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.)

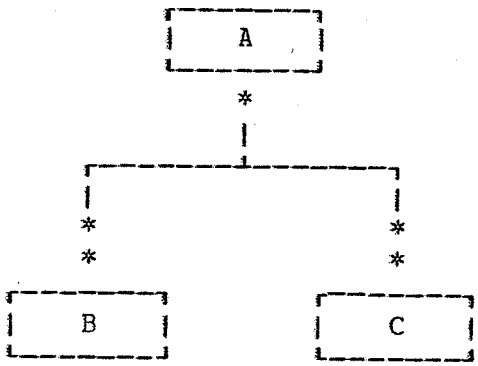
The purpose of a DBMS is to extract data from a pool of data or "data base", and return the data to the programmer. The data requested by a programmer are referred to as a "segment". A segment is a logical entity created by the DBMS from one or more logical records in the data base. Within limits, the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

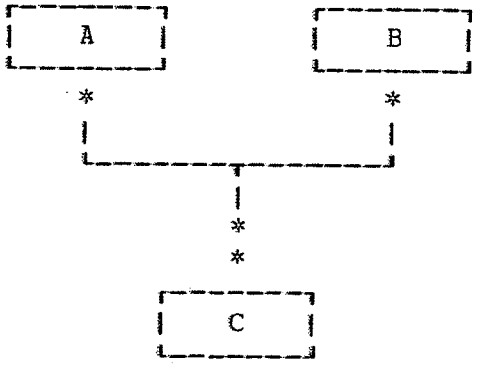
The data models used in most current DBMS's (such as IBM's IMS [IBM75], Cincom's TOTAL [CINC74], MRI Systems' SYSTEM 2000 [MRI74], etc.) are of two basic types: hierarchical models and network models. The hierarchical model, as used in IMS, uses a tree structure to describe the relationships between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

1. No two tuples in a relation are identical.

2. The ordering of tuples in a relation is not significant. (Theoretically, the fact that tuples are not ordered is important, but for most practical applications, the user must be permitted to define an ordering for the tuples processed.)

3. The ordering of columns in a relation is not significant.

4. Each attribute value is single-valued.

The notation used to describe a relation is

RELATION(DOMAIN1, DOMAIN2, ..., DOMAINn) .

For example, consider the data describing students enrolled at a university. The domains in the relation might be: S#, NAME, ADDRESS, AGE. Thus, the relation is defined as:

S(S#, NAME, ADDRESS, AGE) .

Each tuple in this relation describes a student and contains one value for each of the four domains. The relation describes the students at the university.

In the relational data model, a key is a domain or set of domains by which tuples are accessed. A key which uniquely identifies the tuples within a relation is referred to as a "candidate key". One of the candidate keys is chosen as the primary key of the relation. All domains or sets of domains are potentially keys that may be used to access tuples.

In the relational data model, there are no explicit

relationships between relations as there are in the hierarchical and network data models. Instead of defining a relationship (one-one, one-many, etc.) directly between two relations, the relationship is defined implicitly by including the same domain in two or more relations. The importance of the implicit definition of relationships is emphasized by Whitney [WHIT74].

> "A particularly important aspect of the relational data structure is the use of implicit value links between tuples of relations to indicate relationships between tuple items."

For example, the relations

S(S#, NAME, COL#) and COL(COL#, CNAME, DESCR)

are not explicitly related but there is an implicit relationship between them since they have the domain COL# in common. (We assume that within a data base, the use of the same domain name in different relations implies that a common domain is being referred to.) By defining relationships implicitly, the user of the relational data model is not limited to already-defined logical structures. New logical structures can be defined quite easily without the problems inherent in the hierarchical or network data models.

In the relational data model, relations should be defined so that the amount of redundant data is minimized. (Formally, it is sufficient if the relations are in 3rd or

4th normal form. Normal forms are described, for example, by Date [DATE77].) For example, in the relation

SC(S#, NAME, ADDRESS, C#, DESCR)

which describes students and courses, the student's name and address are repeated once for each course in which the student is enrolled. The relation should be split into the equivalent relations

S(S#, NAME, ADDRESS)
C(C#, DESCR)
SC(S#, C#)

in which the student's name and address occur only once. The relation SC is a relation whose purpose is to link together tuples in two other relations. Hierarchical and network DBMS's both contain information that is equivalent to the information in the relation SC (in IMS, it is stored in "logical child segments" [IBM74c]). In the hierarchical and network data models, such information is not normally available to the user; however, in the relational data model, the information in SC is available to the user and can be processed in the same manner as any other relation. Such relations can be used to define the equivalent of hierarchical and network data structures in the relational data model.

The second major advantage of using the relational data model is that it does not impose a specific physical structure on the data. The data can be stored using a

hierarchical storage structure, a network storage structure, or any other convenient storage structure. (This lack of an obvious physical structure for relations makes the choice by the DBA of the structure used to store each relation very critical. In order to aid the DBA, it is important that the DBMS generate statistics which indicate how efficiently each relation is stored.) With the relational data model, the user is not expected to know how a particular type of tuple should be accessed (for example, by accessing the parent tuple first, as in the hierarchical model); the user requests a specific tuple or group of tuples and the relational data base management system (RDBMS) determines a way to access the tuple(s). Since the user no longer needs to know the physical structure of the data in the relation, it is possible to change the physical structure without affecting the user.

Another advantage in using the relation data model is that precise mathematical languages have been developed for expressing queries against a relation. Two of the languages, the relational calculus and the relational algebra, have been shown to be sufficient for expressing any query [CODD72].

1.6 Current Relational Systems

In this section, we survey some of the major RDBMS's that are currently being tested. The systems examined are

intended to be representative of relational systems, but not a complete list. Only the low-level structures of each system, such as the storage mechanism and any data recovery mechanism, are examined; higher-level facilities, such as the data-manipulation languages, are not examined.

1.6.1 INGRES

The Interactive Graphics and Retrieval System (INGRES) [STON76] is being developed at the University of California at Berkeley. INGRES is implemented on a PDP 11/40 machine using the UNIX operating system. INGRES is being used to examine the decomposition of complex queries into queries involving only one variable, the support of integrity constraints by modifying queries, and the manipulation of data bases by casual users.

INGRES stores tuples on 512-byte pages; each relation is stored in a separate data set. Both indexed and hashed access by primary key are provided: the index contains the largest primary key on each primary page; the hashing function used with hashed access is a modulo-division technique. The internal identifier of a tuple (TID) consists of a primary page number and an indirect-address number within the page. Tuples are not ordered by primary key within a page; so primary-key access involves searching the page for the desired tuple. Pages are initially loaded to approximately 80 percent of capacity; when a primary page

becomes full, overflow pages are chained to the primary page. The overflow pages must be searched sequentially so that locating a tuple that is stored on an overflow page may involve several page accesses. INGRES maintains secondary indexes to provide access to tuples by secondary key.

INGRES guarantees the integrity of each relational calculus command (which normally involves more than one INGRES command) by using deferred updating, that is, by saving all modified tuples in a deferred-update file until the entire relational calculus command has been processed; then, the actual modifications are made to the tuples. Should the system fail during the actual updating, INGRES completes the operation by reprocessing the deferred-update file. In order to roll back a relation to the state that it had at an earlier time, INGRES must use a journal file to determine the changes made to the various tuples. (A journal file contains a list of all changes made to a data base.)

1.6.2 ZETA

ZETA is a RDBMS that is being developed at the University of Toronto [BROD75]. It is written in PLI for IBM machines with the O.S. operating system. ZETA is being used to examine the efficiency of relational representations and a variety of user interfaces.

ZETA stores tuples in fixed-length pages; tuples are

always added to the end of a relation. The TID consists of the tuple's sequence number within the relation. ZETA permits the user to create new relations called "marks" which contain the TID's of tuples in another relation which satisfy a qualification. A mark, itself, can also be marked. A mark, however, is not kept up to date and so must be recreated if the original relation changes. Secondary indexes are being added to ZETA.

