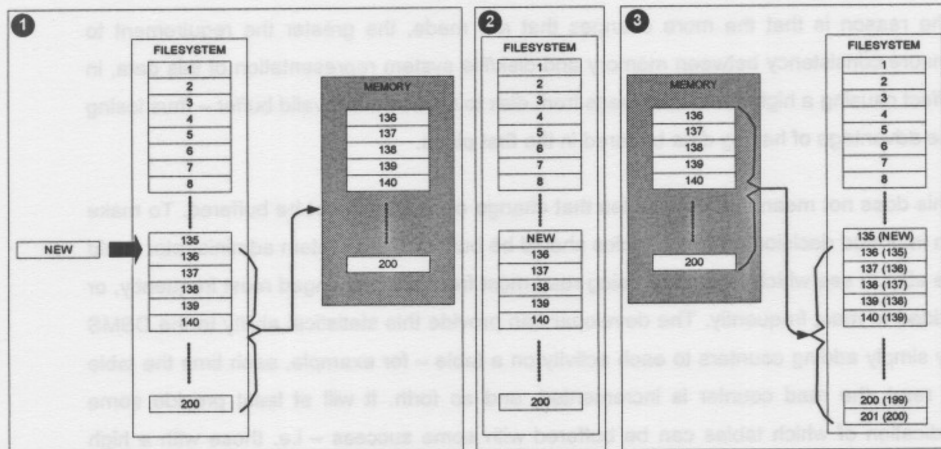


Figure 12: Record by record, disk to memory to disk

As the above indicates, the program reads from disk to memory (1) and then writes back to disk once more (3), after inserting the new record (2) and subsequent ones (3). One can imagine the implication on performance if this is done record by record. On the other hand, if re-ordering had been done in memory, the second write to disk would've been unnecessary. However, at some point in time (preferably at a time where there is little database activity), the index can be written to disk for integrity purposes in case of software or power failure. With pointers (direct memory addressing) available in programming languages such as Java and C/C++, one is able to do bit/byte shift operations on a low level which adds to the speed of the operation.

Little redundancy is required for the memory storage of indexes since indexes can be regenerated from the data available in a table through a sequential scan of said table. Nevertheless, for it to remain effective in speeding up the process of data reading, the indexes should remain synchronised with the state on disk (though not necessarily at all points in time).

3.2.2 Data into memory

For the same reason that data access speeds are improved when placing indexes into memory, so too is there benefit in placing certain types of data into memory (data buffering). Typically one would want data loaded into memory that will not change often.

The reason is that the more changes that are made, the greater the requirement to ensure consistency between memory and disk/file system representation of this data, in effect causing a high number of reads from disk to refresh this invalid buffer – thus losing the advantage of having data buffered in the first place.

This does not mean that data/tables that change often should not be buffered. To make an informed decision on which tables should be buffered, the system administrator could be able to see which tables are being read most frequently, changed most frequently, or added to most frequently. The developer can provide this statistical ability to the DBMS by simply adding counters to each activity on a table – for example, each time the table is read, the read counter is incremented, and so forth. It will at least provide some indication of which tables can be buffered with some success – i.e. those with a high ratio of reads compared to changes.

Further, the statistical measure of buffer hit ratios is a common term whereby the measure indicates the ratio between the number of reads done from buffer against the number of reads from disk (caused by invalidation, as described in the next paragraph), which can form the basis for buffer optimisation. Since this kind of configuration optimisation is beyond the scope of this study, the statistical measures are not incorporated further into the design.

To allow for consistency, each time data is changed in a table that is buffered, the buffer for this table should be updated with the changes as reflected in the data stored on disk – sometimes referred to as buffer invalidation (it is invalid because it does not reflect the actual state of the data). Thus if a table that changes very often is buffered, there would be a high number of invalidations, which would force the whole table to be reloaded into memory. This could effectively be far more detrimental to the system than the case where no buffering was used at all. To provide greater flexibility in buffering, the DBMS could also allow for segmented buffering where tables are buffered in segments and not in completion, providing the DMBS with the ability to reload only that segment that has been invalidated / changed.

The process of invalidation is triggered by changes/deletions to a table that is buffered. The table dictionary also contains information on whether a table is buffered or not, along with the structure and relationship information. It is the logical place to store this

information since it's the very first place consulted by the DBMS software when undertaking any kind of operation.

Note that it is also possible to use memory mapped I/O, where a file is mapped to a region of memory, which can then be accessed like an array within the program. This is more efficient than read or write since only the areas of the file that a program actually accesses are loaded into memory. The theoretical limit for files loaded in this manner is 4 GB on a 32-bit machine.

3.3 Concurrent usage

Concurrent access to files of the database system is required to allow effective multiple user access to an application utilising the database. However, to preserve consistency, this access cannot happen without proper controlling mechanisms, part of which is to use a central lock handler, which effectively regulates the access to certain tables for writes/changes.

3.3.1 Random file access

It has already been stated that functions such as *fseek* in C/C++ allow random file access. This is an important component in allowing the DBMS the capability to address multiple areas within database files – provided this occurs in a controlled manner. The controlling mechanism takes the form of a locking table, which contains entries indicating which records in which tables are locked.

With random file access, the program can access specific areas within the file(s) used for storing the data. For this to be useful, an index indicating the address assignment of some sort must be maintained, along with structure definitions of tables and their respective fields (either directly in terms of bytes, or in terms of data types [with encoding / decoding] that indirectly translate into byte level definitions).

3.3.2 Locking table

The key to maintaining data consistency/integrity is that of central management. Creating a single point of "awareness" in terms of what activity is currently taking place

in the database is essential in ensuring all activities do not violate restraints imposed by the database model or the logical interpretations thereof. This centralised process can then coordinate all activities within the database, in keeping with all restraints.

A proposed mechanism of a central locking table consists simply of a list indicating which records/tables are being actively used in writing operations. This is mainly passive as the locking mechanism itself is regulated by what it contains – it does not actively communicate with processes that write to the database. Alternatively, using a more memory based and active approach, a lock handler process can be implemented programmatically which would then communicate with the various write processes active at any given time. In the latter approach, a "lock table" is still used, but it's stored in memory and the lock handler process acts as go-between for the writing process and the locking table. In the prior method, each writing process effectively contains the "intelligence" to address and interpret the information presented by the locking table.

There can be decided to make use of row locking – thus just the affected row is unavailable for further processing – or table locking, which excludes the whole table from any write activity – other than that taking place at present. Once the relevant process that is active on the record or table completes its activity, the lock entry is released/removed from this locking table. Normally locking only takes place on record level, with the further possibility of extending it to field level. However, some kind of notification is required to indicate to a user that someone else is busy making a change to the relevant record – possibly negating the changes he/she had made.

One obvious problem is the loss of parallelism – restricting access to the database to one single point or process, instead of allowing multiple areas of access, which provides greater speed. This can be circumvented by creating a locking mechanism for a specified area within the database which controls access to that area alone. However, this might not balance the actual activity in certain areas, unless table usage is known. With table usage available, these "hot spots" of activity can be distributed between different locking processes.

Furthermore, the lock handling mechanism itself can operate in various ways – either directing each write/change process as it attempts to access the database – allowing it to continue, or placing it in a wait state. This has the negative effect of delaying the

activity, but with the benefit that each writing process handles the data it is about to write, with the lock handler process simply controlling the queue. Or, it can gather the information for the change/write, "release" the process which was attempting to write to a locked record/table, and write the change to the database once the lock is released. This would require an internal queuing method within the lock handling process itself, but would release any resources used by the write process.

In the context of this study, the passive method of central locking is proposed to limit the complexity of the implementation, whereby locks for each record change is listed in a locking table until such time that the writing process completes, after which the lock is released / removed. This requires the writing process to incorporate lock handling – i.e. reading whether a lock is present, waiting if that is the case, and retrying at arbitrary intervals.

3.3.3 Transactions

Transactions are the logical equivalent for record changes. A transaction might encompass various write/change operations on the database in different tables, that together in a business or logical context make "sense". For instance, buying a new piece of equipment would result in an entry in a financial table to subtract the cost from current capital, while adding data to the inventory and assets tables to indicate an increase in these. Individually these changes are merely table changes, but from a transactional perspective they form a logical or business oriented unit of change, the one inter-dependent of the other, i.e. a transaction.

With the use of record locking, the option is introduced to use transactional based locking as well by grouping logically dependent changes together. It can be useful to ensure logical consistency as the transactional lock controls a change to all related entries to describe some or another business process. A further possible advantage is the logging of these transactions to a file or table, allowing erroneous changes to be reversed (rollback), or allowing one to reapply changes for instance with a hardware or operating system malfunction, which preserves database integrity.

3.3.4 Business process

A business process is a recipe for achieving a commercial result. Each business process has inputs, method and outputs. The inputs are a pre-requisite that must be in place before the method can be put into practice. When the method is applied to the inputs then certain outputs will be created.

A business process is a collection of related structural activities that produce a specific outcome for a particular customer. It can be part of a larger, encompassing process and can include other business processes that have to be included in its method.

3.3.5 Other issues

3.3.5.1 "Dirty reads"

Oracle achieved an advancement that no other database vendor accomplished for quite some time afterwards (ORACLE, 2001), when they touted database read consistency with their 1984 release. Microsoft SQL Server 2000 for example, does not offer this feature. This ensured that when a read was performed on the data, the data would be exactly as it was at the time the read was performed. To get accurate data from a point in time, you had to freeze transactions – lock everyone else out of the database – rather impractical for large institutions with a definite impact on system performance.

Otherwise it could happen that changes written to the database in a transaction format could not be fully applied to the database, and with an error occurring, this data change could be reversed. If, at a point just prior to the reversal or roll-back, a read process reads data already changed in the database that forms part of this "transaction", it would read data that has not truly been applied to the database since there's a dependency on the success of all other changes within the transaction. Thus, if this transaction is rolled back and a read is done on the same records again, different results may be returned.

To accomplish "clean reads" in the prototype database, the simplest method is to create an empty temporary table with the structure of the entity to be queried. Then, at the instant where the query is initiated, entries in the locking table that will result in changes to the entity to be queried are copied to this temporary table to keep a history of changes – a "before" snapshot of the data, so to speak.

Once the query is completed, data in the temporary table is compared to the query results. If changes were made to data during the query run, these records are updated. If new entries were created/deleted, these entries are appended/removed. This process works on the assumption that entries can be read from the lock table and passed to the application layer of the DBMS faster than data can be written to the lock table.

3.3.5.2 *Choice of programming language*

Assembler was utilised in many of the first database systems. Oracle was known for having developed their first RDBMSs for specific hardware platforms using assembler. Assembler, or assembly language as it is also known, can provide highly optimised systems since it's a very low-level language allowing one to access all resources available on the hardware platform used. However there is also the inescapable OS/hardware specific nature of it, making it difficult to distribute to a number of customers using different operating systems and hardware.

C/C++ is a powerful language with OS independence (note that there are differences between C and C++, but C++ is widely regarded as an extension of C and is thus grouped together with it) due to the standardised development tools and portable core. It provides enough functionality to allow explicit resource allocation and control with sufficient high level functions to provide a wide variety of interfaces and data manipulation techniques as well as defined data types.

Perl was developed as a scripting language for mainly administrative functions on Unix systems, but it has since found various new applications thanks to its rich functionality. It has a deep routed C-based founding, including powerful C functionality (such as fseek in the function seek), but also incorporating strong string manipulation capabilities. It has no specific variable types other than scalars, arrays and hashes – all variables are handled internally – and the language is compiled at runtime. It further provides easy integration with web servers as it can be used to implement a Common Gateway Interface (CGI) which allows for web-based scripting and control – thus supplying one with the means to create a widely available user interface based on a web server, requiring no new software other than a standard web browser. This language is used in the practical portion of this study.

3.3.5.3 Further possible additions or enhancements

- XML support – de facto standard for data transfer
- SQL support
- Proprietary compression methods for use in a backup for example, or to store segmented BLOBs.
- Proprietary encryption for security
- Backup functionality

4 Implementation

4.1 Description of system

To practically implement the design in a typical real-world system, it is applied to a Knowledge Management System (KMS). KMSs are utilised primarily in service related industries where a certain set of problems can occur in the process of maintenance of products or defined service delivery. These types of systems also find application in research and innovation.

Knowledge management can be described as the systematic process of finding, selecting, organising, distilling and presenting information in a way that improves an employee's comprehension in a specific area of interest. Knowledge management assists an organisation in gaining insight and understanding from its own experience.

Specific knowledge management activities help focus the organization on acquiring, storing and utilising knowledge for such things as problem solving, dynamic learning, strategic planning and decision making. It also protects intellectual assets from decay, adds to firm intelligence and provides increased flexibility. The concept of knowledge management is illustrated as a graphical process shown in figure 13 (Zack, 1999).

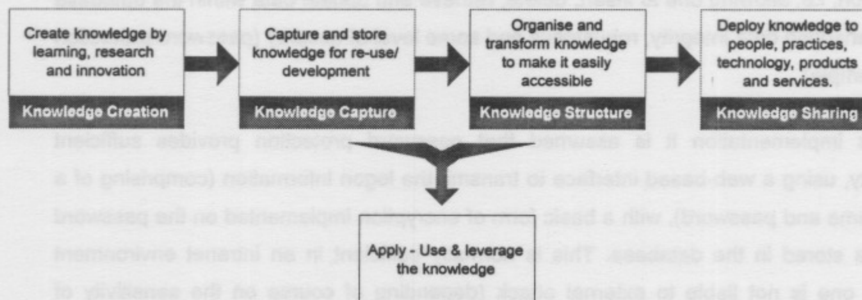


Figure 13: Knowledge management

The benefits of such a system in a large organisation where it is often difficult to distribute information on certain areas or issues to the people who will benefit most from it, is obvious. This also translates into a cost saving through minimising the duplication of

effort, decreasing man-hours spent and also indirectly improving customer relationships through perceived improved performance – albeit only in the initial stages.

Now with the use of a knowledge management system based on a computer architecture (which includes a physically implemented database of some kind), this information can be made accessible to employees almost anywhere in the world. Current information can be categorised, searches conducted for specific areas and information can be updated or removed as required.

4.2 Assumptions

For the purpose of this implementation, only certain features are implemented to limit the complexity. Databases and their management systems comprise of a multitude of components that require in-depth knowledge in their own right, such as security, graphical user interfaces tools, backup capability and SQL compatibility to name but a few.

These areas of functionality are discussed in the paragraphs that follow, but they are not necessarily implemented physically or in their entirety. The main focus of the implementation lies with the standard functionality to be provided by the DBMS as per definition, i.e. allowing one to insert, delete, retrieve and update data within the database while ensuring data integrity, robustness and some level of security (password protection for example).

