A

Data-Management System

For

Relational Data Bases

By

David Harvey Scuse

A Thesis

Submitted to the Faculty of Graduate Studies

of the University of Manitoba

in partial fulfillment of the requirements

for the degree of

Doctor of Philosophy

Department of Computer Science

University of Manitoba

Winnipeg, Manitoba

May 1979

A DATA-MANAGEMENT SYSTEM FOR RELATIONAL DATA BASES

BY

DAVID HARVEY SCUSE

A dissertation submitted to the Faculty of Graduate Studies of the University of Manitoba in partial fulfillment of the requirements of the degree of

DOCTOR OF PHILOSOPHY

Abstract

In this thesis, we define a data-management system which provides efficient data-management facilities in a changing environment. The data-management system is designed to support the relational view of data. The system provides several facilities that are not found in current data-management systems, both relational and non-relational.

The data-management system is developed in four independent subsystems. The device subsystem manipulates the pages on which the components of a relation are stored. The pages can be stored, permanently and temporarily, on a hierarchy of storage devices. The tuples of a relation are stored on logical pages which the storage subsystem maps onto the physical pages manipulated by the device subsystem. The storage subsystem provides the capability to roll back the contents of a relation by maintaining multiple copies of tuples. In the access-path subsystem, a powerful access path, the multiple-relation access path, is used to determine where tuples with given characteristics are stored. A data-manipulation language which provides associative access to tuples is supported by the retrieval subsystem. The retrieval subsystem translates the associative requests into the necessary requests to the access-path subsystem.

Acknowledgements

I would like to thank Professors R. G. Stanton and C. R. Zarnke for giving their time so freely in supervising both this thesis and the related research. I would also like to thank Professors D. D. Cowan, R. S. D. Thomas, and M. S. Doyle for the time spent reading the thesis and discussing improvements.

Finally, the financial assistance of the National Research Council of Canada during the preparation of this thesis is gratefully acknowledged.

To Barbara

Table of Contents

Chapter 1: Data-Management Systems

1.1 Introduction

During the past 20 years, there has been a major change in the data-management software provided for computer users. Until recently, very primitive data-management software was provided and, frequently, the user wrote his own data-management routines; but, as the amount of data increased and the underlying data structure became more complex, the user had to write more sophisticated software. Gradually, it was realized that it should not be the users' responsibility to provide data-management software; such software should be part of the operating system with which the user interacts. In this chapter, we examine the growth of data management from basic record-oriented access to complex data base management systems which provide powerful data-management facilities.

1.2 Data Management

The purpose of data-management systems is to manage large amounts of data. By large, we mean that there is more data to be processed than can conveniently be stored in main memory while the data are being processed. If this were not true, the data could be processed using standard in-core techniques. Thus, we assume that only a small portion of

the data to be processed can be stored in main memory at a given time; the rest of the data are stored on a secondary storage device. When necessary, the data-management routines transfer portions, which are referred to as "pages", of the data between the secondary storage device and main memory.

A collection of pages stored on a secondary storage device is referred to as a "data set". Normally, many data sets are stored on each device. The pages processed by a user are collectively referred to as a "file". A data set is a physical entity and a file is a logical entity. In basic data-management systems, each file is stored in one data set and each data set contains only one file. However, in the more complicated data-management systems, the pages in a file may be stored on several data sets and each data set may contain pages from more than one file.

A "physical record" is the amount of data stored on a secondary storage device without any intervening device timing/synchronization control information. In basic data-management systems, a page consists of one physical record but in more complicated systems, a page may consist of several physical records. A "logical record" is the amount of data required by the programmer. The data-management system extracts logical records from the pages and returns the logical records to the programmer.

Currently, the real time required to transfer a page between secondary storage and main memory is several orders of magnitude greater than the time required to process the page. For example, on an IBM 3330 disk drive, approximately 30.0 milliseconds are required to move the access arm of the disk to the required cylinder, approximately 8.4 milliseconds are required for the required page to rotate under the access arm, and then approximately 5.0 milliseconds are required to transfer a 4096-byte page to main memory [IBM74b]. This average of 43.4 milliseconds is in contrast to the main memory cycle time of only 115 nanoseconds (.000115 milliseconds) on an IBM 370/158 [IBM74a]. Thus, as the size of data files grows from million-character files towards billion-character files, it becomes increasingly important that the number of pages transferred to main memory in order to process requests be as small as possible. Executing extra instructions in main memory is usually justified if it causes the number of pages transferred to be reduced.

Records can be accessed by "address" or by "key". The address of a record is a numeric value which identifies the record by its position within the file. The key of a record is a set of characters which identify the record by value instead of by position. There are two basic types of keys: "primary keys" and "secondary keys". A primary key is a key

which uniquely identifies each record within a file and whose value is normally used in determining the position of the record within the file. A secondary key is any key that is not the primary key. The secondary key need not be unique, that is, more than one record may have the same secondary-key value.