1.6.3 XRM

The Extended Relational Memory System (XRM) [LORI74] is being developed at the IBM Cambridge Scientific Center. XRM is being used to test relational storage structures and to test languages designed for casual users of a data base.

XRM is built on top of the Relational Memory (RM) System which supports binary relations. Tuples are stored in 4096-byte pages; TID's consist of the page number and an indirect-address number within the page. XRM uses hashing to determine the page on which a tuple is stored. Within a page, tuples are linked together in ascending order. XRM also maintains secondary indexes to permit access by secondary keys.

1.6.4 System R

System R [ASTR76] is being developed at the IBM Research Lab in San Jose. It is implemented on an IBM 370

machine using a special VM/370 operating system which is modified to permit data to be shared by several virtual machines. System R is being used to test automatic concurrency control, recovery, and integrity, and the support of high-level language interaction with data bases. System R has features in common with the DBMS ADABAS [SOFT74].

System R stores relations in segments: a segment is a collection of pages which can, if necessary, be shared by more than one relation. TID's consist of a page number and an indirect-address number within the page. Pages are allocated to segments from a common page area and System R uses a page map to map the logical page-number in a segment to the physical page-number in the file. When there is not enough room in a page for a tuple, the tuple is placed in an overflow page which is linked to the primary page. System R avoids the problem in INGRES of searching several overflow pages by storing a pointer in the primary page to each overflow tuple on an overflow page. Thus an overflow tuple can be retrieved with at most two data page accesses.

System R uses "images" (secondary indexes) to provide keyed access to relations. One image in each relation may be defined as "clustered", causing System R to use primary-key ordering to store tuples. Tuples in different relations can be joined together over a common value using the "link" facility: links create a parent-child hierarchy. System R

uses TID's to link the tuples together. One link in each relation may be defined as clustered, causing System R to store tuples that are linked together as close as possible to each other. The use of clustered links causes the number of pages accessed to be reduced. The use of links provides faster access to tuples than the use of images, but the addition or deletion of an image is much easier than the addition or deletion of a link since the manipulation of images does not cause data tuples to be modified.

System R uses an interesting technique to maintain the integrity of a relation while a group of transactions is being processed. After making changes to a page, the page is written to a new location, not back on top of the old version of the page. This new location is recorded in a new copy of the page-map tables. At the end of the group of transactions, System R has two page-map tables: the page-map table in secondary storage indicates the state of the relation before the changes were made; the page-map table in main memory indicates the state of the relation after all transactions are processed. By saving the new page-map table to secondary storage, the relation is brought up to date; by not saving the page-map table, the relation is rolled back to the state it had before the transactions were processed. The use of this technique requires System R to have complete control of the I/O facilities; an I/O system

in which a file is considered to be an extension of the user's address space could not be used since System R must maintain control of where pages are written. Once the page-map tables are saved, if the relation must be rolled back, System R must use the same technique as INGRES to restore a relation: processing the journal file in order to reverse all changes made.

1.7 Thesis Overview

As data-management systems become more complex, it is important to analyze not only the data-management system as a whole, but, also, the facilities provided by the individual subsystems within the data-management system. In the CODASYL report [CODA71] and the ANSI/X3/SPARC interim report [ANSI75], an attempt is made to define standards for the subsystems of data-management systems. IBM's Data Independent Access Model (DIAM) also defines a generalized data-management model [SENK72], [SENK75], [SENK76]. However, these reports tend to define general systems and do not examine throughly the specific problems encountered when relational data base management systems are implemented.

In this thesis, we define a data-management system in terms of its necessary subsystems. The data-management system is designed specifically to support the relational view of data but it could also be used to support other models of data. The data-management system is not based on

an existing system; instead, the features required by a relational data base management system are examined and a system that provides the necessary functions is developed.

The data-management system consists of four subsystems which are described in the following four chapters. Each subsystem provides a service to the other subsystems but the manner in which this service is performed is independent of the other subsystems. Thus, it is possible to remove one subsystem and replace it with a different subsystem which performs the same task but in a different manner. For example, one subsystem stores and retrieves tuples and only that subsystem is permitted to manipulate stored tuples. If necessary, the format of stored tuples can be changed by modifying only the one subsystem.

The device system is the "lowest" of the four subsystems. This system manipulates the pages used to store the components of a relation. The pages may be moved from one location to another (within or between data sets) and it is the responsibility of the device system to be able to find a particular page. The storage system manipulates the tuples within a page. Pages are retrieved by the device system and the storage system extracts the required tuples from the page. The storage system also attempts to minimize problems encountered when a relation must be rolled back to its state at an earlier time by maintaining copies of the tuples in

the relation. The access-path system is used to determine where tuples with a given set of characteristics are stored. Access paths themselves consist of tuples which are manipulated by the storage system. The access-path system is designed so that access paths can be created, modified, and deleted as easily as possible without having to change other portions of the system. The retrieval system is the interface through which requests are made by users to the data-management system. A simple data-manipulation language (DML) which permits users to access tuples associatively instead of by location is defined. The retrieval system also determines how a particular request can be satisfied efficiently using the currently available access paths.

Chapter 2: Device System

2.1 Introduction

In this chapter, we develop the device system of the data-management system. This subsystem reads and writes the pages used by the storage system. The device system manipulates pages which may be distributed over data sets and devices with differing characteristics.

2.2 Device System

The device system manipulates pages in data sets stored on secondary-storage devices. When requested, the device system reads a page into main memory where it can be manipulated by the storage system. Pages are stored in main memory in a "buffer pool", an area of main memory reserved by the device system for pages. Normally, a buffer pool is large enough to contain many pages at a given time. When the storage system requests a page, the device system first determines whether or not the page is already in the buffer pool. If the page is not in the buffer pool and all of the available locations in the buffer pool are being used, the device system removes one of the pages from the buffer pool to make room for the required page. Normally, the page in the buffer pool that has been least recently used is selected for replacement. If the page has been modified, it

is written back to the corresponding data set before the new page is read. (Thus, a page that is modified by the storage system is not necessarily saved immediately; it is saved when its location in the buffer pool is needed for another page.) The device system uses a common buffer pool for all data sets that are currently active. By sharing the buffer-pool locations among data sets, the main memory used by the buffer pool is utilized more efficiently than if a separate buffer pool is allocated to each data set.

2.3 Page Reference Numbers

The storage system requests pages in a relation using a "page reference number" (PRN). The PRN is a number that uniquely identifies a page within a relation.

There are many page-addressing algorithms that can be used for the PRN. Direct device-addresses such as the full disk address (MBBCCHHR), the relative track address (TTR), and the relative record number (RRN) provide fast access to the required page. However, the use of direct addresses has the disadvantage that if a page must be moved, then all references to that page must be modified. To avoid having to modify PRN's, indirect page addresses may be used. If indirect addresses are used, the PRN contains a pointer into a list of direct addresses instead of containing the direct address itself. The list of direct page addresses used by the device system is called the "device management table"

(DMT). Each entry in the DMT is a fixed length and contains the direct address of a page. Since the DMT entries are of a fixed length, the PRN can be used as a subscript into the DMT. If it is necessary to move a page, only the entry in the DMT must be changed; the PRN itself need not be modified.

2.4 Device Management Tables

When storing a large relation, it may be convenient to spread the pages over several data sets. For example, pages which are frequently referenced could be stored on a fast device while pages which are less frequently referenced could be stored on a slower device.

In order to indicate where a page is stored, each DMT entry must contain not only the direct address of the page within a data set but also the "data set reference number" (DSRN) of the data set on which the page is stored. The data set reference number is a pointer to the description of the data set within the device system. A page can be moved within a data set by changing only its direct address or it can be moved between data sets by changing the DSRN and the direct address. Only the entry in the DMT is modifed; the PRN is never modified. The format of each entry in the DMT is illustrated in Figure 2.1.