In this implementation it is assumed that password protection provides sufficient security, using a web-based interface to transmit the logon information (comprising of a username and password), with a basic form of encryption implemented on the password as it is stored in the database. This is normally sufficient in an intranet environment where one is not liable to external attack (depending of course on the sensitivity of information stored in such systems). It can be supported by the use of cookie validations, stored by the browser.

Due to the nature of the data which can comprise of fairly long text descriptions, not all data will be buffered in memory. For the purpose of searching for records without taxing

disk usage and hardware, indexes are buffered when sorted. This will also limit the complexity to ensure data integrity with concurrent users, power and hardware failures.

Data as stored on the file system is not encoded. This is to allow for transparency in the design and to simplify debugging in the development process. By encoding the data the DBMS can do physical redundancy checks on the data and it can also result in compression on the data itself so that less physical disk space is utilised. This encoding and decoding would however translate into added processing by the hardware in question, and as such must be optimised to provide maximum functionality, with a calculated overhead in resource consumption. Optimisation should accordingly be achieved by balancing the benefit of such encoding against the processing time and resources required to this end.

Also, due to the nature of the data and the typically anticipated usage profile (i.e. mainly for searching and reading), it is unlikely that a large volume of changes will take place on concurrent records. Additionally, since only a few users are likely to access the same records at the same point in time, the real performance benefit of placing the data in memory is restricted.

By not encoding the data, strict byte size limited variable definitions are not enforced, effectively resulting in all variables consisting of a base unit type of one byte. Thus, a variable that can consist of eight (8) characters is defined with a size of 8 bytes. This is admittedly a non-optimised solution, but in essence encoding and decoding can be accomplished by adding two inter-dependent modules within the systematic chain of processes that constitute the DBMS, without these modules altering the way in which the system operates, as per figure 14.

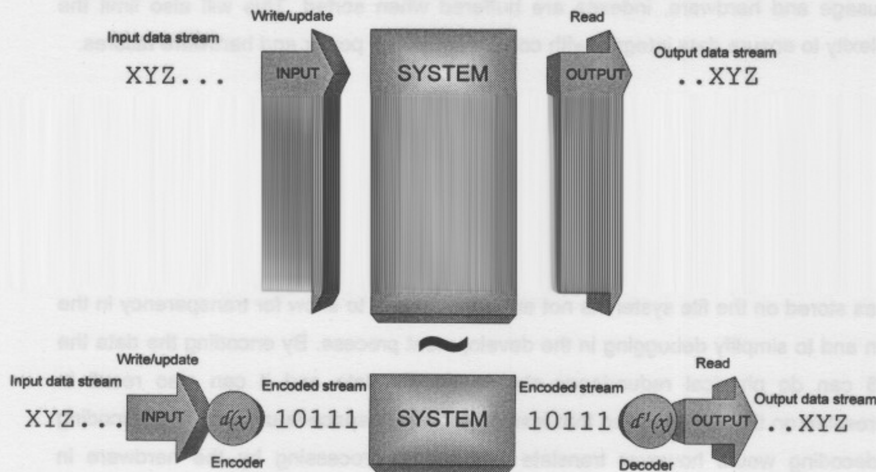


Figure 14: Encoding & decoding

Figure 14 shows how this can be accomplished where $d(x)$ is the encoding function and where $d^{-1}(x)$ is the decoding function, so that $d(d^{-1}(x)) = x = d^{-1}(d(x))$, i.e. where the transformation performed by the one function is reversed by the other. When a write/update/delete is sent to the database system, the encoding function encodes the stream for reasons of integrity, security or compression, and lets the database system save the transformed data. Upon a read or extraction of this data, the decoder function transforms the data to its original input state, but without actually changing the data in the database. Perceptually the data itself does not change to the user viewing/using it, and as such is equivalent in terms of the input received in relation to the output generated.

No secondary indexes are implemented as the principles for access remain the same – they are just more elaborate. Further, due to the relatively short record length, the performance gain obtained through this would be negligible as the index size compared to the actual table size would be much the same. The real benefit results with tables that are accessed or scanned using field values that are not contained in the primary index fields, in which case these secondary indexes can serve as short (in terms of field length) pointers to the relevant fields – thus allowing faster reads. This obviously makes the insert process slower since for every new entry an entry has to be added to the index as well.

4.3 *The nature of the input*

This database will be concerned with an information technology business which renders some supporting services relating to the maintenance of an information technology infrastructure (servers, printers, workstations, software, etc.). It will provide a user with search functionality (on certain key words or phrases designated by the author) and the ability to enter new knowledge in certain areas or to update/remove existing information. There is also the possibility of uploading documents relating to this knowledge to provide further supporting information on the issue. These documents are then merely catalogued in the database, while being stored on the file system of a selected server.

In more specific terms, the data that serves as input for the system will mainly be of the form of a username and password (to restrict access), a category for the type of information stored, key words to allow for searching, a short text description, and file system references to indicate any documents related to this data.

In more detail, we have the following four main tables:

- a table containing user credentials (USERS),
- a table containing the various items entered in the knowledge management database (KMITEM),
- a table containing the knowledge category and descriptions (KMCAT) that is associated with each KMITEM, and
- a table containing references to documents relating to some items (KMDOC) – merely a path to a shared network resource containing the documents related to the item in question.

A table matrix is shown below in figure 15 which displays the relations between the tables mentioned above.

	USERS	KMITEM	KMCAT	KMDOC
USERS		1:N		
KMITEM	1:1		1:1	1:1
KMCAT		1:N		
KMDOC		1:1		

Figure 15: Table relation matrix

This in effect translates to the ERD for the database layout shown in figure 16, using the simplification rules as provided by Hernandez (2003:349-350). That is, there exists a one to many relationship between USERS and the KMITEM table – one user can have many entries in the KMITEM table; there is a one to many relationship between the KMCAT and KMITEM tables since a knowledge management category can be assigned to multiple items; and finally, there is a one to one relationship between the KMITEM and KMDOC tables since each item can only refer to a single document directory/repository.

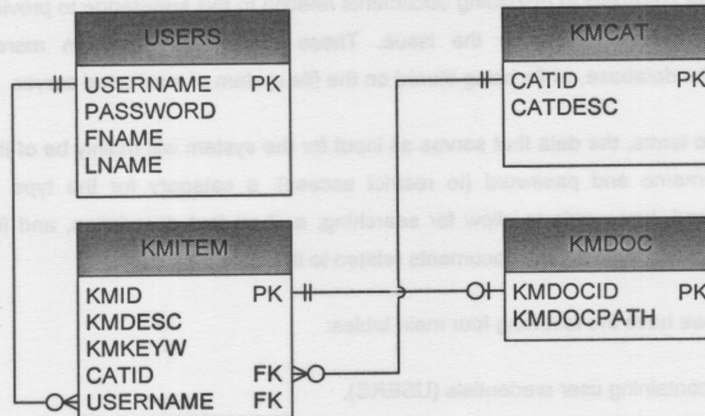


Figure 16: ERD for KM system

The fields shown in the ERD are defined in the table listed in figure 17.

Table	Field	Description	Field type	Length
USERS	USERNAME	Unique username	Primay key	15
	PASSWORD	Password (encrypted)	Standard	10
	FNAME	User's first name	Standard	20
	LNAME	User's last name	Standard	20
KMITEM	KMID	Unique KM item identifier	Surrogate primary key	7
	KMDESC	Description of the item	Standard	30
	KMKEYW	Multi-valued field containing key words for the item	Standard	50
	CATID	Linked category ID	Foreign key	5
	USERNAME	Linked username	Foreign key	15
KMDOC	KMDOCID	KM document ID	Surrogate primary key	8
	KMDOCPATH	Path to documents	Standard	20
KMCAT	CATID	Category ID	Surrogate primary key	5
	CATDESC	Category description	Standard	30

Figure 17: Fields for KM system

However, to accommodate the multi-valued key word field, a new table needs to be created. This results in an amendment to the ERD, as illustrated in figure 18.

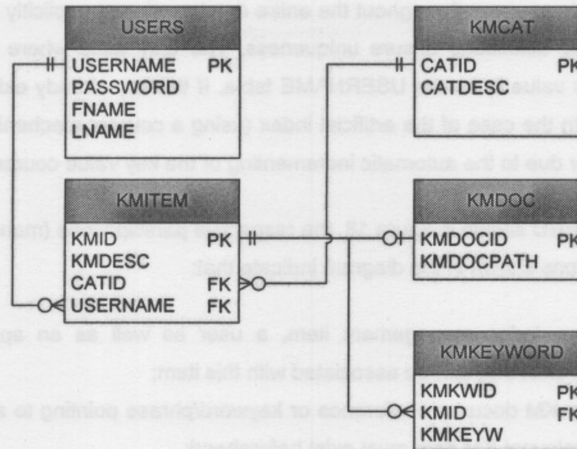


Figure 18: Updated ERD for KM system

Note from figure 18 that the KMKEYW field, as defined previously, has been removed and it has been replaced by a KMKEYWORD table. This table then in turn links the multiple values with the knowledge management item as in the KMITEM table by way of the KMID foreign key. The field descriptions for this additional table are listed in the table shown in figure 19.

Table	Field	Description	Field type	Length
KMKEYWORD	KMKWID	Unique KM keyword identifier	Surrogate primay key	10
	KMID	Knowledge management item ID	Foreign key	7
	KMKEYW	Keyword or phrase	Standard	20

Figure 19: Multi-value field to table expansion

Note that there are no default values specified for the foregoing tables. If a record is entered in any given tables, all field entries in that table are required upon the time of insertion with an option to edit all fields excluding the key fields, at a later point. Surrogate/artificial primary keys are of the form XX#####, where XX is a character format table descriptor and ##### is a number which is continually incremented with the

addition of each new record. The artificial keys were chosen in absence of suitable candidate keys.

The uniqueness of primary keys throughout the entire database is not explicitly enforced since the artificial key definitions ensure uniqueness. The only table where one can specify a primary key value is on the USERNAME table. If the key already exists, then an error is returned. In the case of the artificial index (using a counter mechanism), this situation will not occur due to the automatic incrementing of the key value counter.

Referring to the final ERD shown in figure 18, the respective participations (mandatory or optional) for the relations shown in the diagram indicate that:

a. to enter a knowledge management item, a user as well as an appropriate category must exist that can be associated with this item;
b. to add either a KM document reference or keyword/phrase pointing to a specific KM item, the relevant KM item must exist beforehand;
c. and lastly, all other relations do not require the existence of related fields.

4.4 System overview

The system can be segregated into three layers or tiers.

- *Presentation layer*. The presentation layer consists of workstations with web capabilities provided through some or another web browser, or via web-enabled devices. Since the scripting employed here mainly executes on the server, there is a limited requirement for any additional components (however, security components can be added).
- *Application layer*. The application layer comprises of the web server components as well as certain data manipulation programs embedded in the DBMS. The web server has to be CGI compatible, in this case specifically with Perl.
- *Data(base) layer*. This is where the database functionality resides as well as the storage portion.

The elements mentioned above are diagrammatically represented in figure 20, which symbolises the systematic layout for an environment where the system is deployed on an intranet – that is, within a private corporate network.

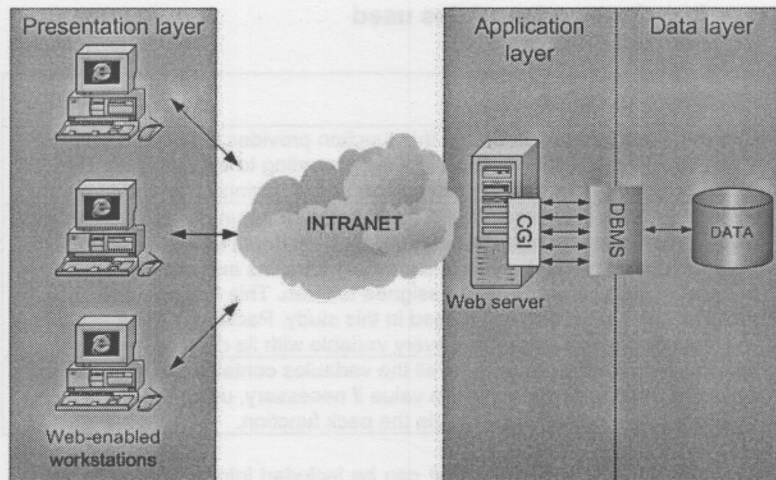


Figure 20: KM system overview

For the web server, the freely available, open-source, multi-platform Apache HTTP server is used. This, along with a Perl runtime environment is used to provide a powerful application interface to the database itself, with the main functionality built into the Perl scripts that are invoked with each connection attempt through the web server. This leaves the database in a fairly dormant state when not in use, but using operating system tools or uninterrupted processes, states can be triggered which result in certain database activities, such as a backup for example.

4.5 Perl

Perl executable code is only compiled at run-time when invoked by the web server. This results in a small overhead which could become significant in systems with a high number of transactions. However, with the assumption that this is a small system where load is mostly dominated by reading in this application, this problem is negated.

Perl provides a number of useful built-in functions that provide a fairly simple way of dealing with records, text formatting, arrays, creating of dynamic HMTL with CGI and disk access, to name but a few. Some of the functions used are briefly discussed in the following paragraphs.

4.5.1 Primary functions and modules used

Function name	*Function description*
seek	Like the fseek function in C/C++ this function provides a program with ability to move to a specific area in a file according to a byte offset. This offset can be from the current file cursor, or the beginning or end of the file.
pack	Pack provides a program with the ability to place a number of variables into a stream of data according to certain variable formats for each variable. Using this, one is able to give fields certain domains and allow the necessary storage areas to be assigned to them. This function also provides the set record length used in this study. Packing is done according to a template which associates every variable with its domain/type.
unpack	This function returns an array of all the variables contained in a stream (or record in this case), decoding the value if necessary, using the same template used to store the data via the pack function.

Modules are classes of function modules that can be included into programs. In effect, they comprise of a pool of functions to choose from. In this study, only one non-standard module was utilised.

Modules used	*Module description*
perl/pod.pm	This is the module containing the standard functionality of Perl such as mathematics and I/O operations. It is implicitly included in each program written in Perl.
CGI.pm	This module provides functions to quickly and neatly create CGI/dynamic HTML page output.

Some self-defined functions were required to allow for modular ease of programming and efficiency. The main functions are listed with a description of each, as they are implemented in the main program code.

Self-defined function	*Description*
get_rep_info	Reads and returns the repository information for a given table. This includes the field names, definitions, relations and constraints placed on this table.
read_pindx	Reads either the address or the record for any given primary key value for either ordered or unordered indexes.
read_scan	Scans through the index and reads all data according to a match or exact value criterion.
write_to_add	Writes a specific data stream to a given address (block number and in the data file).
read_data_add	Read data from a specific block and record location in the data file
insert_rec	Insert a new record in the data file

Self-defined function	Description
update_rec	Update a read record (on primary key) with provided field values. Primary keys cannot be update.
delete_rec	Similar function to update_rec except that a hash (#) value is written to the primary key and the primary index entry for this record is removed. Thus the database has no way of referencing to the entry as associated with this table.