The person in charge of a data-management system is the "data base administrator" (DBA). The DBA makes the decisions as to how data are to be structured, such as which access methods are to be used to manipulate the data and the type of storage devices to be used. He is also responsible for monitoring the performance of the system (occasionally with the help of system-generated statistics but too often he must rely on his intuition) and, if possible, making adjustments to reduce any inefficiencies in the system. In the future, it should be possible to automate many of the decisions now made by the DBA but, currently, the DBA has a very important role in "tuning" the system so that it operates as efficiently as possible.

1.3 Basic Data Access

The basic access methods (such as IBM's BSAM, QSAM, BDAM [IBM76], and VSAM-ES [IBM73a]) provide the user with a means of accessing records (both logical records and physical records) as they are physically stored in a file. The records can be processed sequentially or randomly if the

user knows the address of the record. The access methods normally provide such services as grouping several logical records together into one physical record ("blocking") in order to increase the utilization of the secondary storage device. (For example, on an IBM 3330 disk drive, there is a fixed device overhead of 135 bytes per physical record regardless of the size of the physical record. If each physical record contains one 80-byte logical record, then 61 logical records can be stored on a track. However, if each physical record contains 80 80-byte logical records, then 160 logical records can be stored on each track [IBM74b].) Blocking also reduces the the number of I/O requests that must be made since several logical records are transferred to/from main memory with each I/O request. The use of the basic access methods provides fast access to records with a minimal amount of CPU overhead.

One of the major disadvantages of using a basic access method is that the user must be aware of all aspects of how the records are stored. The systems programmer normally has no difficulty in manipulating the actual records in a file; however, the applications programmer and the casual user often find the intricacies of such low-level data access difficult to master. Such users may not make the suitable choices when designing files and then must rewrite portions of their programs when it becomes necessary to change the

file structure. It often takes these users several weeks to create the file and write and debug their programs so that the records are accessed properly.

Another problem which the user of a basic access method must face is that records can not be physically inserted into or deleted from the middle of a file without rewriting the entire file. If records must be inserted or deleted, special routines to perform logical insertions/deletions on the file must be written. If several programs access this file, then the special routines must be included in each program and the user must ensure that any changes in the routines are reflected in all copies of the routines.

The benefits that are gained from fast record access using a basic access method are usually offset by the amount of time required to design and maintain the programs which access the records. The basic access methods are best used for files which have a very simple data structure and from which records are not deleted and to which records are inserted only at the end.

1.4 Primary Key Data Access

The primary-key access methods (such as IBM's ISAM [IBM71] and VSAM-KS [IBM73a]) are more powerful and easier to use than the basic access methods. Primary key access methods permit the user to access records by the primary key, a logical identifier, instead of by their addresses in

the file. Primary key access methods also permit the user to insert records into and delete records from any position in the file.

The primary-key access method determines a record's physical location either by looking up its key in a directory (or index) or by performing a transformation ("hashing") on the key. It then uses the equivalent of a basic access method to retrieve the record. Insertions are normally handled either by inserting the record in an overflow area and adding a pointer to the inserted record or by leaving some unused record locations throughout the file (called "distributed free space"), and then moving some of the existing records to make room for the new record.

The primary-key access methods require extra secondary storage space if an index is used and extra page accesses to search the index. The access method is also larger than a basic access method because it performs more functions for the user. However, these disadvantages are offset by the fact that it takes a user less time to write and debug a program if a primary-key access method is used.

The primary-key access methods provide good data-management facilities as long as the data structure of the file remains relatively simple. However, as data structures become more complex, the primary-key access methods fail to provide the needed facilities. For example, as applications

become integrated, the data required by an application program may be in records that are stored in several data sets instead of in just one data set. Instead of reading one record and processing it, the application programmer must read records from several data sets and build a composite data record before performing any processing. Thus, the programmer becomes responsible not only for processing data correctly, but also for building the records correctly.

1.5 Data Base Management Systems

In order to shift the responsibility for data management to the operating system, complex data-management systems, called "data base management systems" (DBMS's), are being designed. (For the purpose of this thesis, we view data base management systems only as sophisticated access methods; other facilities provided by DBMS's such as the control of on-line terminals are not discussed.)

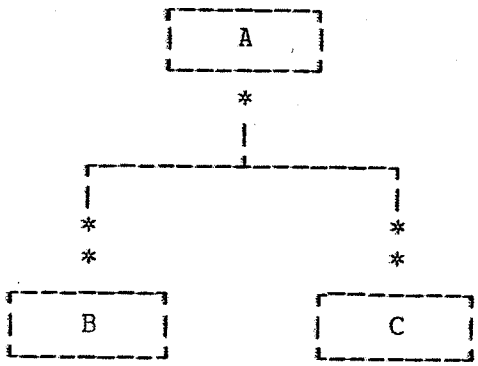
The purpose of a DBMS is to extract data from a pool of data or "data base", and return the data to the programmer. The data requested by a programmer are referred to as a "segment". A segment is a logical entity created by the DBMS from one or more logical records in the data base. Within limits, the definition of a segment can be changed for each program.