Figure 2.1 DMT Entry Format

In a data-management system, a page may temporarily be stored in several different locations. For example, a page may be permanently stored on a slow device, but, when the page is referenced, the device system may temporarily "stage" the page to an intermediate, faster device. The page may then be copied to the main memory buffer pool. If the page is modified and then removed from the buffer pool, it is written back to the intermediate device. The page is copied back to the original device when it is no longer required. In order to support such a migration of pages, the device system permits entries in the (permanent) DMT to point to temporary DMT's. The DSRN in the permanent DMT entry identifies the temporary DMT and the direct address in the permanent DMT entry is a pointer into the temporary DMT. Figure 2.2 illustrates how the entry in a permanent DMT can point to an entry in a temporary DMT which can point to an entry in arcther temporary DMT. Thus, the current location of the page being referenced is (4,5).

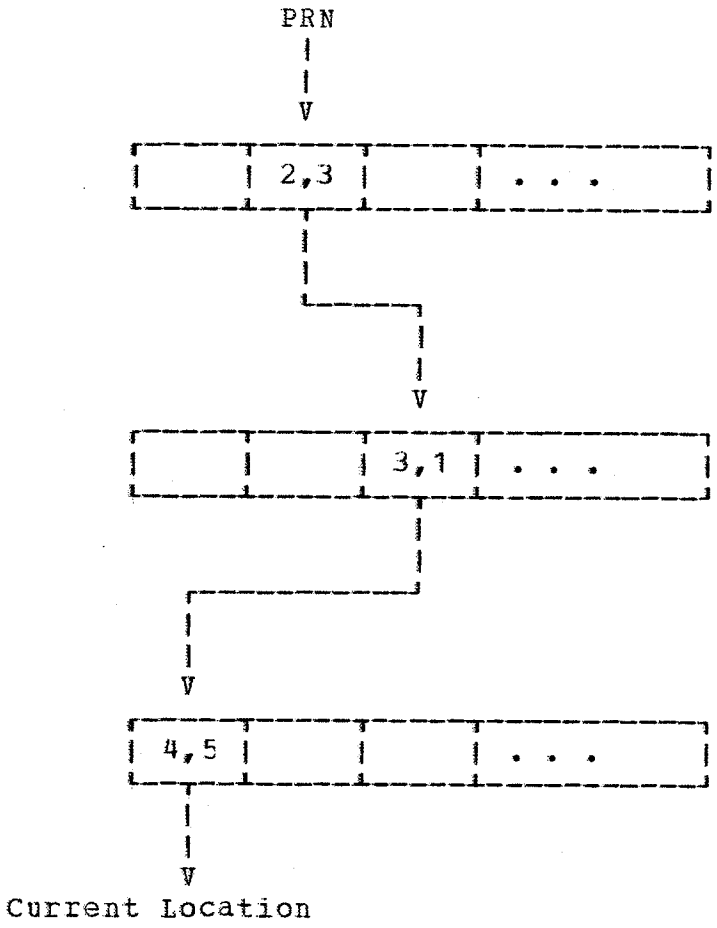


Figure 2.2 Hierarchy of DMT's

The entry in the temporary DMT contains the current location of the page. This current location is either an actual location (direct address) or a pointer into another temporary DMT. In addition to the current location of a page, each temporary DMT entry must also contain the value that was originally in the permanent DMT entry. Also, in order to be able to remove a temporary DMT entry, each

temporary DMT entry must contain a back pointer to the corresponding DMT entry that points to the current entry. If a back link is not included, then all active DMT's must be searched for the reference to the DMT entry to be deleted. Figure 2.3 illustrates the format of each temporary DMT entry. The back link, previous value, and current value all contain a DSRN and direct address.

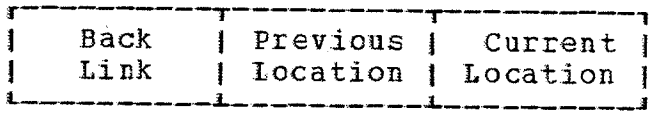


Figure 2.3 Temporary DMT Entry

The use of temporary DMT'S removes the need for special buffer-pool tables; the buffer pool can be viewed as a temporary data set. When the device system moves a page to the buffer pool, it modifies the pointer in the corresponding DMT to point to the buffer-pool DMT. When it is necessary to purge a page from the buffer pool, the page (if modified) is written back to the location specified in the current DMT entry, and the original DMT entry is changed to point to the new location.

2.5 Physical Records

In order to reduce the complexity of the storage system, all pages in a relation are the same size. However, the page size chosen for a relation may not be optimal for

all devices on which the pages may be stored. To overcome this problem, the device system can break up a page into one or more physical records to provide better space utilization on a particular device. These portions of a page are stored contiguously on the device so that they can be processed in one I/O operation. Figure 2.4 illustrates how a page can be viewed by the device system as consisting of two physical records.

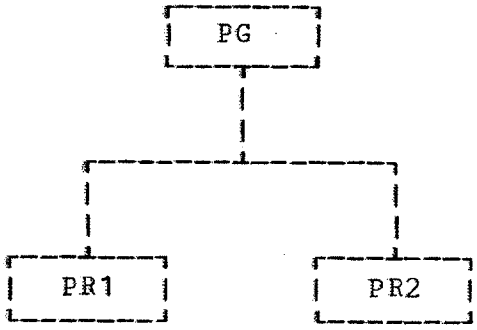


Figure 2.4 Segmenting a Page

If the page size for a relation is small, the device system may store several pages in each physical record in order to increase the number of pages that can be stored on a particular device. Figure 2.5 illustrates how a physical record can be viewed by the device system as containing two pages.

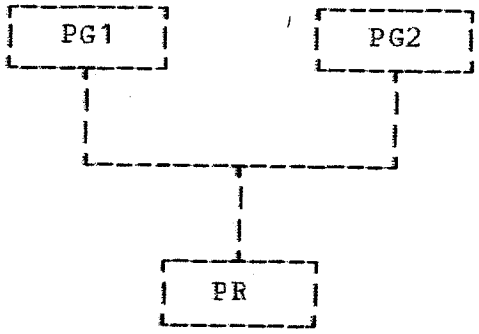


Figure 2.5 Segmenting a Physical Record

This technique for the efficient utilization of device space is an extension of the technique used in IBM's VSAM [IBM77].

Chapter 3: Storage System

3.1 Introduction

In this chapter, we develop the storage system of the data-management system. This system manipulates the tuples in a page. First, we examine the facilities that the storage system should provide. We then describe a storage structure which can be used to provide these facilities.

3.2 Storage-Structure Properties

In this section, we describe some of the properties we would like to see in a storage structure. These properties describe ideal storage structures which it may not be possible to implement completely, but they give us a standard which can be used to evaluate other structures.

The amount of secondary storage used to store a relation should be minimized in order to decrease the number of pages accessed while processing requests. However, it is often necessary to increase the amount of secondary storage used in order to decrease the response time for on-line applications. For example, adding indexes (as described in Chapter 4) requires extra storage but reduces the total number of page accesses. Nevertheless, when data bases are large, the cost of storing the data is a major consideration. One of the goals of the storage system is to reduce

the amount of unused space on each page thereby reducing the amount of secondary storage space and increasing the amount of data transferred in each page.

There should be no (or little) redundant data in order to avoid consistency problems when modifying tuples. For example, if the data which describe a student are duplicated in several tuples, then the RDBMS must reflect the changes made to one tuple in the other duplicate tuples. This often requires extra pointers to link the duplicate tuples together (as illustrated by IMS in its support of logical data bases by "physical pairing" [IBM74c]) which adds another level of complexity to the storage system. Reducing the amount of redundant data also decreases the amount of secondary storage required to store the relation.