4.5.2 Custom functions in detail

4.5.2.1 Read

The normal read is dependent on the primary index as it retrieves all related records from this table – i.e. a read is conducted on every index and address relation. In case the primary index is lost through deletion or hardware failure, the pointer record contained in every data block can be used as a reference to the next data block for the table in question.

The reading algorithm can be disabled when reading data from unsorted indexes. This will then result in a sequential read through records in the table until the required record is found.

4.5.2.2 Insert

With each new inserted record, the index(es) have to be updated as well, after which these indexes are resorted to allow the reading algorithm to function. If the reading algorithm is not used (with an impact on disk or memory activity), no re-ordering is needed.

Where artificial keys are used, these number sequences are generated from the key counter. The last inserted counter value is stored in the repository file and it is incremented with each new addition. In this case no reorganisation of the index is needed after insertion, as the counter mechanism will automatically result in an ordered entry system within the database.

Once a block is filled, a new data block of 8 KB is allocated, filled with temporary place holding characters, hashes. These hashes are also used when deleting records to

indicate a null entry. This address is then written to the previous data block to allow for sequential reads directly on the data file.

4.5.2.3 Update

An update reads the record in question first (according to a primary key value), then allows a user to change an entry, after which this is written to the original data address for this record, in effect over-writing the original record. This results in no change to the primary index as primary indexes cannot be changed via updates. In case secondary indexes exist, these will have to be updated – this is not catered for in the prototype database.

4.5.2.4 Delete

Deletion uses the same methods as implemented in the Update function where it over-writes the record in question with hashes. To lessen the I/O on the file system, only the primary field value is over-written. With the deletion, the index(es) need to be updated and re-ordered as well.

4.5.2.5 Cascaded delete/deletion rules

Deletion rules can be maintained in the repository. By default, the deletion of records that have child tables related to them (this information is also in the repository), will result in a cascaded delete. That is, the parent entry will be deleted and any entries in the child tables that have the same foreign key value as the primary key value being deleted. The name of this field will be the same as the field name of the primary key in the parent table.

4.5.2.6 Relation maintenance

For each table definition, the repository also contains two pieces of information that describes the relationships for this table. First it indicates which tables are child tables to this table (in case it is a parent table). Secondly, in the instance where it is a child table, the parent table name and the type of participation with the parent table are indicated (mandatory/option).

5 Results

5.1 Database functions

The basic functions of the relational database were implemented successfully, as applied to a relational database as per the ERD in the previous chapter. This proved to be somewhat more difficult than first anticipated due to the inherent requirement of flexibility, even though the assumptions simplified the system to a great extent. The DBMS accordingly does allow for extension by adding table and relation definitions in the repository (and as such it is effectively a RDBMS), thus enabling it to be used for different applications and is thus not strictly confined to this application.

The code for these functions together with the CGI functionality embedded (mainly just consisting of formatted HTML output) are included in Appendix B with actual programmatic display and function examples shown in Appendix A. Security related checks consist of password and cookie checking, but encryption was not activated to allow for greater transparency during checking.

Data file locking – to allow for consistency during multi-user access – is provided via a central lock file which is occupied for the remainder of a change action with a timestamp and table name indicator to show the table change target. If added changes are requested these are queue-based in 250 ms iterations. This is not suitable for high performance systems such as financial or service systems where any delays could have a financial impact or add to customer frustration. In ERP systems such as SAP, response times of more than 1 second per activity are deemed to be slow. In the designed system, that would be exceeded the moment more than four people attempt to access the same record. However, for non-critical systems (e.g. web-based information systems) which do not process the same volume as business systems, it would be within the acceptable performance parameters (usually measured in multiple seconds rather than split-seconds) where the database response time is comparatively small compared to the transmission and rendering times before information can be viewed by the end user.

5.2 Shortcomings

Even with the non-sequential access the prototype proved to be more disk I/O dependent than was initially planned due to lacking full data buffering functionality. This can also be attributed to the program functions operating as independent units and not in unison - not sharing resources and information between them. This also leads to higher processing requirements that could be contained in other instances of programs performing the same function. Combined with this is the separation of index and data files, which fragments the system structure and thus increases its manageability demand.

There is no provision for degrees of participation whereby minimum or maximum record numbers can be specified for relations between tables. Together with this, the type of participation relates directly to deletion whereby related child table entries are deleted if the parent entity is removed. There are applications where different deletion rules could be necessary.

The addition of new tables would require the explicit definition and implementation of a function to enforce the primary key uniqueness throughout the database. This uniqueness is implicitly the result of the artificial key selection in the prototype system which takes place automatically without user intervention. This is one of the relational model requirements.

The usage of Perl has the limitation that the code used to implement the functions is compiled at run-time. This would result in significant overhead in high transaction systems. Also, the usage of Perl and a web interface warrants stricter security mechanisms for using the system on the World Wide Web.

The centralised lock handling method implemented can prove to be a major pitfall as the system is scaled to multi-user transactional or volume-intensive applications. Greater provision for parallel processing, while maintaining data integrity, is required to scale the system properly.

Even though the design method followed the rules imposed by the relational data model, a fully functional RDBMS would require programmatic enforcement of these rules.

Lastly, the true limitations of these techniques were not tested on a large scale. True usage performance analyses require further testing to prove the efficiency or limitations of some of the suggested techniques under high load in OLAP or OLTP applications. Each system requires its own enhancements, but there are sure to be common ground which can be studied and optimised for use in both.

5.3 Enhancements/improvements

Classes or function modules are available to provide inter process communication that would enable the database system to spawn/create dependent processes that can communicate with one another. For example the WIN32 class is available for the Microsoft Windows® operating system which allows a program to create processes and send variables between these processes. This will allow for greater "awareness" within the system in terms of the operations therein, providing greater flexibility and lessening superfluous computations where data/information can be drawn from a central pool which can, for example, be utilised for handling reliable, concurrent access.

Furthermore, processes can be spawned which can react to system event triggers. These triggers have to be bounded to a well-planned process flow within the DBMS to operate effectively. Special care should be taken so that no child processes are created without being shut down properly by themselves or under the control of the parent process. If these processes are not controlled, errant ones would result in operating platform instability that will definitely impact the DBMS as well.

References or Pointers are available to share variable assignments to memory locations. Pointers in Perl are not as definite as that of C/C++ in terms of returning actual memory addresses, but these functions are useful for quickly passing variables (scalars, arrays, associative arrays) between functions since only memory locations are passed and not the variables themselves. If implemented correctly, this can prove highly effective in optimising the system.

Where inter-operation is required with other software, a suitable and standardised means of interfacing with these systems is needed. This is where XML and SQL support come into play, allowing applications to integrate with the multitude of other systems and software that already follow the standard.

To provide greater flexibility a higher degree of modularisation is also required so that new components can be added without affecting the core elements of the system. The concerned modularisation would require a few iteration steps to finally determine the best system process configuration. This can then be implemented with due diligence of hardware and operating platform limitations and capabilities.

6 Conclusion

In view of the tendency that software and data storage will continue to engulf the increasing capacity of hardware resources available to them (Moore's Law, etc.), it is difficult to envision truly optimised systems. Optimisation often comes at the cost of re-engineering. This occurrence, coupled with the rapid advancement and high levels of competition in all software and microcomputer hardware markets, often lead to non optimised solutions since vendors cannot afford to "re-invent the wheel".

There can no longer be a jack-of-all-trades system that caters for all forms of data storage applications, as their design and configuration limitations ultimately lead to poor performance. Instead application specific solutions are more likely to become evident as systems and requirements grow, since they provide various performance gains by being moulded around the data being processed. This moulding or adaptation of software to the data used can very likely be addressed, at least to an extent, by fuzzy logic or neural network systems that can recognise and adapt to measures/key features of the system to provide optimum efficiency.

A further method is returning to hardware and software inter-dependence where software is developed specifically for hardware or peripheral platforms. This has the pitfall of restricting customers to software and hardware vendors that operate as partners. As diversity often results in less risk exposure than exclusivity, this is an unlikely stance to be taken in the market, unless the vendor itself produces both the software and physical infrastructure, as is the case with a company like IBM.

On the other hand the need for "open" systems that integrate seamlessly could outweigh performance concerns, driving the market towards more open and feature rich systems. This is in contrast to the streamlined optimisation that is represented by the high performance application specific systems. A possible compromise could be reached by including so-called "middleware" or portals that interpret proprietary data and transforms it into an open standard such as XML. This will allow for optimum performance on the data layer, whilst providing greater functionality and integration capacity on the application layer (through the implementation of aforementioned standards) of the system.

In an analogy, the human neuron/memory element processes a signal at a slower speed than a modern microcomputer, however, due to the amount of parallelism the human brain remains far superior to these microcomputers. Similarly the relation holds for database systems – the greater the parallelism in a database system, the greater the speed of processing within this system. The crux of this remains coordination of these parallel processes in such a way that data integrity is not affected, and this presents a major challenge.

Stored databases will continue to become ever increasing parts of our daily life in the information technology era. At some point physical limitations will put a stop to the explosion of processing power and storage capacity and at that junction many software vendors will have to head back to the drawing board to be able to provide a better solution than their competitors. There is a great possibility that a new system would then overrun existing market giants, simply because its founding is of such a nature that it allows it to be optimised in view of hardware technology at that time. Since there is always a trade-off between flexibility and optimisation, it remains to be seen which will triumph.

7 References/Bibliography

BELLINGER, G. (2004). "Knowledge Management—Emerging Perspectives", [Web:] http://www.systems-thinking.org/kmgmt/kmgmt.htm, [Accessed:] Dec, 2003.

CHEN, P. P. (1976). The Entity-Relationship Model – toward a unified view of data. ACM Transactions on database systems 1, March.

CHEN P.P. (1977). The Entity-Relationship Approach to logical Database Design, QED information sciences Wellesley (Mass.), September.

CHOO, C. W. (1995). "Information Management for an Intelligent Organization: The Art of Environmental Scanning". Medford, NJ: Learned Information.

CODD, E. F. (1970). "A Relational Model of Data for Large Shared Data Banks", Communications of the ACM, vol. 13 #6 (June).

CODD, E. F. (1979). "Extending the Database Relational Model to Capture More Meaning", ACM Transactions on Database Systems, vol. 4 #4 (December).

CODD, E. F. (1985). "Is Your DBMS Really Relational?", ComputerWorld, (Part 1: October 14, Part 2: October 21).

CODD, E. F. The Relational Model of Database Management Version 2.

CORMEN, T., LEISERSON, C. & RIVEST, R. (1990). "Introduction to Algorithms". MIT Press.

DATE, C. J. (1995). "An Introduction to Database Systems", Addison-Wesley Publishing Company.

FLEMING, N. (1996). "Coping with a Revolution: Will the Internet Change Learning?", Lincoln University, Canterbury, New Zealand.

FOLK, M.J. & ZOELLICK, B. (1992). File Structures, 2nd Ed., Addison-Wesley.

FRICK, D. R. (2000), "Data integrity", [Web:] http://www.frick-cpa.com/ss7/Theory_DataIntegrity.asp; last updated 02/01/2000 [Accessed:] Dec 2003.

GOODLAND, M. & SLATER, C. (1995). SSADM -- A practical approach (version 4). McGraw-Hill.

HERNANDEZ, M.J. (2003). Database Design for Mere Mortals: a hands-on guide to relational database design, 2nd ed. Addison-Wesley.

INFOSEC, 1999. National Information Systems Security Glossary, NSTISSI No. 4009, January (Revision 1).

KROENKE, D.M. (2000). Database processing: fundamentals, design & implementation, 7th ed. Prentice Hall.

MEYER, T. H. (1997). Non-spatial Database Models, NCGIA Core Curriculum in GIScience, [Web:] http://www.ncgia.ucsb.edu/giscc/units/u045/u045.html, posted November 10. [Accessed:] Dec. 2003.

MOORE, G. E. (1965). "Cramming more components onto integrated circuits", Electronics, Volume 38, Number 8, April 19.

NAVATH, E. S. (1994). Fundamentals of Database Systems. 2nd Ed., Benjamin/Cummings Pub. Comp.

ORACLE CORPORATION, (2001). "50 Defining Moments", Oracle Magazine, November. [Web:] http://www.oracle.com/technology/oramag/oracle/01-nov/o61cover_50define.html

PARKINSON, C. Northcote (1958). "Parkinson's Law: The Pursuit of Progress". London, John Murray.

PASCAL, F. (1990). SQL and Relational Basics. Paperback: 336 pages. Publisher: Hungry Minds, Inc; (February).

RAO, B.R. (1994). Object-Oriented Databases: Technology, Applications, and Products. McGraw-Hill, Inc., New York.

RATLIFFE, W. (1988). "Interview", DBMS Magazine, vol. 1 #4 (December).

RUMBAUGH, J., BLAHA M., PREMERLANI, W., EDDY, F., AND LORENSEN, W. (1991) Object-Oriented Modeling and Design. Prectice Hall, New Jersey.

SIMMEL, S.S. AND GODARD, I. (1991) The KALA Basket: A Semantic Primitive Unifying Object Transactions, Access Control, Versions and Configurations. In Proceedings of OOPSLA '91: Conference on Object-Oriented Programming Systems, Languages and Applications, Nov., pp. 230-246.

TECHTARGET, (2003). Technical definitions search engine [Web:] http://searchdatabase.techtarget.com/sDefinition/0,,sid13_gci211895,00.html
http://searchdatabase.techtarget.com/sDefinition/0,,sid13_gci212885,00.html
http://searchdatabase.techtarget.com/sDefinition/0,,sid13_gci214260,00.html
[Date of access: Nov. & Dec. 2003]

OPENGROUP, (2003). The Open Group Base Specifications Issue 6 (IEEE Std 1003.1, 2003 edition) [Web:] http://www.opengroup.org/onlinepubs/007904975/functions/fseek.html. [Date accessed] Jan. 2003.

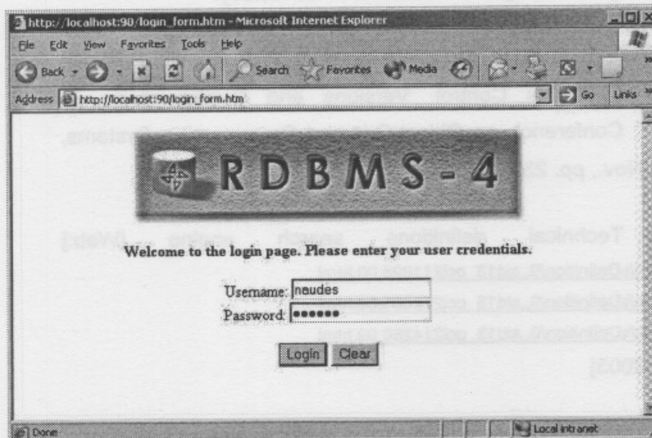
ULLMAN, J.D. (1988) Principles of Database and Knowledge-Base Systems, volumes I and II. Computer Science Press, Inc.