Physically, the data in a data base may be stored in

more than one data set but only the DBMS need be concerned with such details; the application programmer is concerned with the logical structure (segments) not the physical structure (records) of data. This separation of the programmer from the method by which data are physically stored is a major advance in data management.

1.5.1 Hierarchical and Network Models

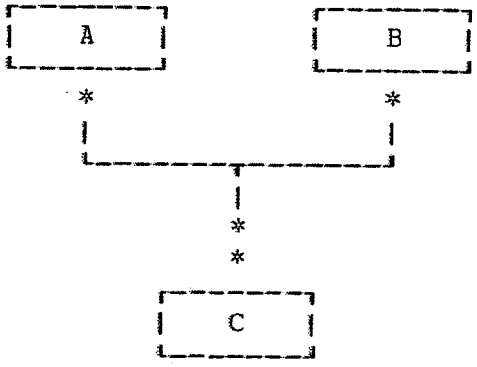
The data models used in most current DBMS's (such as IBM's IMS [IBM75], Cincom's TOTAL [CINC74], MRI Systems' SYSTEM 2000 [MRI74], etc.) are of two basic types: hierarchical models and network models. The hierarchical model, as used in IMS, uses a tree structure to describe the relationships between segments. For example, the hierarchical structure



defines a "parent" segment, A, that has two child segments: B and C. Normally, a parent segment is permitted to have more than one occurrence of each type of child segment:

this is a one-many relationship. (In the diagrams, one asterisk is used to represent an x-one relationship and two asterisks are used to represent an x-many relationship.) Thus, the segment Ai might have as its children: Bi1, Bi2, ..., Bim, and Ci1, Ci2, ..., Cin.

The network data model uses either a simple plex structure or a complex plex structure to describe the relationships between segments. The data structure



is an example of a simple plex structure of the type used in TOTAL [CINC74] in which both A and B are permitted to share a common child, C; this is a many-one relationship.

The structure

is an example of a complex plex structure: the segment B is a child of A, but A is also a child of B; this is a many-many relationship. The complex plex structure is difficult to implement directly, and so many DBMS's do not permit the direct use of complex plex structures.

Both the hierarchical and the network data models describe the logical organization of segments in a data base and the programmer must understand the segment structure in a data base before he can process the segments. For example, in the hierarchical model, before a child segment can be accessed, the corresponding parent segment must first be accessed (even though the parent segment may not be needed).

Processing a data base frequently involves searching for specific parent segments and then examining some or all of the segments' children. This type of processing is referred to by Bachman as "navigating" through a data base [BACH73].

> "This revolution in thinking is changing the programmer from a stationary viewer of objects passing before him in core into a mobile navigator who is able to probe and traverse a data base at will."

While the procedure of navigating through a data base may be easy for the experienced programmer, it is often quite difficult for the less experienced programmer and almost

impossible for the casual user of the data base. (Inexperienced programmers frequently do not retrieve all required segments properly and may unknowingly delete the wrong segments.) The actual users of the data are not able to access the data directly and easily; instead, the application programmer becomes an intermediary between the user and his data.

Most DBMS's that use hierarchical or network models to describe the logical organization of data also use the same structure to store the data. Thus, once the data are stored in the data base, the data model can not be changed unless the data base is recreated by copying the data base and then using the copy to create a new version of the data base. (This process is referred to as "unloading" and "reloading" the data base.) If the data model is changed and the data base is recreated, then programs which access the data base may have to be modified so that they use the new model of the data. Some DBMS's, such as IMS [IBM74c], permit the DBA to define "logical data bases" which contain segment types defined in other data bases but which are reordered to present a different "view" of the data for the user. The use of logical data bases permits greater flexibility in defining the ways in which the user sees the data; however, the definition of a new logical data base is normally not trivial (it may involve unloading and reloading the existing

data base) and can be accompanied by complicated rules as to how segments are to be added, deleted, and modified. The overall lack of flexibility in the logical data structure prevents hierarchical and network data models from evolving as the data and the uses of the data change.

1.5.2 The Relational Data Model

In 1970, Codd [CODD70] proposed a new model of data called the relational data model. Codd believed that the user's view of data should be independent of the manner in which data are physically stored. Codd's model presents an abstract view of data which does not directly define the relationships between segments nor does it imply a specific method of storing the data. The relational data model permits the user to view data as elements in a two-dimensional table called a "relation": each row in the table is called a tuple and describes an entity; each column in the table is called a domain and describes an attribute of an entity. We refer to the value of a column as an attribute value (although it is often referred to as a domain value).

Codd also defined the following properties of relations [CODD70].