3.3 Tuple Identifier Properties

In order to permit the storage system to access the tuples in each relation, each tuple is assigned an internal identifier or address called a "tuple identifier" (TID). We now examine some of the properties that we would like TID's to have. These properties are ideals and it may not be possible to satify all properties at once.

In order to be able to process tuples efficiently, the TID should indicate where (if only approximately) the tuple is stored physically. If an index is required in the mapping between a TID and its location, then not only must

the index be searched each time that a tuple is retrieved but the index must also be created, retrieved, and maintained.

The TID should not bind the tuple to a fixed location if we are to permit relations to change over a period of time. Some DBMS's, such as IMS and TOTAL, use identifiers which specify the exact physical location of data in the data base. When physical identifiers are used, it is not possible to reorganize portions of the data base in order to meet performance standards.

As long as a tuple remains in a relation, its TID should not change. As we shall show in Chapter 4, the TID of a tuple may be stored in many access paths. If we permit a tuple's TID to change, then all occurrences of that TID must also be changed. In order to avoid the problems created by changing a TID, the TID of each tuple should not be changed as long as the tuple remains in its relation.

Another desired property of the TID is that for all tuples i and j in a relation, if PKEYi < PKEYj then TIDi < TIDj (where PKEY is the primary key of a tuple). This property of TIDs can be used to reduce the number of tuples that must be examined when processing complex queries involving the primary key. If the TID's are ordered in the same manner as the primary keys, then a query that involves a range of primary keys can be reduced to the simpler, but

equivalent, query involving TIDs.

3.4 Tuple Ordering

There are three basic methods that can be used to order tuples in a relation: sorted ordering, hashed ordering, and chronological ordering. Sorted ordering involves storing tuples in primary-key order. Access to tuples by primary key normally involves the use of a directory (which is discussed in Chapter 4). Sorted ordering permits the sequential processing of a relation in ascending order of primary key. Hashed ordering involves performing a transformation on the primary key using a hashing function in order to determine the position of a tuple. The major advantage of hashing is that access by primary key does not involve the use of a directory; however, while the relation can be processed sequentially, the tuples are not returned in ascending order of primary key. Also, if the hashing function does not distribute the tuples uniformly over the space available for the relation, then storage space may be allocated but not used. Chronological ordering involves storing each new tuple at the end of the relation. This method distributes tuples uniformly over the available space with no unused space, but, again, tuples can not be processed sequentially in primary-key order.

We shall assume that the tuples in most relations are stored in sorted order. By choosing the primary key wisely,

it is possible to reduce or eliminate the sorting of tuples before they are returned to the user. Another reason for storing tuples in sorted order is to permit a query optimizer to reduce the scope of queries. When evaluating complex queries which can not be resolved without scanning a relation, if the query involves the primary key, then a query optimizer can reduce the scope of the scan to the subset of the relation involving the required primary keys.

As is shown in Chapter 4, the use of sorted ordering also causes a reduction in the size of an associated primary-key directory. And, because the directory is smaller, fewer page accesses are required while processing the directory.

A final point in favour of sorted ordering is that the number of page accesses is reduced if a query involves a primary-key "locality of reference". If requests involve tuples with similar primary keys then the number of page accesses required to process the request is reduced since the tuples reside on the same or nearby pages.

In order to provide a general storage system, the user should, however, be permitted to use hashed or chronological ordering. Existing DBMS's use various combinations of ordering. For example, ADABAS and ZETA use chronological ordering while TOTAL and XRM use hashed ordering. Some systems such as IMS and INGRES provide both sorted and

hashed ordering.

3.5 Tuple Format

In this and the following sections, we indicate how the storage system manipulates tuples in order to provide the properties described earlier.

The format of tuples in the storage system is very simple: each tuple contains a prefix and a data portion. The tuple prefix contains the TID of the tuple and various status indicators. The data portion of each tuple contains the attribute values of the tuple. (In Appendix I, we indicate how the various portions of a tuple can be stored efficiently.)

The status indicators in the tuple prefix are used to indicate the various states of the tuple. Two of the states that a tuple can have are: active, deleted. As is indicated later in this chapter, when a tuple is deleted it may not always be possible to remove it immediately from its relation. Until a tuple can be physically deleted, the status indicator is used to mark the tuple as being logically deleted. The other states can be used to indicate that a tuple was inserted, updated, etc., and are discussed later in this chapter.

The attribute values in each tuple are always stored in the same order; however, the storage system returns the attribute values in the order specified by the user. Thus,

the user need not be aware of the physical ordering of the attribute values in a stored tuple.

Since the TID is not to bind a tuple to a physical location, the PRN can not be used as the TID. The primary key is not chosen to be the TID because an index is required to determine the physical location of the tuple with a given primary key within a relation. If tuples are assigned sequence numbers as they are inserted, then the TID does not bind the tuple to a physical location. However, these TID's do not reflect the primary-key order. If the tuples are sorted before being inserted into a relation, then the sequence numbers do reflect the primary-key order. However, when a tuple is inserted after the relation is initially created, some of the tuples must be renumbered. This violates the property that TID's should not be changed.

If the TID consists of two parts: an original sequence number (or tuple number) and an insert number, the problems described above are eliminated. The tuple number is assigned to tuples inserted when the relation is created. These tuples must be ordered by primary key before they are inserted. For these tuples, the insert number is zero and the tuple number is assigned sequentially beginning at one. When a tuple is inserted at a later time and its primary key is greater than the key of the tuple with TID (n.0) but its key is less than the key of the tuple with TID (n+1.0), then

the new tuple is assigned a TID of (n.m) where m > 0. This scheme is similar to that used in MANTES [FERC78a] and to the Dewey decimal notation [KNUT75]. If we assume that there is no limit on the number of tuples that can be inserted, then the TID satisfies our basic properties. (A method of storing such TID's is presented in Appendix I.)

3.6 Mapping to Physical Page

We now examine the process of determining a tuple's physical location using the TID. We shall consider it sufficient if the number of the page on which a tuple resides can be determined; once a page is moved to main memory, it can be searched very quickly for the desired tuple.

One method used to determine the physical page on which a tuple resides is to maintain a dense index of TID's and their associated pages. (A dense index of TID's is an index in which there is an index entry for each unique TID. A non-dense index of TID's contains index entries for only some of the TID's.) This method has the advantage that all or part of a relation can be reorganized; only the index must be changed to reflect the new locations of the tuples. This scheme is used quite successfully in ADABAS [SOFT74]. A second advantage of using a dense TID index is that as tuple-usage patterns emerge, tuples that are frequently accessed together can be stored on the same page. Such a

scheme has been examined by Hoffer [HOFF75].

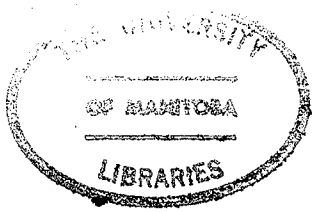
The major disadvantage of using a dense TID index is that it requires a large number of page accesses to process the index before tuples can be processed. For example, if approximately 1000 page numbers can be stored in each index page and there are N tuples in a relation, then N/1000 pages are required to store the index. In a relation with several million tuples, the size of such an index is prohibitively large. If a relation is processed randomly, the number of index-page accesses may approach the number of data-page accesses.

The method used in the storage system to map TID's to pages is to include a "logical page number" in the TID. A logical page number is a number that is assigned sequentially, beginning at one, by the storage system to each "logical page" as the relation is created. The TID becomes

(logical-page number, tuple number.insert number)

where the tuple number is the tuple sequence number within a page instead of within the entire relation. This type of TID is an extension of the TID used in INGRES [STON76] and in System R [ASTR76]. Since we assume that tuples are initially loaded in ascending order of primary key, the TID reflects the primary-key order.

For reasons which are explained later in this chapter, the PRN is not used as the logical page number. However,



since the PRN is required in order to be able to access a page, the storage system maintains a "storage management table" (SMT) which contains the PRN of each logical page. The SMT contains an entry for each logical page in a relation. In the following sections, additional information is added to each entry in the SMT.