WIEDERHOLD, G. (1983). Database Design. 2nd Ed., McGraw-Hill Book Comp., NY.

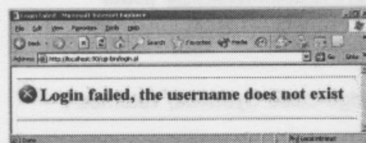
ZACK, M. H. (1999). "Managing Codified Knowledge", Sloan Management Review, Volume 40, Number 4, pp. 45-58.

A. Appendix A: Program output

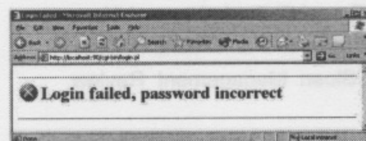
A.1 Login



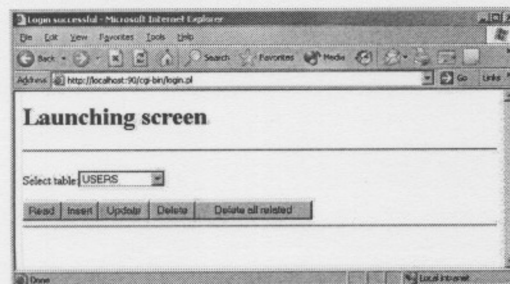
Incorrect username entered:



Incorrect password entered:



Launching screen (successful login):

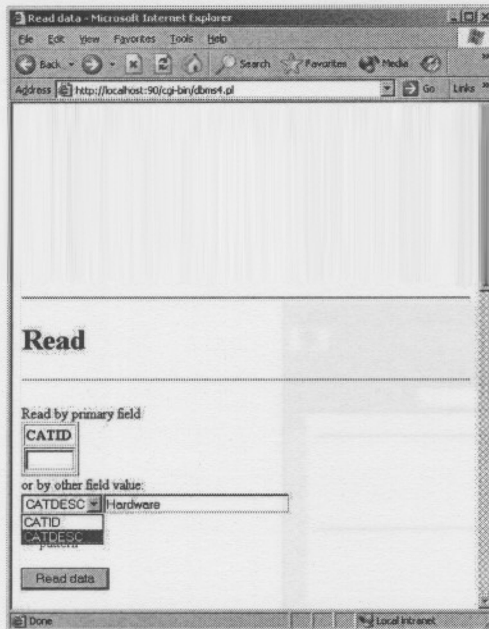


A.2 Reading records

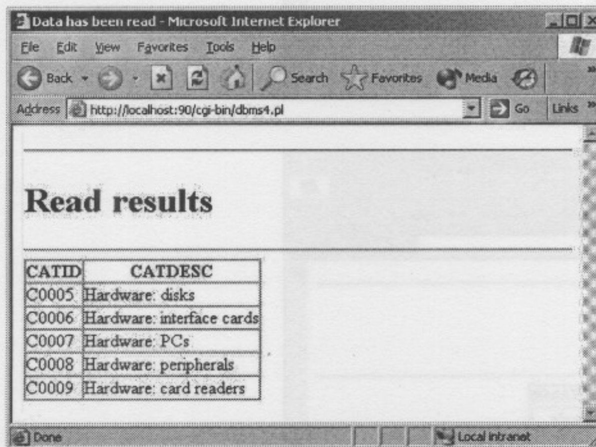
Read on primary key value (exact match):

Result:

Read a pattern (all or fields containing certain characters/strings):

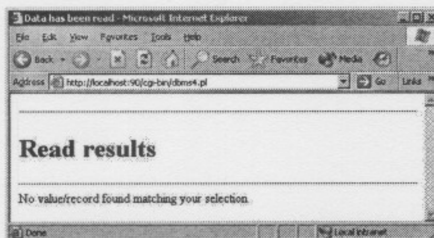


Result (all records in the Category table containing the phrase "Hardware" in the CATDESC field):



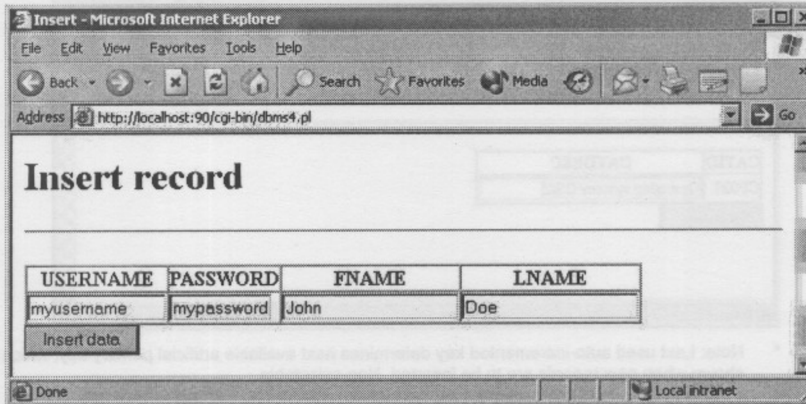
CATID	CATDESC
C0005	Hardware: disks
C0006	Hardware: interface cards
C0007	Hardware: PCs
C0008	Hardware: peripherals
C0009	Hardware: card readers

No record found:



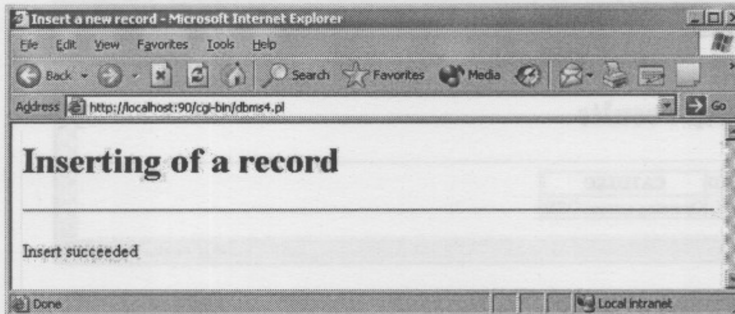
A.3 *Insert a new record*

Arbitrary key value (user selectable):

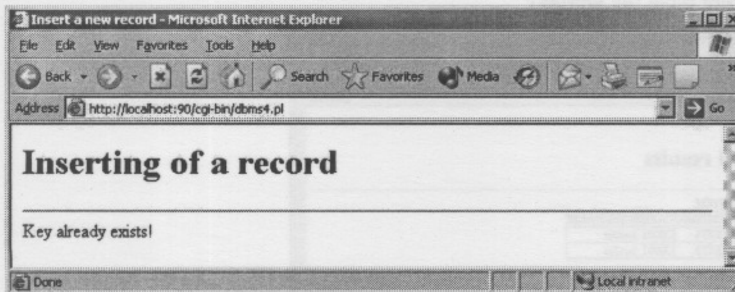


* Fields restricted in size according to field definition (byte-wise).

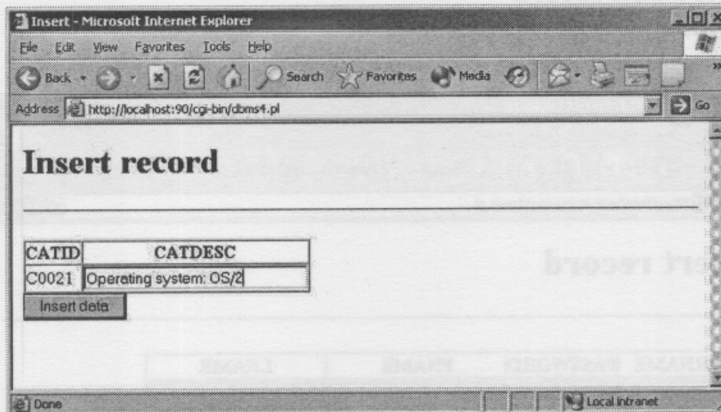
Successful:



Failed (key already exists – prevents duplicate record)

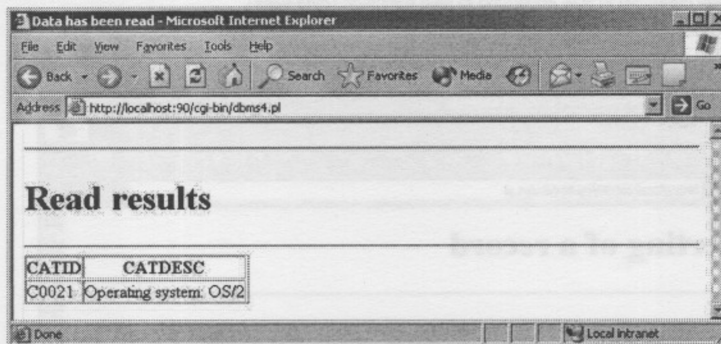


Inserting an artificial key (auto-numbered):



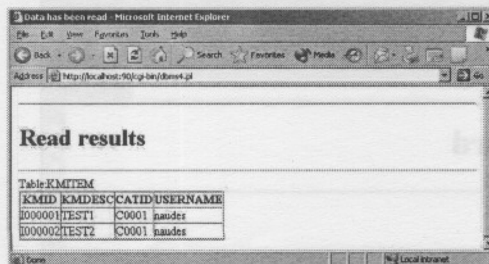
* Note: Last used auto-incremented key determines next available artificial primary key, which is shown when new records are to be inserted. Non-selectable.

Read result after insert:



Inserting record with parent (foreign key) dependency/mandatory participation

Read on parent table (all entries):



Insert on child table where mandatory foreign key does not exist:

Insert - Microsoft Internet Explorer

Address http://localhost:90/cgi-bin/dbms4.pl

Insert record

Table:KMDOC

KMDOCID	KMID	KMDOCPATH
D0000007	I000003	TESTPATH

Insert data

Result:

Insert a new record - Microsoft Internet Explorer

Address http://localhost:90/cgi-bin/dbms4.pl

Inserting of a record

Table:KMDOC
Mandatory key I000003 doesn't exist!
Insert failed.

Insert on child table where mandatory foreign key does exist:

Insert - Microsoft Internet Explorer

Address http://localhost:90/cgi-bin/dbms4.pl

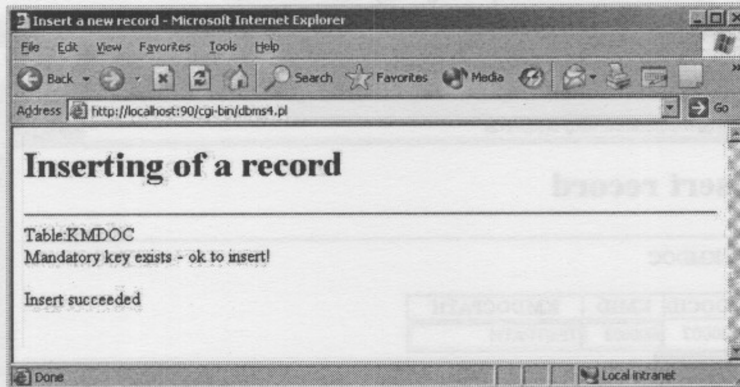
Insert record

Table:KMDOC

KMDOCID	KMID	KMDOCPATH
D0000006	I000002	TESTPATH

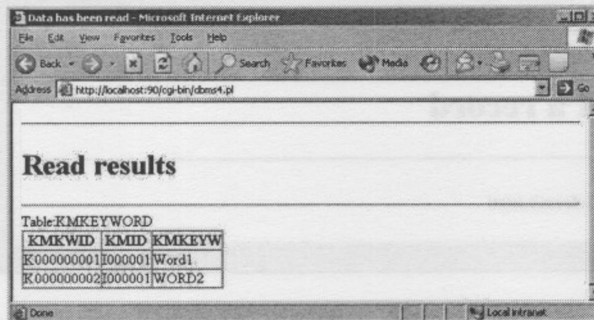
Insert data

Result:



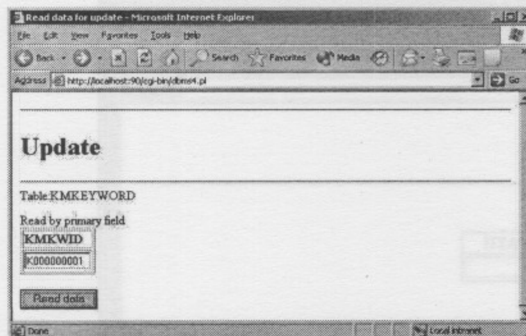
A.3 Update

Initial table contents:

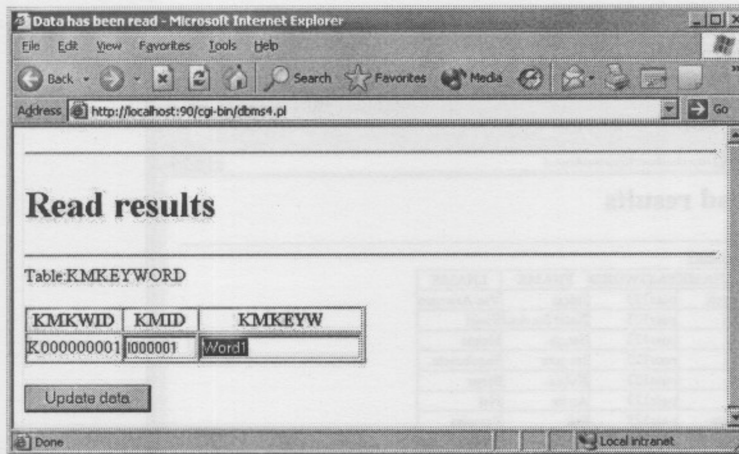


KMKWID	KMID	KMKEYW
K000000001	I000001	Word1
K000000002	I000001	WORD2

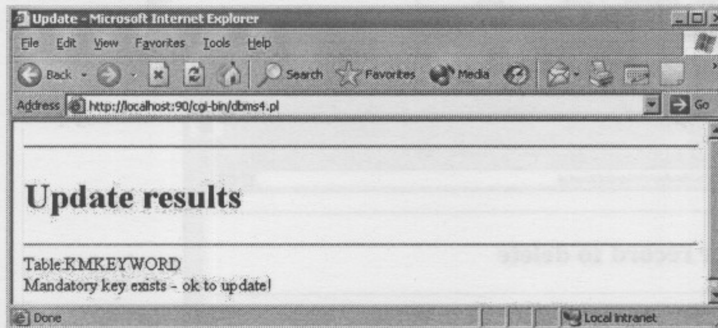
Read data:



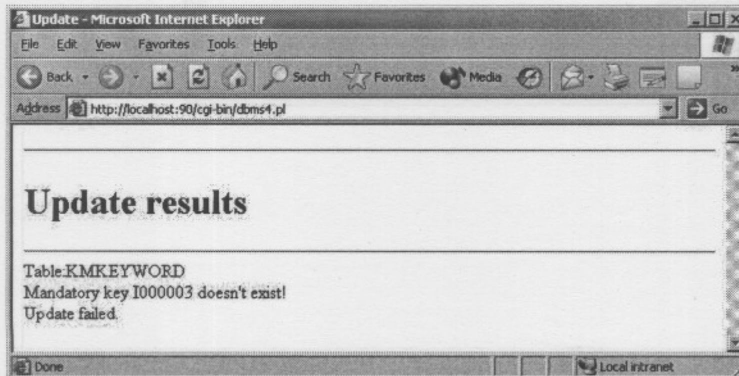
Return:



Successful update of a record with check on foreign key value:

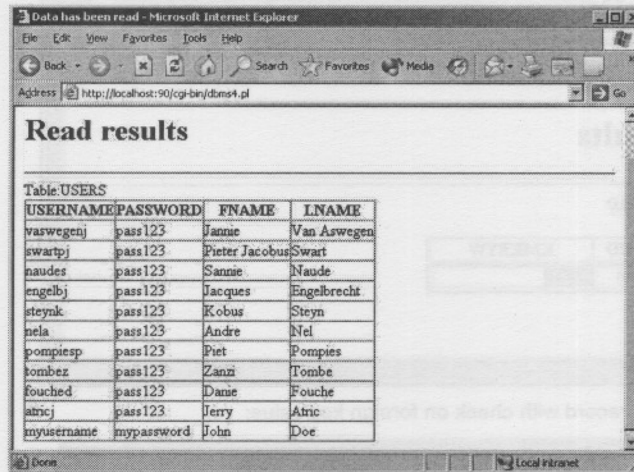


Update of record with failed mandatory foreign key field check:



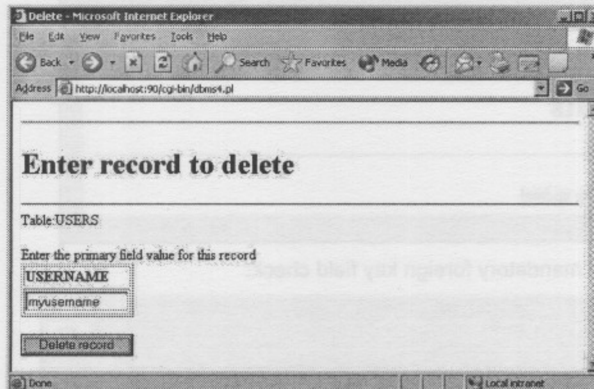
A.4 Delete

Initial table contents:

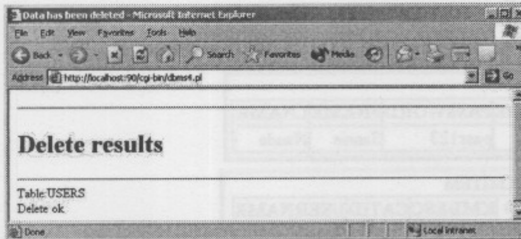


USERNAME	PASSWORD	FNAME	LNAME
vaswegenj	pass123	Jannie	Van Aswegen
swartpj	pass123	Pieter Jacobus	Swart
naudes	pass123	Sannie	Naude
engelbj	pass123	Jacques	Engelbrecht
steynk	pass123	Kobus	Steyn
nela	pass123	Andre	Nel
pompiesp	pass123	Piet	Pompies
tombez	pass123	Zanzi	Tombe
fouched	pass123	Danie	Fouche
atricj	pass123	Jerry	Atric
myusername	mypassword	John	Doe

Single delete (no check for child records):



Successful:



No record found:

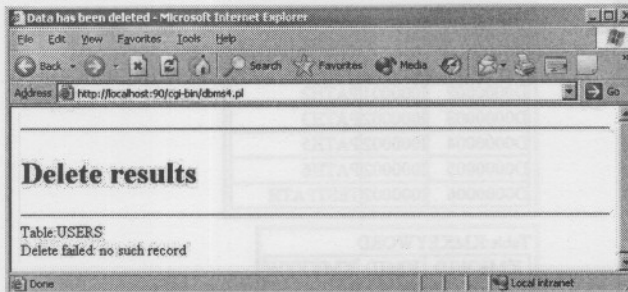
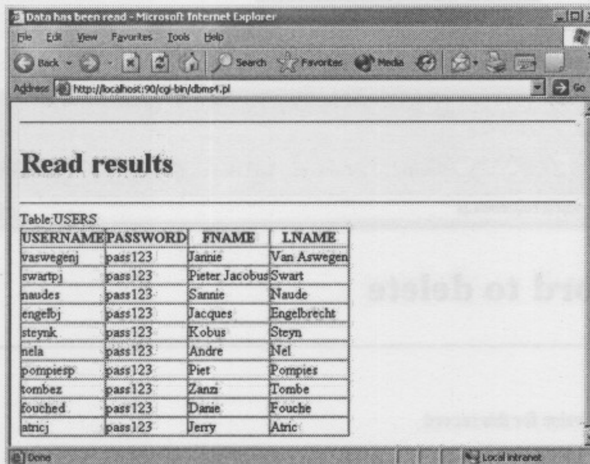


Table after deletion:



USERNAME	PASSWORD	FNAME	LNAME
vaswegenj	pass123	Jannie	Van Aswegen
swartpj	pass123	Pieter Jacobus	Swart
naudes	pass123	Sannie	Naude
engelbj	pass123	Jacques	Engelbrecht
steynk	pass123	Kobus	Steyn
nela	pass123	Andre	Nel
pompiesp	pass123	Piet	Pompies
tombez	pass123	Zanzi	Tombe
fouched	pass123	Danie	Fouche
atricj	pass123	Jerry	Atric

Cascaded delete (delete all related)

Related structures & tables:

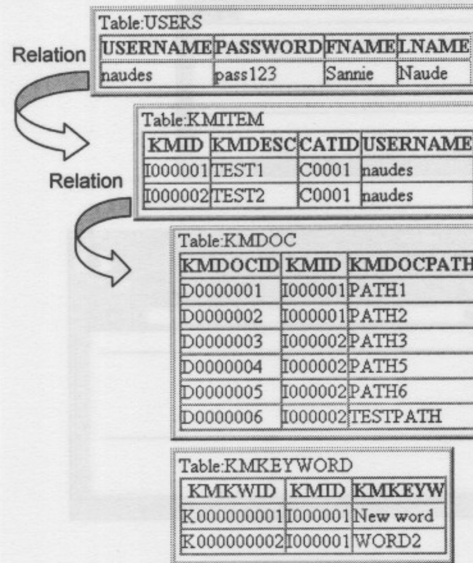


Table:USERS

USERNAME	PASSWORD	FNAME	LNAME
naudes	pass123	Sannie	Naude

Table:KMITEM

KMID	KMDESC	CATID	USERNAME
I000001	TEST1	C0001	naudes
I000002	TEST2	C0001	naudes

Table:KMDOC

KMDOCID	KMID	KMDOCPATH
D0000001	I000001	PATH1
D0000002	I000001	PATH2
D0000003	I000002	PATH3
D0000004	I000002	PATH5
D0000005	I000002	PATH6
D0000006	I000002	TESTPATH

Table:KMKEYWORD

KMKWID	KMID	KMKEYW
K000000001	I000001	New word
K000000002	I000001	WORD2

Selection of primary key, parent table:

Delete cascade - Microsoft Internet Explorer

File Edit View Favorites Tools Help

Back Search Favorites Media

Address http://localhost:90/cgi-bin/dbms4.pl Go

Enter record to delete

Table:USERS

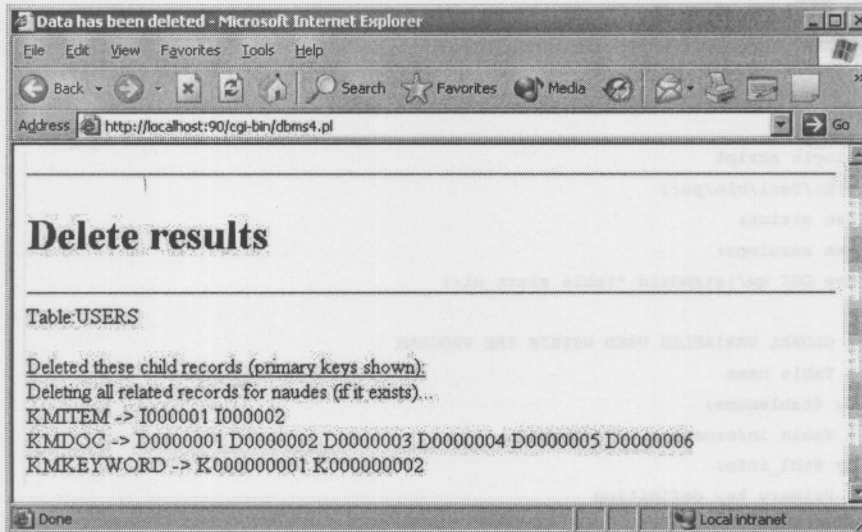
Enter the primary field value for this record

USERNAME
naudes

Delete cascade

Done Local intranet

Results:



* Note: Parent table entry remains, but all related records have been removed.

B. Appendix B: Program code

Login.pl

```
#login script
#!E:/Perl/bin/perl
use strict;
use warnings;
use CGI qw/:standard *table start_ul/;

# GLOBAL VARIABLES USED WITHIN THE PROGRAM
# Table name
my $tablename;
# Table information as returned
my @tbl_info;
# Primary key definition
my $pkdef;
# Record length
my $reclen;
# Packing template
my $rec_PT;
# Field names
my @field_names;
# Field definitions
my @field_defs;
# Fields
my %fields;
# Number of fields
my $numfields;


# READ REPOSITORY FILE/TABLE
sub read_rep_table {
  # INPUT: table name - table info to be read '*' for all tables
  my $tablenm=$_[0];
  # open file for reading
  open (REPOS,"+<repository.dat");
  my @lines = <REPOS>;
  close REPOS;
  # init variables
```

```
  my @all_tab = ();
  my $count_tab = 0;
  # Now @lines holds all the lines, one line in each element.
  foreach my $line(@lines) {
    # split colon separated values in the line
    my @values = split (/;/,$line);
    if ($tablenm eq '*'){
      # skip header line in repository
      push(@all_tab,$values[0]) unless ($count_tab == 0);
    }
    else {
      #checks if table (first entry) is = $tablenm
      return(@values) if ( $values[0] eq $tablenm );
    }
    $count_tab++;
  }
  return(@all_tab);
}

# READ DATA FROM DATA ADDRESS
sub read_data_add {
  # INPUT: address - from index, block & rec format
  # OUTPUT: record, packed format
  my $block = substr($_[0],0,6);
  my $blrec = substr($_[0],-4,4);
  open (DATA,"+<data.dat") || die "Can't open 'data.dat' for read: $!";
  # calculate byte address of block:record
  my $byteoffset = $block * 8192 + $blrec * $reclen;
  # move file pointer to new byte position with offset from start of file
  seek (DATA, $byteoffset, 0) || die "Can't seek to this position";
  # read data of the length of a record into a filestream
  read (DATA,my $filestream,$reclen) || die "Can't read";
  close DATA;
  return($filestream);
}

# READ DATA RECORD WITH PRIMARY KEY VALUE
sub readon_pindx {
  # PARAMETERS
  # INPUT: primary key value for read [0]
  # INPUT: primary key field definition [1]
  # INPUT: ordered(1)/unordered(0) primary index [2]
```

```
  # OUTPUT: data record at address - scalar
  my $pkey = $_[0];
  my $pkey_def = $_[1];
  my $ordered = $_[2];
  my $pkey_packed = pack($pkey_def,$pkey);
  open (INDX,"+<index_" . $tablename . ".dat") || die "Can't open 'index.dat'
for read/update: $!";
  # (-s INDX) = file size in bytes of INDX operator
  my $filesize = -s INDX;
  # create pack template for index - A10 size for address - 6 block, 4 rec *
8192 bytes
  my $pack_tpl = $pkey_def . " A10";
  # determine index length from template
  my $indxlen = length (pack($pack_tpl));
  # determine number of records in index (to determine # reads)
  my $numrecords = $filesize / $indxlen;
  # initialise variables for the loop
  my $numreads = 0;
  my $top_pos = $numrecords;
  my $bot_pos = 1;
  my $split_pos;
  my $address;
  my $stream;
  # search for index key
  for (my $i=1; $i<=$numrecords; $i++){
    # for unordered index
    if ($ordered == 0) {
      read INDX, $stream, $indxlen; # || die "Error reading index";
      # variable for number of reads required to get to desired record
      ++$numreads;
      if (substr($stream,0,length(pack($pkey_def))) =~ /$pkey/i) {
        $address = substr($stream,-10,10);
        return(read_data_add($address));
      }
    }
    # reading algorithm - depends on ordered index
    if ($ordered == 1) {
      if ($top_pos == $bot_pos) {
        seek(INDX, ($top_pos -1) * $indxlen, 0);
        read INDX, $stream, $indxlen; # || die "Error reading index";
        if            (uc(substr($stream,0,length(pack($pkey_def))))          ne
uc(pack($pkey_def,$pkey))) {
```

```
    return("ERROR: not key value");
    exit;
  }
  $address = substr($stream,-10,10);
  # gives indication of number of reads
  $numreads++;
  return(read_data_add($address));
}
# use integer type division
use integer;
$split_pos = ($top_pos - $bot_pos)/2 + $bot_pos;
seek(INDX, ($split_pos * $indxlen), 0);
read INDX, $stream, $indxlen; # || die "Error reading index";
my $read_pk = substr($stream,0,length(pack($pkey_def)));
++$numreads;
if (($top_pos - $bot_pos) == 1) {
  if ($bot_pos == 1) {
    seek(INDX, 0, 0);
    read INDX, $stream, $indxlen; # || die "Error reading index";
    my $read_pk = substr($stream,0,length(pack($pkey_def)));
    if (uc($read_pk) ne uc($pkey_packed)){
      seek(INDX, $indxlen, 0);
      read INDX, $stream, $indxlen; # || die "Error reading index";
      my $read_pk = substr($stream,0,length(pack($pkey_def)));
    }
  }
  $read_pk = uc(substr($stream,0,length(pack($pkey_def))));
  if (uc($read_pk) ne uc($pkey_packed)) {
    return("ERROR: not key value");
    exit;
  }
  $address = substr($stream,-10,10);
  # read data from address indicated by index
  return(read_data_add($address))
}
# read record is smaller than $pkey_packed (A < Z)
if ((uc($read_pk) cmp uc($pkey_packed)) == 1) {
  $top_pos = $split_pos;
}
# read record is larger than $pkey_packed (Z > A)
if ((uc($read_pk) cmp uc($pkey_packed)) == -1) {
  $bot_pos = $split_pos;
```

```
      }
      if ((uc($read_pk) cmp uc($pkey_packed)) == 0) {
        $address = substr($stream,-10,10);
        # read data from address indicated by index
        return(read_data_add($address));
      }
    }
  }
  close INDX;
}

# GET REPOSITORY INFORMATION
sub get_rep_info {
  # INPUT - table name - [0]
  @tbl_info = read_rep_table($_[0]);

  # Array structure for repository as returned
  # 0 TABLE
  # 1 FIELDS (separated by a space)
  # 2 SIZE (of record - pack format)
  # 3 PK (field name of primary key(s))
  # 4 RELATION (information for setting up relations)

  # determine record pack template
  $rec_PT = $tbl_info[2];
  # determine record length from template
  $reclen = length (pack($rec_PT));
  # read the field names
  @field_names = split(/ /, $tbl_info[1]);
  @field_defs = split(/ /, $tbl_info[2]);
  # create associative array with field name as index
  # and field size/template as associated field
  @fields{@field_names}=@field_defs;
  # determine number of fields
  $numfields = @field_names;
}

# FIND A VALUE IN A LIST
sub list_index {
  my ($value, @list) = @_;
  my $count = 0;
  foreach my $i(@list){
```

```
    return($count) if ($i eq $value);
    $count++;
  }
}

my $query = new CGI;
my $cookiename = "login_ticket";
my $checkcookie = cookie($cookiename);

# TABLE NAME
$tablename = "USERS";
my @record_val;
# get repository information
get_rep_info($tablename);
my $username = $query->param('uname');
my $password = $query->param('upass');

if ($username) {}
else {$username = "NULL";}
if ($password) {}
else {$password = "NULL";}

# READ ON PRIMARY INDEX
my $result = readon_pindx($username,$fields{$tbl_info[3]},1);
my $error_image = "<img border=\"0\" src=\"..\\images\\error.gif\">";
if ($result =~ /error/i) {
print header;
print
  start_html('Login failed'), # start the HTML
  hr,
  table({-border=>0},
         Tr({-align=>'RIGHT',-valign=>'TOP'},
         [
            td([$error_image  ,  h1('Login  failed,  the  username  does  not
exist')])
         ]
         )
      ),
  hr;
}
else
{
```

```
  @record_val = unpack($rec_PT,$result);
  if ($password eq $record_val[1]){
    # Password correct - save session cookie
    my $cookie = cookie(
    -name=> $cookiename,
    -value=> $username
#   ,-expires=> '+1h'    # if ommitted, stored in memory
    );
    print header(-COOKIE => $cookie, -expires=>'now');
    print
      start_html('Login successful'), # start the HTML
      h1('Launching screen'),         # level 1 header
      hr;
    my @tbl_list = read_rep_table('*');
    print
      start_form(-action=>'dbms4.pl');
    print "Select table:";
    print $query->popup_menu(-name=>'table_name',-values=>[@tbl_list]);
    print "<BR><BR>";
    print
      submit(-name=>'sub',-value=>'Read'),
      submit(-name=>'sub',-value=>'Insert'),
      submit(-name=>'sub',-value=>'Update'),
      submit(-name=>'sub',-value=>'Delete'),
      submit(-name=>'sub',-value=>'Delete all related');
    print "<BR>";
    print
      hr,
      endform;
  }
  else {
  print
    header,
    start_html('Login failed'), # start the HTML
    hr,
    table({-border=>0},
           Tr({-align=>'RIGHT',-valign=>'TOP'},
           [
              td([$error_image , h1('Login failed, password incorrect')])
           ]
           )
       ),
```

```
    hr;
  }
}
end_html;                    # end the HTML
```