This TID reflects the primary-key ordering but it violates the property that no extra page accesses be required when retrieving a tuple since the SMT must be accessed. However, the SMT's are small compared with the number of tuples that can be referenced since each entry in the SMT defines the location of all tuples in a page.

3.7 Logical Pages

A major problem with including a page number in the TID is that at some time, there will probably be too many tuples to fit on the page to which they have been assigned. At that time, it would be convenient to be able to increase the size of the page, but it is normally not possible to extend a physical page once the data set is created. To get around this problem, if a (logical) page is too large to be stored on one physical page, the storage system splits the logical page into "logical page segments" which are stored on physical pages. The tuples within a logical page are ordered by TID so that the tuples within and among logical-page segments are properly ordered. In order to avoid extra

I/O requests (as are required in INGRES and System R) when accessing the segments of a logical page, the entry for a logical page in the storage-management tables is modified to point to a list of special logical-page-segment entries. Each special entry contains a PRN and the maximum TID that is in the segment on that physical page. Thus, by modifying only the storage-management tables, the storage system can extend a logical page to any size. As long as the TID of the desired tuple is known, only one data I/O request is required to find the segment on which the tuple is stored.

A few entries at the end of each SMT page are left empty for use when a logical page is split into two or more segments. By storing the split-page entries on the same SMT page as the original SMT entry, extra SMT-page accesses are not required when processing a split logical page.

The storage system allocates space for a logical page only when the space is required. If a logical page is small, then the logical page is stored on the same physical page as other small logical pages. Thus, the storage system attempts to minimize the number of physical pages on which a relation is stored and to maximize the amount of data transferred during each I/O request.

The format of each physical page is shcwn in Figure 3.1.

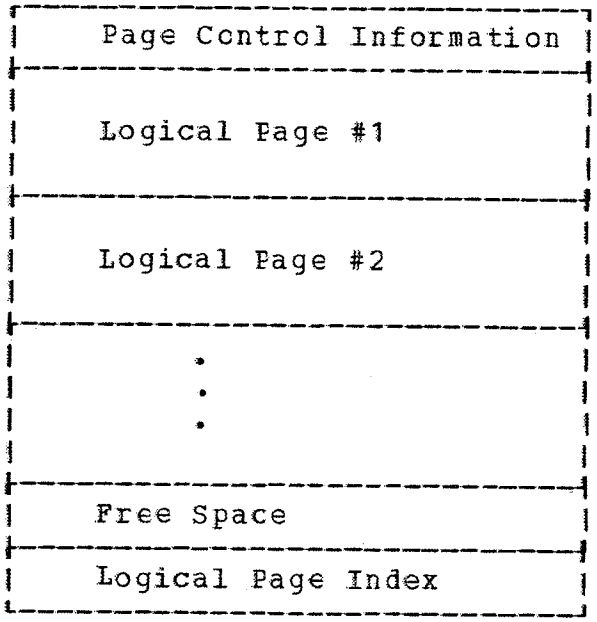


Figure 3.1 Physical Page Format

At the beginning of a physical page is a small amount of page control information. The page control information contains system information such as the number of logical pages in the physical page. Following the page control information are the logical pages. The standard method of allocating free space is to store the free space either at the end of each logical page or at the end of the physical page. However, some logical pages, after the initial changes, may not be modified so the free space allocated to them is wasted. If the free space is stored at the end of the physical page instead of at the end of each logical

page, it can be shared by all logical pages in the physical page. However, when tuples are moved within a page, all following logical pages must also be moved. In order to reduce the amount of data movement within a page, the storage system keeps a small amount of free space within each logical page and the remainder of the free space at the end of each physical page. Thus, a small number of changes can be made to a logical page without having to move the other logical pages. When the free space within a logical page is exhausted, some extra free space is made available from the physical-page free-space area. If there is no free space at the end of the physical page, the storage system attempts to find some by taking free space from the other logical pages in the physical page.

The logical page index at the end of a physical page identifies each of the logical pages in the physical page and contains the displacement of the logical page within the physical page. In order to determine where a tuple is stored on a physical page, the logical page index is searched to determine the position of the logical page and then the logical page itself is searched for the tuple.

The format of a logical page is shown in Figure 3.2.

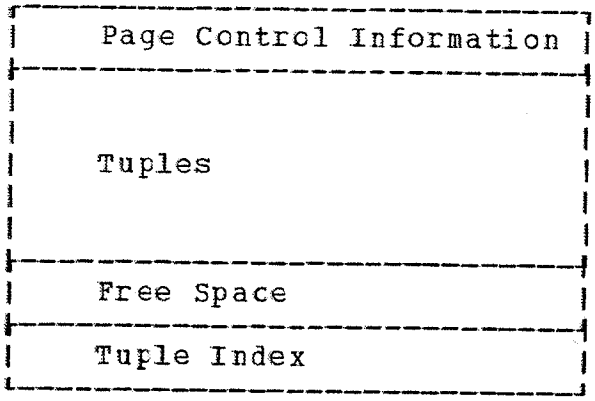


Figure 3.2 Logical-Page Format

At the beginning of each logical page is a small amount of page control information which contains information such as the amount of free space in the logical page. The tuples in the logical page are stored in ascending order of TID following the page control information. Following the tuples is the free space for the logical page and following the free space is a "tuple index". The tuple index is used to reduce the time required to find a tuple in a page. The tuple index contains pairs of TID's and pointers: the pointer contains the displacement within the logical page of a group of 5-10 tuples; the TID is the largest TID in the group of tuples. Thus, the tuple index is a non-dense index into the tuples in a logical page. The tuple index takes up little room compared with the number of bytes of data in a logical page.

Tuples are inserted intc a logical page in ascending order by TID. Thus, inserting a tuple causes the tuples with higher TID's to be moved towards the end of the page and deleting a tuple causes the tuples with higher TID's to be moved towards the beginning of the page. Insertions and deletions also cause the tuple index to be modified. The free-space area at the end of the logical page contracts or expands as tuples are inserted or deleted. However, all pointers within a logical page are displacements from the beginning of the logical page, not from the beginning of the physical page. Thus, a logical page can be moved without having to change any of the pointers within the page; only the pointer in the logical-page index is changed.

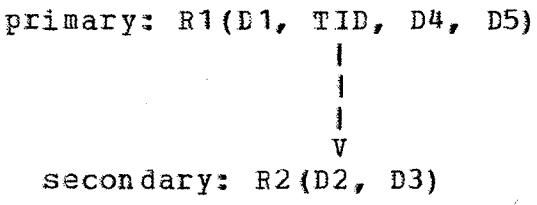
If a relation is stored using hashed ordering, the use of logical pages provides a natural method for resolving collisions. The hashing function generates the logical page number for each tuple and the storage system orders the tuples by primary key within the logical page. (By ordering the tuples by primary key within a logical page, even though the tuples are not ordered within the relation, the time required to find a tuple in a page is reduced.) If few tuples are stored in a logical page, storage space is not wasted by allocating an entire physical page to the logical page; instead, the storage system stores several small logical pages in the same physical page. The storage system

does not allocate space for a logical page until tuples are stored in the page so the hashing function does not have to distribute tuples over all possible logical pages.

3.8 Pointers

The storage system permits the user to define not only normal domains in a tuple but also "pointer domains". A pointer domain is a domain which contains a pointer (TID) to another tuple. A pointer domain is used to provide fast low-level access to associated tuples. The user of the system is not aware of the existance of pointer domains since they are a performance-oriented feature.

If a domain is defined as a pointer, then the storage system, if requested, retrieves both the primary tuple and any subordinate tuples and then concatenates the tuples. For example, the tuple structure



causes the tuple R1(D1, D2, D3, D4, D5) to be returned to the user. The process of retrieving subordinate tuples is recursive: one subordinate tuple can point to another subordinate tuple. If such a composite tuple is to be modified, the storage system saves the TID's of all subor-

dinate tuples in order to be able to make the necessary changes to the component tuples efficiently.

An interesting consequence of using a pointer in the data portion of a tuple is that a relation can be split into two or more parts without the user's being aware of the split. This technique is proposed by Severance [SEVE76b]. For example, in order to increase the number of tuples that can be stored on a page, each tuple in a relation could be split into two parts. The first part of each tuple contains the domains of the tuple that are most frequently used plus the TID of the second part of the tuple. When a tuple is accessed, if only the first part of the tuple is required, then the primary tuple is returned to the user. If both parts are required, the corresponding secondary tuple is also retrieved automatically by the storage system. For example, in the student relation, the student's previous academic history is not needed during most processing of the relation. Thus, it could reasonably be stored in a second relation and retrieved only when necessary. The storage system makes this division of the relation invisible to the user; each user specifies the domains required for his processing and the storage system retrieves the secondary information when necessary.

Pointer domains can also be used to replace key domains in order to provide faster access to associated tuples.

This technique was first proposed by Tsichritzis [TSIC74] and [TSIC75]. However, the replacement of a key by a pointer must not be visible to the user. Thus, if the user accesses the key domain, the storage system must also retrieve the associated tuple in order to make the key value available. If sufficient space is available, both the key value and the equivalent pointer value could be stored in each tuple.

3.9 BASE and MOD Files

In order to guarantee the integrity of a relation as changes are made to it, the tuples in a relation are stored in two files. When a relation is initially loaded (or when it is reorganized) all tuples are placed in a file called the "BASE" file. This file (except during a reorganization of the relation) is never modified. When a tuple is modified, the modified version of the tuple is stored in a second file, called a "MOD" file. When a tuple is inserted after the relation is created, it is also placed in the MOD file. When a tuple is deleted, a copy of the tuple is placed in the MOD file and its status indicator is set to indicate that the tuple is logically deleted. It is the responsibility of the storage system to be able to find a tuple regardless of which file it is stored in. The user of the relation need not be aware that the tuples are stored in two files: the storage system makes it appear as though

only one file is used to store the tuples.

The BASE and MOD file concept is very powerful and has been examined recently by Severance [SEVE76a]. The two major areas where MOD files simplify the work of the RDBMS are in the maintenance of system integrity and in the management of free space.

One of the major problems facing the designers of DBMS's is keeping the data in a data base secure from both system failures and programming errors. In order to minimize the time required to recover from a failure, most DBMS's provide both a backup facility to create a copy of the contents of a data base at a given time and a logging facility to write a copy of all changes made to the data base on a journal file. To recreate a data base, the backup file and the journal file are merged to produce an up-to-date copy of the data base. Creating a backup copy of a large data base can be very expensive due to the number of I/O requests. Also, users must normally be locked out of the data base while it is being backed up; thus, the data base may not be available for a substantial period of time. However, using BASE and MOD files, the backup problem becomes much more manageable. Since the BASE file is never modified, it is necessary to create only one backup copy of the BASE file (and this copy is made when the BASE file is created). The MOD file is similar to the journal file since

it contains changes made to the data base after a given point in time. If the MOD file is small compared with the BASE file, then recreating it after a failure is not as great a problem as recreating the entire file.

In addition to being able to recreate a data base after a system failure, it is also necessary to be able to recover data lost due to programming errors. For example, an application programmer might accidentally delete a portion of a data base. In a DBMS such as IMS [IBM74c], the data are physically deleted from the file. However, when BASE and MOD files are used, the tuples are not physically deleted; instead, the status indicator is used to mark the tuples as logically deleted. To restore tuples accidentally deleted, only the status indicators need to be reset.

The use of MOD files also makes the data base more secure during the testing phase of a new operating system or a new version of the RDBMS. Traditionally, when testing a program, it is necessary to create a copy of the data base and run the test programs against the copy. Then, the test version and the production version of the data base are compared to ensure that they are the same. However, if BASE and MOD files are used, many programs can share the BASE file but maintain different versions of the MOD file. This process reduces the amount of duplication required during testing. This technique can also be used to permit students

to share a data base in an educational environment. Each student has his own MOD file but shares the BASE file with the other students. If a student wishes to start again with the original data base, he has only to create a new MOD file.

A major advantage of BASE and MOD files is that since the BASE file is never modified, it is not necessary to leave any free space in it. The only free space is left in the MOD file. We assume that the ratio of the number of tuples in the MOD file to the number of tuples in the BASE file is small so the management of free space becomes much easier.

One of the interesting consequences of using a MOD file is the ease with which historical data can be stored. In many DBMS's (such as IMS), a percentage of distributed free space is left in each page when the data base is created. However, with historical data arranged in chronological order, free space is necessary only in that part of the file that contains data relating to the current year. Thus, if a large percentage of free space is left in each page, much of the storage space allocated to the file is never used; but, if little free space is left, performance suffers when many insertions are made in the pages containing the current year's data. However, when BASE and MOD files are used, the previous years' data are stored in the BASE file (with no

free space) and the current year's data are stored in the MOD file where free space is automatically maintained. At the end of each year after all changes are made to the current year's data, the relation is reorganized: the current year's data are moved into the BASE file. The next year's data are then stored in the empty MOD file. (The reorganization of a relation is discussed later in this chapter.)

A somewhat unusual use of MOD files is to support updating of sequential files. If a data base is always processed sequentially, it is much cheaper to store it on a tape instead of a direct-access storage device. When changes are made to the data base, they are saved in the MOD file. This technique eliminates having to rewrite the tape file for only a few changes. When sufficient changes accumulate in the MOD file, the MOD file can be merged with the tape file to create a new tape file.

If a relation is extremely volatile, it could be stored entirely in a MOD file. Thus, the use of a MOD file without a corresponding BASE file provides conventional access to the file, but the advantages of using both a BASE file and a MOD file are no longer present.

There do exist two disadvantages when BASE and MOD files are used. The first is that when a tuple in the BASE file is modified or deleted, a copy of that tuple is added

to the MOD file; thus, extra storage is required. Normally, this extra storage is offset by the reduction in free space required to store the file using BASE and MOD files.

A more serious problem is that using a MOD file causes the number of page accesses to increase. When a tuple is requested, the storage system does not know whether the tuple is in the MOD file or in the BASE file. If it is assumed that the tuple is in the BASE file, the tuple retrieved may not be up to date since a modified version of it could be in the MOD file. If the MOD file is accessed first and the tuple is not there, then an additional page access is required to retrieve the tuple from the BASE file.

There are several techniques which can be used to avoid "double-file accesses", accessing first the MOD file and then the BASE file to retrieve a record that is in the BASE file. One technique involves using a dense index which contains the location of every tuple in a relation. The pointer for each tuple would point into the BASE file or into the MOD file. However, the use of a dense TID index was examined earlier in this chapter and found to require too much secondary storage. Severence [SEVE76a], based on work by Bloom [BLOO70], describes a "filter" that can be used to indicate (approximately) in which file a tuple is stored. The filter is a bit map associated with a relation. Initially, all the bits in the filter are set to zero. When

a tuple is added to the MOD file, its TID is hashed (using one or more hashing functions) and the bits in the filter indicated by the result of the hashing function are set to 1's. Then, when searching for a tuple, its TID is hashed and the corresponding bits in the filter are examined. If all of the bits are 1's, then the tuple is probably in the MOD file. If any of the bits are not 1's, then the tuple must be in the BASE file. The size of the filter, the type and number of hashing functions, and the number of changes made to the file determine the number of 1-bits in the filter. As the number of 1-bits in the filter increases, the probability of a double-page access increases. If the filter is large enough and the hashing functions are uniform over the filter, then it is possible to eliminate most double-file accesses. Severence shows that with a 3125-byte filter and three hashing functions, it is possible to reduce the probability of double-file accesses in a file with 10 million tuples to at worst 0.1 with an average of 0.0333.