DBMS4.pl

```
#!C:/Perl/bin/perl
use strict;
#use warnings;
use CGI qw/:standard *table start_ul/;

# GLOBAL VARIABLES USED WITHIN THE PROGRAM
# Table name
my $tablename;
# Table information as returned
my @tbl_info;
# Primary key definition
my $pkdef;
# Record length
my $reclen;
# Packing template
my $rec_PT;
# Field names
my @field_names;
# Field definitions
my @field_defs;
# Fields
my %fields;
# Number of fields
my $numfields;
# Primary key type
my @pkey_type;
# Child tables
my @c_tables;
# Parent tables
my @p_tables;
# Participation (parent entry required)
my @participate;


# READ REPOSITORY FILE/TABLE
```

```
sub read_rep_table {
  # INPUT: table name - table info to be read
  my $tbl_nm=$_[0];
  # open file for reading
  open (REPOS,"+<repository.dat");
  my @lines = <REPOS>;
  close REPOS;
  # Now @lines holds all the lines, one line in each element.
  foreach my $line(@lines) {
  # split colon separated values in the line
    my @values = split (/;/,$line);
    #checks if table (first entry) is = $tbl_nm
    return(@values) if ( $values[0] eq $tbl_nm );
  }
}

# GET REPOSITORY INFORMATION
sub get_rep_info {
  # INPUT - table name - [0]
  @tbl_info = read_rep_table($_[0]);

  # Array structure for repository as returned
  # 0 TABLE
  # 1 FIELDS (separated by a space)
  # 2 SIZE (of record - pack format)
  # 3 PK (field name of primary key(s))
  # 4 Primary key type (ARTIF - artificial (auto numbered)/CUSTOM - arbitrary
  # 5 Child tables
  # 6 Parent tables
  # 7 Participation - O -> optional; M -> mandatory

  # determine record pack template
  $rec_PT = $tbl_info[2];
  # determine record length from template
  $reclen = length (pack($rec_PT));
  # read the field names
  @field_names = split(/ /, $tbl_info[1]);
  @field_defs = split(/ /, $tbl_info[2]);
  # create associative array with field name as index
  # and field size/template as associated field
  @fields{@field_names}=@field_defs;
  # determine number of fields
```

```
  $numfields = @field_names;
  @pkey_type = split(/ /,$tbl_info[4]);
  @c_tables = split(/ /,$tbl_info[5]);
  @p_tables = split(/ /,$tbl_info[6]);
  @participate =split(/ /,$tbl_info[7]);
}


# READ DATA FROM DATA ADDRESS
sub read_data_add {
  # INPUT: address - from index, block & rec format [0]
  # INPUT: record length [1]
  # OUTPUT: record, packed format
  my $block = substr($_[0],0,6);
  my $blrec = substr($_[0],-4,4);
  my $reclength = $_[1];
  open (DATA,"+<data.dat") || die "Can't open 'data.dat' for read: $!";
  # calculate byte address of block:record
  my $byteoffset = $block * 8192 + $blrec * $reclength;
  # move file pointer to new byte position with offset from start of file
  seek (DATA, $byteoffset, 0) || die "Can't seek to this position";
  # read data of the length of a record into a filestream
  read (DATA,my $filestream,$reclength) || die "Can't read";
  close DATA;
  return($filestream);
}

# READ DATA RECORD WITH PRIMARY KEY VALUE
sub readon_pindx {
  # PARAMETERS
  # INPUT: primary key value for read [0]
  # INPUT: primary key field definition [1]
  # INPUT: ordered(1)/unordered(0) primary index [2]
  # INPUT: address - 1 or record - 0 [3]
  # INPUT: table name [4]
  # OUTPUT: data record at address (scalar) - if input [3] = 0
  # OUTPUT: data address for index - if input [3] = 1
  my $pkey = $_[0];
  my $pkey_def = $_[1];
  my $ordered = $_[2];
  my $pkey_packed = pack($pkey_def,$pkey);
  my $tbl_nm = $_[4];
```

```
#   print "Pkey = $pkey, Pkey_def = $pkey_def, Order = $ordered, addres/rec
$_[3], Table = $tbl_nm <BR>";
  open (INDX,"+<index_" . $tbl_nm . ".dat") || die "Can't open 'index.dat' for
read/update: $!";
  # (-s INDX) = file size in bytes of INDX operator
  my $filesize = -s INDX;
  # create pack template for index - A10 size for address - 6 block, 4 rec *
8192 bytes
  my $pack_tpl = $pkey_def . " A10";
  # determine index length from template
  my $indxlen = length (pack($pack_tpl));
  # determine number of records in index (to determine # reads)
  my $numrecords = $filesize / $indxlen;
  # initialise variables for the loop
  my $numreads = 0;
  my $top_pos = $numrecords;
  my $bot_pos = 1;
  my $split_pos;
  my $address;
  my $stream;
  # search for index key
  for (my $i=1; $i<=$numrecords; $i++){
    # for unordered index
    if ($ordered == 0) {
      read INDX, $stream, $indxlen; # || die "Error reading index";
      # variable for number of reads required to get to desired record
      ++$numreads;
      if (substr($stream,0,length(pack($pkey_def))) =~ /$pkey/i) {
        $address = substr($stream,-10,10);
        return(read_data_add($address,$reclen)) if ($_[3] == 0);
        return($address) if ($_[3] == 1);
      }
    }
    # reading algorithm - depends on ordered index
    if ($ordered == 1) {
#      my $t1 = uc(substr($stream,0,length(pack($pkey_def))));
#      my $t2 = uc(pack($pkey_def,$pkey));
#      print "T1 = $t1, T2 = $t2 <BR>";
      if ($top_pos == $bot_pos) {
        seek(INDX, ($top_pos -1) * $indxlen, 0);
        read INDX, $stream, $indxlen; # || die "Error reading index";
```

```
    if          (uc(substr($stream,0,length(pack($pkey_def))))          ne
uc(pack($pkey_def,$pkey))) {
      return("ERROR: not key value");
      exit;
    }
    $address = substr($stream,-10,10);
    # gives indication of number of reads
    $numreads++;
    return(read_data_add($address,$reclen)) if ($_[3] == 0);
    return($address) if ($_[3] == 1);
  }
  # use integer type division
  use integer;
  $split_pos = ($top_pos - $bot_pos)/2 + $bot_pos;
  seek(INDX, ($split_pos * $indxlen), 0);
  read INDX, $stream, $indxlen; # || die "Error reading index";
  my $read_pk = substr($stream,0,length(pack($pkey_def)));
  ++$numreads;
  if (($top_pos - $bot_pos) == 1) {
    if ($bot_pos == 1) {
      seek(INDX, 0, 0);
      read INDX, $stream, $indxlen; # || die "Error reading index";
      my $read_pk = substr($stream,0,length(pack($pkey_def)));
      if (uc($read_pk) ne uc($pkey_packed)){
        seek(INDX, $indxlen, 0);
        read INDX, $stream, $indxlen; # || die "Error reading index";
        my $read_pk = substr($stream,0,length(pack($pkey_def)));
      }
    }
    $read_pk = uc(substr($stream,0,length(pack($pkey_def))));
    if (uc($read_pk) ne uc($pkey_packed)) {
      return("ERROR: not key value");
      exit;
    }
    $address = substr($stream,-10,10);
    # read data from address indicated by index
    return(read_data_add($address,$reclen)) if ($_[3] == 0);
    return($address) if ($_[3] == 1);
  }
  # read record is smaller than $pkey_packed (A < Z)
  if ((uc($read_pk) cmp uc($pkey_packed)) == 1) {
    $top_pos = $split_pos;
```

```
      }
      # read record is larger than $pkey_packed (Z > A)
      if ((uc($read_pk) cmp uc($pkey_packed)) == -1) {
        $bot_pos = $split_pos;
      }
      if ((uc($read_pk) cmp uc($pkey_packed)) == 0) {
        $address = substr($stream,-10,10);
        # read data from address indicated by index
        return(read_data_add($address,$reclen)) if ($_[3] == 0);
        return($address) if ($_[3] == 1);
      }
    }
  }
  close INDX;
}

# READ DATA BY SCANNING THROUGH TABLE ON INDEX
sub read_scan{
  # PARAMETERS
  # INPUT: field value [0]
  # INPUT: field size [1]
  # INPUT: field order in record [2]
  # INPUT: primary key field definition [3]
  # INPUT: exact (1) or match (0) value [4]
  # INPUT: packing template [5]
  # INPUT: table name [6]
  my $field_val = $_[0];
  my $field_def = $_[1];
  my $field_ord = $_[2];
  my $pkey_def = $_[3];
  my $ex_ma = $_[4];
  my $pack_tpl = $_[5];
  my $reclength = length(pack($pack_tpl));
  my $tbl_nm = $_[6];
  my %temp = read_prim_index($pkey_def,$tbl_nm);
  my @addresses = values %temp;
  # sort address to make reads more sequential
  # instead of jumping from address to address
  @addresses = sort @addresses;
  my $count = 0;
  my @tempb;
  my $temp;
```

```
  my @all_tab = ();
  my $count_tab = 0;
  # Now @lines holds all the lines, one line in each element.
  foreach my $line(@lines) {
    # split colon separated values in the line
    my @values = split (/;/,$line);
    if ($tablenm eq '*'){
      # skip header line in repository
      push(@all_tab,$values[0]) unless ($count_tab == 0);
    }
    else {
      #checks if table (first entry) is = $tablenm
      return(@values) if ( $values[0] eq $tablenm );
    }
    $count_tab++;
  }
  return(@all_tab);
}

# READ DATA FROM DATA ADDRESS
sub read_data_add {
  # INPUT: address - from index, block & rec format
  # OUTPUT: record, packed format
  my $block = substr($_[0],0,6);
  my $blrec = substr($_[0],-4,4);
  open (DATA,"+<data.dat") || die "Can't open 'data.dat' for read: $!";
  # calculate byte address of block:record
  my $byteoffset = $block * 8192 + $blrec * $reclen;
  # move file pointer to new byte position with offset from start of file
  seek (DATA, $byteoffset, 0) || die "Can't seek to this position";
  # read data of the length of a record into a filestream
  read (DATA,my $filestream,$reclen) || die "Can't read";
  close DATA;
  return($filestream);
}

# READ DATA RECORD WITH PRIMARY KEY VALUE
sub readon_pindx {
  # PARAMETERS
  # INPUT: primary key value for read [0]
  # INPUT: primary key field definition [1]
  # INPUT: ordered(1)/unordered(0) primary index [2]
```

```
  # OUTPUT: data record at address - scalar
  my $pkey = $_[0];
  my $pkey_def = $_[1];
  my $ordered = $_[2];
  my $pkey_packed = pack($pkey_def,$pkey);
  open (INDX,"+<index_" . $tablename . ".dat") || die "Can't open 'index.dat'
for read/update: $!";
  # (-s INDX) = file size in bytes of INDX operator
  my $filesize = -s INDX;
  # create pack template for index - A10 size for address - 6 block, 4 rec *
8192 bytes
  my $pack_tpl = $pkey_def . " A10";
  # determine index length from template
  my $indxlen = length (pack($pack_tpl));
  # determine number of records in index (to determine # reads)
  my $numrecords = $filesize / $indxlen;
  # initialise variables for the loop
  my $numreads = 0;
  my $top_pos = $numrecords;
  my $bot_pos = 1;
  my $split_pos;
  my $address;
  my $stream;
  # search for index key
  for (my $i=1; $i<=$numrecords; $i++){
    # for unordered index
    if ($ordered == 0) {
      read INDX, $stream, $indxlen; # || die "Error reading index";
      # variable for number of reads required to get to desired record
      ++$numreads;
      if (substr($stream,0,length(pack($pkey_def))) =~ /$pkey/i) {
        $address = substr($stream,-10,10);
        return(read_data_add($address));
      }
    }
    # reading algorithm - depends on ordered index
    if ($ordered == 1) {
      if ($top_pos == $bot_pos) {
        seek(INDX, ($top_pos -1) * $indxlen, 0);
        read INDX, $stream, $indxlen; # || die "Error reading index";
        if            (uc(substr($stream,0,length(pack($pkey_def))))            ne
uc(pack($pkey_def,$pkey))) {
```

```
    return("ERROR: not key value");
    exit;
  }
  $address = substr($stream,-10,10);
  # gives indication of number of reads
  $numreads++;
  return(read_data_add($address));
}
# use integer type division
use integer;
$split_pos = ($top_pos - $bot_pos)/2 + $bot_pos;
seek(INDX, ($split_pos * $indxlen), 0);
read INDX, $stream, $indxlen; # || die "Error reading index";
my $read_pk = substr($stream,0,length(pack($pkey_def)));
++$numreads;
if (($top_pos - $bot_pos) == 1) {
  if ($bot_pos == 1) {
    seek(INDX, 0, 0);
    read INDX, $stream, $indxlen; # || die "Error reading index";
    my $read_pk = substr($stream,0,length(pack($pkey_def)));
    if (uc($read_pk) ne uc($pkey_packed)){
      seek(INDX, $indxlen, 0);
      read INDX, $stream, $indxlen; # || die "Error reading index";
      my $read_pk = substr($stream,0,length(pack($pkey_def)));
    }
  }
  $read_pk = uc(substr($stream,0,length(pack($pkey_def))));
  if (uc($read_pk) ne uc($pkey_packed)) {
    return("ERROR: not key value");
    exit;
  }
  $address = substr($stream,-10,10);
  # read data from address indicated by index
  return(read_data_add($address))
}
# read record is smaller than $pkey_packed (A < Z)
if ((uc($read_pk) cmp uc($pkey_packed)) == 1) {
  $top_pos = $split_pos;
}
# read record is larger than $pkey_packed (Z > A)
if ((uc($read_pk) cmp uc($pkey_packed)) == -1) {
  $bot_pos = $split_pos;
```

```
      }
      if ((uc($read_pk) cmp uc($pkey_packed)) == 0) {
        $address = substr($stream,-10,10);
        # read data from address indicated by index
        return(read_data_add($address));
      }
    }
  }
  close INDX;
}

# GET REPOSITORY INFORMATION
sub get_rep_info {
  # INPUT - table name - [0]
  @tbl_info = read_rep_table($_[0]);

  # Array structure for repository as returned
  # 0 TABLE
  # 1 FIELDS (separated by a space)
  # 2 SIZE (of record - pack format)
  # 3 PK (field name of primary key(s))
  # 4 RELATION (information for setting up relations)

  # determine record pack template
  $rec_PT = $tbl_info[2];
  # determine record length from template
  $reclen = length (pack($rec_PT));
  # read the field names
  @field_names = split(/ /, $tbl_info[1]);
  @field_defs = split(/ /, $tbl_info[2]);
  # create associative array with field name as index
  # and field size/template as associated field
  @fields{@field_names}=@field_defs;
  # determine number of fields
  $numfields = @field_names;
}

# FIND A VALUE IN A LIST
sub list_index {
  my ($value, @list) = @_;
  my $count = 0;
  foreach my $i(@list){
```

```
    return($count) if ($i eq $value);
    $count++;
  }
}

my $query = new CGI;
my $cookiename = "login_ticket";
my $checkcookie = cookie($cookiename);

# TABLE NAME
$tablename = "USERS";
my @record_val;
# get repository information
get_rep_info($tablename);
my $username = $query->param('uname');
my $password = $query->param('upass');

if ($username) {}
else {$username = "NULL";}
if ($password) {}
else {$password = "NULL";}

# READ ON PRIMARY INDEX
my $result = readon_pindx($username,$fields{$tbl_info[3]},1);
my $error_image = "<img border=\"0\" src=\"..\\images\\error.gif\">";
if ($result =~ /error/i) {
print header;
print
  start_html('Login failed'), # start the HTML
  hr,
  table({-border=>0},
         Tr({-align=>'RIGHT',-valign=>'TOP'},
         [
            td([$error_image , h1('Login failed, the username does not
exist')])
         ]
         )
      ),
  hr;
}
else
{
```

```
  @record_val = unpack($rec_PT,$result);
  if ($password eq $record_val[1]){
    # Password correct - save session cookie
    my $cookie = cookie(
    -name=> $cookiename,
    -value=> $username
#   ,-expires=> '+1h'    # if ommitted, stored in memory
    );
    print header(-COOKIE => $cookie, -expires=>'now');
    print
      start_html('Login successful'), # start the HTML
      h1('Launching screen'),         # level 1 header
      hr;
    my @tbl_list = read_rep_table('*');
    print
      start_form(-action=>'dbms4.pl');
    print "Select table:";
    print $query->popup_menu(-name=>'table_name',-values=>[@tbl_list]);
    print "<BR><BR>";
    print
      submit(-name=>'sub',-value=>'Read'),
      submit(-name=>'sub',-value=>'Insert'),
      submit(-name=>'sub',-value=>'Update'),
      submit(-name=>'sub',-value=>'Delete'),
      submit(-name=>'sub',-value=>'Delete all related');
    print "<BR>";
    print
      hr,
      endform;
  }
  else {
  print
    header,
    start_html('Login failed'), # start the HTML
    hr,
    table({-border=>0},
           Tr({-align=>'RIGHT',-valign=>'TOP'},
           [
              td([$error_image , h1('Login failed, password incorrect')])
           ]
           )
        ),
```

```
    hr;
  }
}
end_html;                        # end the HTML
```