There are several disadvantages to using the filter in the method proposed by Severence. The first disadvantage is that for a large data base, the filter may be too large to fit on one page. This adds an extra level of complexity to the storage system which must keep the filter in main memory while a relation is being processed. Another problem with the filter is that modifications to a relation are assumed

to be uniform over the entire relation. However, in many data bases, changes involve a "locality of reference", that is, there may be many changes to a small, contiguous area of the data base. These changes still affect the filter for the entire relation, and, as more changes are made, the number of 1-bits in the filter increases, causing the number of double-file accesses to increase. A final problem is that a relation can not be partially reorganized without recreating the entire filter (which would mean reading all MOD pages). With a large relation, it may be too time-consuming to reorganize the entire relation at one time; instead, the reorganization should be performed page by page. But, having to recreate the filter after each page reorganization makes it too expensive to perform a partial reorganization.

To avoid these problems, the storage system maintains an individual filter for each logical page. Since there are not many tuples per page, the filter can be quite small. By maintaining a filter for each page, the total space required for the filter is increased but the filter becomes much easier to manipulate and changes made to one page do not affect double-file accesses on any other page. Also, it is possible to perform a partial reorganization of a relation since only the filters for the reorganized pages must be recreated. In order to eliminate an extra I/O request to

process the filter, the filter is stored in the storage-management tables and is extracted with the other SMT information for the requested page. Each SMT entry for a logical page contains the TID filter and two PRN's (one PRN points to the associated BASE page; the other PRN points to the associated MOD page). Figure 3.3 illustrates the use of the SMT when accessing a logical page.

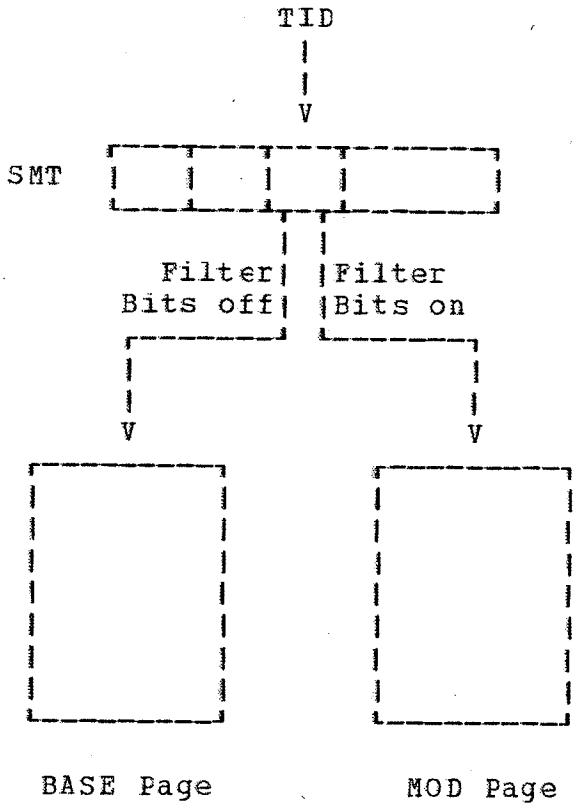


Figure 3.3 Access to a Logical Page

3.9.1 BASE-Page Format

When a relation is created, the tuples are inserted into the BASE pages in ascending order of primary key. The

size of each BASE logical page is, for convenience, normally chosen to be size of a physical page. Tuples are stored in each BASE page until the page can not hold any more tuples. At that time, the logical page number is increased by one and the tuples are stored in the next BASE page. The only free space left in each page is an amount too small to permit a tuple to be stored in it.

3.9.2 MOD-Page Format

The MOD file contains changes made to a relation after the BASE file was loaded. The storage system stores as many MOD pages on each physical page as possible. MOD pages that are not used are not allocated space; instead, the entry in the storage-management table indicates that the MOD page does not exist. The storage system automatically extends a MOD page that becomes too large to fit on a physical page and notes the locations of the page segments in the storage-management table.

As each tuple is placed in the MOD file, its status indicator is used to indicate the purpose of the tuple. A tuple can be marked as deleted, modified, and inserted (for tuples that are added to the relation after the relation is created). These three indicator values can be set in any combination; for example, it is possible to mark a tuple as inserted, then modified, then deleted. These indicator values are used when the relation is reorganized and during

the backup/recovery process.

3.10 Data Base Integrity

We now examine additional functions provided by the storage system to ensure that the integrity of data in a relation is maintained. The major areas of concern are: recovering a relation after the loss of data, restoring a relation to a previous state (rollback), and maintaining the relation in a consistent state.

To the user, being able to recover lost data is extremely important. In the event that data are lost (due to system error/failure, vandalism, etc.), the RDBMS must be able to recreate an up-to-date copy of the relation.

The restoration of a relation to its state at a previous point in time is an easier task than having to recover lost data. The reason for this is that we assume that the data in the relation are currently in the correct format; the only thing required is to remove some of the changes that have been made to the relation. There are two major reasons for restoring a relation: a user has damaged part of the relation by performing changes that were not correct; a higher-level system has decided that some changes made to the relation must be rolled back (possibly only temporarily).

Maintaining the integrity of a relation is also extremely important. Should the computer hardware, operat-

ing system, or data base management system fail, the RDBMS must be able to restore a relation to a consistent state, that is, the state the relation had either before a set of transactions was processed or after having completed the entire set of transactions; the relation must not be left with only some of the changes made.

3.10.1 Data Base Recovery

Creating a backup copy of a large relation is normally a very expensive task. First, the backup operation itself is time-consuming and also uses many computer resources (the channels to the device containing the current copy and to the device containing the backup copy are monopolized by the backup process). Secondly, during the backup operation it is usually necessary to restrict or forbid access to the relation in order to create a consistent copy of the relation. This means that many users of a relation are locked out of the relation for a period that may be as long as several hours. Thus, the backup process is often performed infrequently in order to minimize the amount of time that the relation is not available to users.

By using MOD files, the storage system reduces the problems associated with creating a backup copy of a relation. Since the BASE file is never modified, it is necessary to create a backup copy of it only when it is created. It is expected that the MOD file is quite small

compared with the BASE file and so creating a backup copy of a MOD file is much faster than creating a backup copy of an entire relation. Severence [SEVE76a] refers to backing up a file with 10 million tuples. To backup the entire file takes approximately 6 hours while backing up the MOD file after a week's changes at the rate of 100 changes per hour takes approximately two minutes. Thus, backup copies can be created more frequently and the backup process does not seriously restrict the use of the relation.

The storage system also maintains a journal file which contains a copy of a tuple before it is modified ("before image") and a copy of the tuple after it is modified ("after image"). Each entry on the journal file is also "date stamped" in order to record the day and time on which the change is made.

Creating a new copy of a relation is reasonably straightforward. If the BASE file is still intact, then it need not be restored. (With some devices, it is possible to set them so that only read access is permitted; thus, the file is protected against everything but a hardware error.) In a very important data base, the backup copy of the BASE file could be kept on a mountable direct-access volume so that in order to restore the file, the only action required is to mount the backup volume in place of the damaged volume. Next, the MOD file is restored. (If the backup

copy of the MOD file is also kept on a mountable direct-access volume, then restoring the backup copy requires only that the volume be mounted.) It is then necessary to scan the journal file, searching for all after-images which apply to the relation being restored. (It is normally too expensive to maintain a separate journal file for each relation; instead, one journal file is maintained either for all data bases or for each data base.) Since the entries on the journal file are date stamped, it is possible to create a new copy of the relation as it existed at any point in time after the backup copy was created.