DBMS4.pl

```
#!C:/Perl/bin/perl
use strict;
#use warnings;
use CGI qw/:standard *table start_ul/;

# GLOBAL VARIABLES USED WITHIN THE PROGRAM
# Table name
my $tablename;
# Table information as returned
my @tbl_info;
# Primary key definition
my $pkdef;
# Record length
my $reclen;
# Packing template
my $rec_PT;
# Field names
my @field_names;
# Field definitions
my @field_defs;
# Fields
my %fields;
# Number of fields
my $numfields;
# Primary key type
my @pkey_type;
# Child tables
my @c_tables;
# Parent tables
my @p_tables;
# Participation (parent entry required)
my @participate;


# READ REPOSITORY FILE/TABLE
```

```
sub read_rep_table {
  # INPUT: table name - table info to be read
  my $tbl_nm=$_[0];
  # open file for reading
  open (REPOS,"+<repository.dat");
  my @lines = <REPOS>;
  close REPOS;
  # Now @lines holds all the lines, one line in each element.
  foreach my $line(@lines) {
  # split colon separated values in the line
    my @values = split (/;/,$line);
    #checks if table (first entry) is = $tbl_nm
    return(@values) if ( $values[0] eq $tbl_nm );
  }
}

# GET REPOSITORY INFORMATION
sub get_rep_info {
  # INPUT - table name - [0]
  @tbl_info = read_rep_table($_[0]);

  # Array structure for repository as returned
  # 0 TABLE
  # 1 FIELDS (separated by a space)
  # 2 SIZE (of record - pack format)
  # 3 PK (field name of primary key(s))
  # 4 Primary key type (ARTIF - artificial (auto numbered)/CUSTOM - arbitrary
  # 5 Child tables
  # 6 Parent tables
  # 7 Participation - O -> optional; M -> mandatory

  # determine record pack template
  $rec_PT = $tbl_info[2];
  # determine record length from template
  $reclen = length (pack($rec_PT));
  # read the field names
  @field_names = split(/ /, $tbl_info[1]);
  @field_defs = split(/ /, $tbl_info[2]);
  # create associative array with field name as index
  # and field size/template as associated field
  @fields{@field_names}=@field_defs;
  # determine number of fields
```

```
  $numfields = @field_names;
  @pkey_type = split(/ /,$tbl_info[4]);
  @c_tables = split(/ /,$tbl_info[5]);
  @p_tables = split(/ /,$tbl_info[6]);
  @participate =split(/ /,$tbl_info[7]);
}


# READ DATA FROM DATA ADDRESS
sub read_data_add {
  # INPUT: address - from index, block & rec format [0]
  # INPUT: record length [1]
  # OUTPUT: record, packed format
  my $block = substr($_[0],0,6);
  my $blrec = substr($_[0],-4,4);
  my $reclength = $_[1];
  open (DATA,"+<data.dat") || die "Can't open 'data.dat' for read: $!";
  # calculate byte address of block:record
  my $byteoffset = $block * 8192 + $blrec * $reclength;
  # move file pointer to new byte position with offset from start of file
  seek (DATA, $byteoffset, 0) || die "Can't seek to this position";
  # read data of the length of a record into a filestream
  read (DATA,my $filestream,$reclength) || die "Can't read";
  close DATA;
  return($filestream);
}

# READ DATA RECORD WITH PRIMARY KEY VALUE
sub readon_pindx {
  # PARAMETERS
  # INPUT: primary key value for read [0]
  # INPUT: primary key field definition [1]
  # INPUT: ordered(1)/unordered(0) primary index [2]
  # INPUT: address - 1 or record - 0 [3]
  # INPUT: table name [4]
  # OUTPUT: data record at address (scalar) - if input [3] = 0
  # OUTPUT: data address for index - if input [3] = 1
  my $pkey = $_[0];
  my $pkey_def = $_[1];
  my $ordered = $_[2];
  my $pkey_packed = pack($pkey_def,$pkey);
  my $tbl_nm = $_[4];
```

```
#   print "Pkey = $pkey, Pkey_def = $pkey_def, Order = $ordered, addres/rec
$_[3], Table = $tbl_nm <BR>";
  open (INDX,"+<index_" . $tbl_nm . ".dat") || die "Can't open 'index.dat' for
read/update: $!";
  # (-s INDX) = file size in bytes of INDX operator
  my $filesize = -s INDX;
  # create pack template for index - A10 size for address - 6 block, 4 rec *
8192 bytes
  my $pack_tpl = $pkey_def . " A10";
  # determine index length from template
  my $indxlen = length (pack($pack_tpl));
  # determine number of records in index (to determine # reads)
  my $numrecords = $filesize / $indxlen;
  # initialise variables for the loop
  my $numreads = 0;
  my $top_pos = $numrecords;
  my $bot_pos = 1;
  my $split_pos;
  my $address;
  my $stream;
  # search for index key
  for (my $i=1; $i<=$numrecords; $i++){
    # for unordered index
    if ($ordered == 0) {
      read INDX, $stream, $indxlen; # || die "Error reading index";
      # variable for number of reads required to get to desired record
      ++$numreads;
      if (substr($stream,0,length(pack($pkey_def))) =~ /$pkey/i) {
        $address = substr($stream,-10,10);
        return(read_data_add($address,$reclen)) if ($_[3] == 0);
        return($address) if ($_[3] == 1);
      }
    }
    # reading algorithm - depends on ordered index
    if ($ordered == 1) {
#      my $t1 = uc(substr($stream,0,length(pack($pkey_def))));
#      my $t2 = uc(pack($pkey_def,$pkey));
#      print "T1 = $t1, T2 = $t2 <BR>";
      if ($top_pos == $bot_pos) {
        seek(INDX, ($top_pos -1) * $indxlen, 0);
        read INDX, $stream, $indxlen; # || die "Error reading index";
```

```
    if              (uc(substr($stream,0,length(pack($pkey_def))))              ne
uc(pack($pkey_def,$pkey))) {
      return("ERROR: not key value");
      exit;
    }
    $address = substr($stream,-10,10);
    # gives indication of number of reads
    $numreads++;
    return(read_data_add($address,$reclen)) if ($_[3] == 0);
    return($address) if ($_[3] == 1);
  }
  # use integer type division
  use integer;
  $split_pos = ($top_pos - $bot_pos)/2 + $bot_pos;
  seek(INDX, ($split_pos * $indxlen), 0);
  read INDX, $stream, $indxlen; # || die "Error reading index";
  my $read_pk = substr($stream,0,length(pack($pkey_def)));
  ++$numreads;
  if (($top_pos - $bot_pos) == 1) {
    if ($bot_pos == 1) {
      seek(INDX, 0, 0);
      read INDX, $stream, $indxlen; # || die "Error reading index";
      my $read_pk = substr($stream,0,length(pack($pkey_def)));
      if (uc($read_pk) ne uc($pkey_packed)){
        seek(INDX, $indxlen, 0);
        read INDX, $stream, $indxlen; # || die "Error reading index";
        my $read_pk = substr($stream,0,length(pack($pkey_def)));
      }
    }
    $read_pk = uc(substr($stream,0,length(pack($pkey_def))));
    if (uc($read_pk) ne uc($pkey_packed)) {
      return("ERROR: not key value");
      exit;
    }
    $address = substr($stream,-10,10);
    # read data from address indicated by index
    return(read_data_add($address,$reclen)) if ($_[3] == 0);
    return($address) if ($_[3] == 1);
  }
  # read record is smaller than $pkey_packed (A < Z)
  if ((uc($read_pk) cmp uc($pkey_packed)) == 1) {
    $top_pos = $split_pos;
```

```
      }
      # read record is larger than $pkey_packed (Z > A)
      if ((uc($read_pk) cmp uc($pkey_packed)) == -1) {
        $bot_pos = $split_pos;
      }
      if ((uc($read_pk) cmp uc($pkey_packed)) == 0) {
        $address = substr($stream,-10,10);
        # read data from address indicated by index
        return(read_data_add($address,$reclen)) if ($_[3] == 0);
        return($address) if ($_[3] == 1);
      }
    }
  }
  close INDX;
}

# READ DATA BY SCANNING THROUGH TABLE ON INDEX
sub read_scan{
  # PARAMETERS
  # INPUT: field value [0]
  # INPUT: field size [1]
  # INPUT: field order in record [2]
  # INPUT: primary key field definition [3]
  # INPUT: exact (1) or match (0) value [4]
  # INPUT: packing template [5]
  # INPUT: table name [6]
  my $field_val = $_[0];
  my $field_def = $_[1];
  my $field_ord = $_[2];
  my $pkey_def = $_[3];
  my $ex_ma = $_[4];
  my $pack_tpl = $_[5];
  my $reclength = length(pack($pack_tpl));
  my $tbl_nm = $_[6];
  my %temp = read_prim_index($pkey_def,$tbl_nm);
  my @addresses = values %temp;
  # sort address to make reads more sequential
  # instead of jumping from address to address
  @addresses = sort @addresses;
  my $count = 0;
  my @tempb;
  my $temp;
```

```
        @field_dfn = split(/ /, $rep_info[2]);
        $pkey_nms[$i] = $rep_info[3];
        $ord[$i] = list_index ($pkey_nms[$i-1],@field_nms);
        my @temp = ();
        @ret_recs                                                              =
read_scan($fkey,$field_dfn[$ord[$i]],$ord[$i],$field_dfn[0],0,$rep_info[2],$tbl
);
        foreach (@ret_recs) {
          if (defined($new_pk)) {
            $new_pk = $new_pk . " " . unpack($field_dfn[0],$_);
          }
          else {
            $new_pk = unpack($field_dfn[0],$_);
          }
        }
        $foreign_kflat    =    $foreign_kflat    .    "    "    .    $new_pk    if
defined($foreign_kflat);
        $foreign_kflat = $new_pk unless defined($foreign_kflat);
        $new_pk = undef;
      }
      my @chg_kflat = split(/ /,$foreign_kflat);
      @chg_kflat = splice(@chg_kflat,$count_keys);
      $foreign_table[$k] = $tbl if (@chg_kflat);
      $foreign_def[$k] = $field_dfn[0] if (@chg_kflat);
      $foreign_keys[$k] = undef;
      if (@chg_kflat) {
        foreach (@chg_kflat) {
          $foreign_keys[$k]    =    $foreign_keys[$k]    .    "    "    .$_    if
defined($foreign_keys[$k]);
          $foreign_keys[$k] = $_ unless defined($foreign_keys[$k]);
        }
      }
     # print "*DELETING HERE $foreign_keys[$k] in table $foreign_table[$k] type
$foreign_def[$k]*<BR>" if ($foreign_keys[$k]);
      my @to_del = split(/ /, $foreign_keys[$k]);
      my $z = 0;
      foreach (@to_del) {

delete_rec($_,$foreign_def[$k],$foreign_table[$k],length(pack($rep_info[2])))
unless ($_ eq "");
      }
      $k++ if (@chg_kflat);
```

```
    }
    # print deleted key names, if they exist
    if (@foreign_keys) {
      my $l = 0;
      foreach (@foreign_keys) {
        print "$foreign_table[$l++] -> $_<BR>\n" unless ($_ eq "");
      }
    }
  }
}

#==================================================================#
#==================================================================#
# START OF WEB PAGES
#==================================================================#
#==================================================================#

my $query = new CGI;
my $cookiename = "login_ticket";
my $checkcookie = cookie($cookiename);

# Check cookie:
if ($checkcookie) {
  print header(-expires=>'now');
  # Cookie checked ok
}
else {
  print $query->redirect('http://localhost:90/login_form.htm');
  exit;
}

my $break = "<BR>";

# TABLE NAME
$tablename = $query->param('table_name');
# get repository information
get_rep_info($tablename);
#print "Table: $tablename $break";

#==================================================================#
```

```
# Initial page
#==================================================================#
if    (!$query->param('sub')    &&    !$query->param('sub_read')    &&    !$query-
>param('status')){
print
  start_html('Start page'), # start the HTML
  h1('Start'),         # level 1 header
  hr,
  param('login'),
  hr,
  h2('You have not been logged in. Please return to the login page');
}

#==================================================================#
# Read web pages
#==================================================================#
# Read data page
if ($query->param('sub') eq "Read"){
  print
    start_html('Read data'), # start the HTML
    hr,
    h1('Read'),         # level 1 header
    hr,
    "Table:", $query->param('table_name'), $break,
    start_form,
    hidden(-name=>'table_name',-value=>$query->param('table_name')),
    "Read by primary field", $break,
    table({-border=>1},
      Tr({-align=>'LEFT',-valign=>'TOP'},
      [
       th([$tbl_info[3]]),
       td([textfield(-name=>'prim_field',-size=>length(pack($field_defs[0])),-
maxlength=>length(pack($field_defs[0])))])
      ]
      )
    );
  print "or by other field value:" . $break;
  print $query->popup_menu(-name=>'rec_fields',
  -values=>[@field_names]);
  print                    textfield(-name=>'scan_field',-size=>largest_field,-
maxlength=>largest_field);
```

```
  print $break;
  print $query->radio_group(-name=>'read_exma',
  -values=>['exact','pattern'],
  -default=>'exact',
  -linebreak=>'true');
  print
    $break,
    submit(-name=>'sub_read',-value=>'Read data'),
    $break,
    endform;
}

# Page display for data read results
if ($query->param('sub_read')){
  print
    start_html('Data has been read'), # start the HTML
    hr,
    h1('Read results'),
    hr, "Table:", $query->param('table_name'), $break;
  my $matching;
  if ($query->param('read_exma') eq "exact") {
    # exact matching - function returns fields on primary key
    my                       $rec_stream                          =readon_pindx($query-
>param('prim_field'),$fields{$tbl_info[3]},1,0,$query->param('table_name'));
    if ($rec_stream =~ /error/i) {
        print "No such key value. Please try again.";
    }
    else {
      my @record_val = unpack($rec_PT,$rec_stream);
      my $num_ret = @record_val;
      print "<table border='1' cellpadding='0' cellspacing='0'>";
      print "<tr>";
      # headers then field values
      for (my $i=0; $i<=$num_ret; $i++) {
        print "<th>" . $field_names[$i] . "</th>";
      }
      print "<tr></tr>";
      foreach (@record_val){
        print "<td>" . $_ . "</td>";
      }
      print "</tr></table>";
    }
```

```
  }
  else {
    # pattern matching - function returns fields on other entries
    my    @records    =    read_scan($query->param('scan_field'),$fields{$query-
>param('rec_fields')},list_index($query-
>param('rec_fields'),@field_names),$fields{$tbl_info[3]},0,$rec_PT,$query-
>param('table_name'));
    if (@records) {
      print "<table border='1' cellpadding='0' cellspacing='0'>";
    }
    my $i=0;
    foreach (@records) {
      print "<tr>";
      if ($i==0) {
        foreach (@field_names) {
          print "<th>$_</th>";
        }
        print "</tr><tr>";
      }
      my @record=unpack($rec_PT,$_);
      foreach (@record){
        print "<td>$_</td>";
      }
      print "</tr>";
      $i++;
    }
    if ($i==0) {
      print "No value/record found matching your selection";
    }
    print "</table>";
  }
}

#==================================================================#
# Insert web pages
#==================================================================#

# INSERT DATA PAGE
if ($query->param('sub') eq "Insert"){
  print
    start_html('Insert'), # start the HTML
    h1('Insert record'),
```

```
    hr,
    "Table:", $query->param('table_name'), $break,
    start_form,
    hidden(-name=>'table_name',-value=>$query->param('table_name'));
  print "<table border='1' cellpadding='0' cellspacing='0'>";
  print "<tr>";
  my $idx;
  if ($pkey_type[0] eq "ARTIF") {
    $idx = read_last_key($field_defs[0],$query->param('table_name'));
    $idx = substr($idx,1,(length($idx)-1));
    my $t_len = length(pack($field_defs[0])) - length($pkey_type[1]);
    if ($idx == 0) {
      $idx = "0" x ($t_len - 1) . "1";
    }
    else {
      $idx = $idx + 1;
      $idx = "0" x ($t_len - length($idx)) . $idx;
    }
    $idx = $pkey_type[1] . $idx;
  }
  foreach (@field_names) {
    print "<th>$_</th>";
  }
  print "</tr><tr>";
  my $k = 0;
  foreach (@field_defs) {
    print "<td>";
    if (($k == 0) && ($pkey_type[0] eq "ARTIF")) {
      print $idx;
      print hidden(-name=>'field0',-value=>$idx);
    }
    else {
      print              textfield(-name=>('field'.$k),-size=>length(pack($_)),-
maxlength=>length(pack($_)));
    }
    $k++;
    print "</td>";
  }
  print "</tr></table>";
  print
    submit(-name=>'sub',-value=>'Insert data'),
    endform;
```

```
}

# Show results of data insert after committing function
if ($query->param('sub') eq "Insert data"){
  print
    start_html('Insert a new record'), # start the HTML
    h1('Inserting of a record'),        # level 1 header
    hr,"Table:", $query->param('table_name'), $break;
  my @params = $query->param;
  my @f_vals;
  my $i=0;
  foreach (@params) {
    if ($_ =~ /field/i) {
      $f_vals[$i++] = $query->param($_);
    }
  }
#  foreach (@f_vals) {
#    print "$_ $break";
#  }
  insert_rec($field_defs[0],pack($rec_PT,@f_vals),$query->param('table_name'));
}


#==================================================================#
# Update web pages
#==================================================================#
# UPDATE DATA PAGE - FIRST READ
if ($query->param('sub') eq "Update") {
print
  start_html('Read data for update'), # start the HTML
  hr,
  h1('Update'),           # level 1 header
  hr, "Table:", $query->param('table_name'), $break,
  start_form,
  hidden(-name=>'table_name',-value=>$query->param('table_name')),
  "Read by primary field", $break,
  table({-border=>1},
    Tr({-align=>'LEFT',-valign=>'TOP'},
    [
     th([$tbl_info[3]]),
     td([textfield(-name=>'prim_field',-size=>length(pack($field_defs[0])),-
maxlength=>length(pack($field_defs[0]))])])
```

```
    ]
    )
  );
print
  $break,
  submit(-name=>'sub_upd_read',-value=>'Read data'),
  $break,
  hidden(-name=>'status',-value=>'Busy'),
  endform;
}

# UPDATE DATA PAGE - Read results
if ($query->param('sub_upd_read') eq "Read data") {
  print
    start_html('Data has been read'), # start the HTML
    hr,
    h1('Read results'),
    hr, "Table:", $query->param('table_name'), $break;
  # exact matching - function returns fields on primary key
  my                       $rec_stream                       =readon_pindx($query-
>param('prim_field'),$fields{$tbl_info[3]},1,0,$query->param('table_name'));
  if ($rec_stream =~ /error/i) {
      print "No such key value. Please try again.";
  }
  else {
    my @record_val = unpack($rec_PT,$rec_stream);
    my $num_ret = @record_val;
    print start_form;
    print hidden(-name=>'table_name',-value=>$query->param('table_name'));
    print "<table border='1' cellpadding='0' cellspacing='0'>";
    print "<tr>";
    # headers then field values
    for (my $i=0; $i<=$num_ret; $i++) {
      print "<th>" . $field_names[$i] . "</th>";
    }
    print "<tr></tr><tr>";
    my $k = 0;
    foreach (@field_defs) {
      print "<td>";
      if ($k==0) {
        print $record_val[$k];
        print $query->hidden(-name=>'field0',-default=>$record_val[$k]);
```

```
      }
      else {
        print textfield(-name=>('field'.$k),
             -size=>length(pack($_)),
             -default=>$record_val[$k],
             -maxlength=>length(pack($_)));
      }
      $k++;
      print "</td>";
    }
    print "</tr></table>";
    print hidden(-name=>'status',-value=>'Busy');
    print
      $break,
      submit(-name=>'sub_update',-value=>'Update data'),
      $break,
      endform;
  }
}

# UPDATE DATA PAGE - Update results
if ($query->param('sub_update') eq "Update data") {
  print
    start_html('Update'), # start the HTML
    hr,
    h1('Update results'),
    hr, "Table:", $query->param('table_name'), $break;
  my @params = $query->param;
  my @f_vals;
  my $i=0;
  foreach (@params) {
    if ($_ =~ /field/i) {
      $f_vals[$i++] = $query->param($_);
    }
  }
  my $record_str = pack($rec_PT,@f_vals);
  update_rec($f_vals[0],$field_defs[0],$record_str,$query-
>param('table_name'));
}

#===================================================================#
# Delete web pages
```

```
#====================================================================#
# DELETE DATA PAGE - FIRST READ
if ($query->param('sub') eq "Delete") {
print
  start_html('Delete'), # start the HTML
  hr,
  h1('Enter record to delete'),          # level 1 header
  hr, "Table:", $query->param('table_name'), $break,
  start_form,
  hidden(-name=>'table_name',-value=>$query->param('table_name')),
  "Enter the primary field value for this record", $break,
  table({-border=>1},
    Tr({-align=>'LEFT',-valign=>'TOP'},
    [
     th([$tbl_info[3]]),
     td([textfield(-name=>'prim_field',-size=>length(pack($field_defs[0])),-
maxlength=>length(pack($field_defs[0])))])
    ]
    )
  );
print
  $break,
  submit(-name=>'sub',-value=>'Delete record'),
  $break,
  hidden(-name=>'status',-value=>'Busy'),
  endform;
}

# DELETION ACTION AND RESULTS
if ($query->param('sub') eq "Delete record") {
  print
    start_html('Data has been deleted'), # start the HTML
    hr,
    h1('Delete results'),
    hr, "Table:", $query->param('table_name'), $break;
  my                $r_code                =                delete_rec($query-
>param('prim_field'),$fields{$tbl_info[3]},$query-
>param('table_name'),$reclen);
  print $r_code
}

# CASCADED DELETE PAGE
```

```
if ($query->param('sub') eq "Delete all related") {
print
  start_html('Delete cascade'), # start the HTML
  hr,
  h1('Enter record to delete'),          # level 1 header
  hr,"Table:", $query->param('table_name'), $break,
  start_form,
  hidden(-name=>'table_name',-value=>$query->param('table_name')),
 "Enter the primary field value for this record", $break,
  table({-border=>1},
    Tr({-align=>'LEFT',-valign=>'TOP'},
    [
     th([$tbl_info[3]]),
     td([textfield(-name=>'prim_field',-size=>length(pack($field_defs[0])),-
maxlength=>length(pack($field_defs[0])))])
    ]
    )
  );
print
  $break,
  submit(-name=>'sub',-value=>'Delete cascade'),
  $break,
  hidden(-name=>'status',-value=>'Busy'),
  endform;
}

# DELETION ACTION AND RESULTS FOR CASCADED DELETE
if ($query->param('sub') eq "Delete cascade") {
  print
    start_html('Data has been deleted'), # start the HTML
    hr,
    h1('Delete results'),
    hr, "Table:", $query->param('table_name'), $break;
    pc_entries($query->param('table_name'),$query->param('prim_field'));
}

print end_html;                   # end the HTML
```