3.10.2 Data Base Restoration

The storage system uses two methods to restore a relation to a previous state. The first involves processing the journal file backwards (beginning at the most-recent entries) and applying the before-images to the relation in order to cancel the effect of changes made to the relation. This process, while not overly time-consuming, involves manipulating both the MOD file and the journal file.

The second method used to restore a relation involves keeping before-images in the relation as well as in the journal file. For each relation, a "direct recovery period" (DRP) is defined. During this period, all versions of each tuple are kept in the MOD file. Then, if necessary, a relation can be rolled back to a previous state that is

within the current DRP without having to access the journal file. Modifications made to a relation during a DRP are broken up into recovery units (RU's) and each recovery unit is assigned a recovery unit sequence number (RUSN). Special commands are provided to the user to permit the definition of a new recovery unit. At any point during a DRP, the user can specify that a relation is to be rolled back to the state it had at the beginning of a previous recovery unit within the current DRP. This rollback causes changes made during that recovery unit and in all subsequent recovery units to be removed.

In order to identify the changes made in each recovery unit, the recovery-unit sequence number (RUSN) is added to the prefix of each tuple in a MOD page. The format of each tuple is illustrated in Figure 3.4.

Figure 3.4 Tuple Format

The most recent version of a tuple is stored first, followed by the earlier versions. By physically removing tuples whose RUSN is greater than or equal to a specified RUSN, the state of a relation is rolled back to the state it had at the beginning of that recovery unit. In order to identify

the MOD pages which contain changes, the storage-management table also contains a recovery-unit sequence number for each MOD page. If the same page is modified during several recovery units, then the highest recovery-unit sequence number is recorded in the storage-management tables. Thus, by scanning only the storage-management tables and not the actual data pages, the storage system can determine which MOD pages contain tuples that must be removed during the rollback process. After removing tuples from a page, the RUSN in the storage-management tables for that page is set to the largest RUSN remaining on the page. Since the contents of a BASE page never change, it is not necessary to include an RUSN in each BASE-page tuple. Instead, the date stamp on which the page was created is stored in each BASE page.

When a DRP is defined for a relation, a date stamp is added to the relation. Each MOD page that is modified is also given a date stamp. The current DRP can be ended and a new DRP defined in three possible ways. The first method of starting a new DRP is initiated automatically by the storage system. Each relation has defined for it the length (in days) of the DRP. When the current DRP expires, a new DRP is automatically initiated by the storage system by changing the date stamp in the relation. The second method of starting a new DRP is also initiated automatically by the

storage system. For each relation, the number of extra tuples that are permitted to be stored in the MOD pages for the current DRP is maintained. If this number becomes greater than the maximum number of extra tuples permitted for this relation, then the storage system initiates a new DRP. Finally, there is a special storage-system command which can be employed by a user to cause a new DRP to be initiated. Normally, this command is used only by a higher-level system based on statistics that it keeps.

When a new DRP is initiated, all extra tuples saved during the previous DRP are not immediately removed from the MOD pages. Instead, the RUSN's in the storage-management tables are set back to zero to indicate that the pages have not been changed since the beginning of the new DRP. Later, when a MOD page is accessed, the date stamp on the page is compared with the date stamp of the current DRP. If the page's date stamp is not within the current DRP, then all extra tuples are removed from the page and all recovery-unit sequence numbers on the remaining tuples are set back to zero. By delaying the removal of extra tuples after the change of a DRP, extra I/O operations are not required during the change from one DRP to another. Thus, changing the length of a DRP can be performed at any time since it does not cause an immediate change in the actual storage structure.

If it is decided that the expense of keeping previous versions of tuples in the MOD file for a particular relation can not be justified, then by setting either the DRP length or the maximum number of extra tuples permitted during a DRP to zero, the user can indicate that previous versions of tuples are not to be saved. Then, however, the only possible way to roll back a relation is through the use of the journal file.

3.10.3 Relation Consistency

Several methods can be used to ensure the consistency of a relation as it is being modified. At the beginning of a recovery unit, a "relation consistency" flag is set to indicate that the relation is consistent for the previous recovery unit but that a new recovery unit is beginning. (The setting of this flag involves writing a record that indicates that the relation is about to be modified to secondary storage. When the modifications are complete, the record is modified to indicate the successful completion of the recovery unit. When the relation is next processed, the record is retrieved in order to determine whether or not the modifications made in the last recovery unit were completed successfully. This is the strategy used quite successfully in MANTES [FERC78b].) During the recovery unit, if it becomes necessary to save a page from the buffer pool back to the file, first the corresponding page in the storage-

management table is saved, then the modified data page is saved. If the storage-management table is not saved first, then in the event of a system failure between the time that the data page is saved and the time that the SMT page is saved, the data page on disk contains modifications that are not indicated in the storage-management tables. Then, if changes are rolled back, the changes to the data page are not processed, leaving the relation in an inconsistent state. Thus, before a data page can be written, it is important to save the corresponding storage-management table entry.

At the end of a recovery unit, the modified pages in the storage-management tables are saved first, then any modified pages which remain in the buffer area are saved, and finally the relation consistency flag is set to indicate that the current recovery unit completed successfully. Should the system fail at any time before the final setting of the flag, it is possible to tell that the current recovery unit did not complete successfully. If necessary, changes made to the relation during the aborted recovery unit can, using the storage-management tables, be rolled back.

3.11 Relation Reorganization

The reorganization of a relation is one of the more crucial operations. The purpose of reorganizing a relation

is to move tuples in order to provide faster access. For example, the tuples in a MOD file can be merged with the tuples in the corresponding BASE file to create an updated BASE file and an empty MOD file.

There are two main types of reorganization: a partial reorganization where the tuples in a relation are moved from one page to another but TID's are not changed, and a complete reorganization where tuples are moved and the TID's are reassigned. A partial reorganization does not affect any other relations since the TID's are not changed. A complete reorganization causes other relations which use the TID's in the reorganized relation to be updated with the new TID's.

Performing a partial reorganization can be divided into parts based on the three types of tuples in the MOD file.

If a tuple is marked as inserted and deleted, it can be removed from the MOD file. If a tuple is marked as deleted but not inserted, the corresponding tuple in the BASE file is removed. This operation causes some space to be freed in the BASE page.

If a tuple is marked only as modified, then the corresponding tuple in the BASE file is replaced. This operation does not cause the amount of free space in the BASE page to change unless the length of the tuple was changed.

If a tuple is marked as inserted but not deleted, then inserting it into the corresponding BASE page requires some free space. If sufficient space is made available by some deletions, then the tuple can be moved to the BASE page. However, the storage system can not rely on there being sufficient deletions to make room for all insertions. If there is not enough free space in the BASE page to store all insertions, the following strategy is used by the storage system. First, all deletions and modifications for a page are performed. Then, as many inserted tuples as possible are moved to the BASE page and the number of tuples still in the MOD page is determined. If only a few tuples remain, they are left in the MOD page. Hopefully, there will be room for them in the BASE page the next time that the relation is reorganized. If many tuples still remain, then the BASE page is extended by splitting it into two or more segments. The SMT entry for the BASE page is modified to point to the list of entries for the logical page segments so that the required BASE-page segment can be located with only one data I/O request.

During a partial reorganization, as a pair of BASE and MOD pages is reorganized, access to those pages is not permitted. Access to the remainder of the relation, however, may be permitted.

During a reorganization, if sufficient tuples are

deleted, it may be possible to take some or all of the segments of a segmented BASE page and merge them into one segment. After the reorganization of a pair of BASE and MOD pages, the filter in the storage-management tables is recreated to reflect the new contents of the MOD page.

A complete reorganization of a relation causes the relation to be recreated. The entire relation is loaded into a new BASE file so that each BASE page fills a physical page. TID's (with their insert numbers equal to zero) are reassigned. As each tuple is inserted, its previous and new TID's are recorded in a special file. After the loading is complete, all other relations that contain references (pointers) to the reorganized relation are modified using the special file of old and new TID's. During the complete reorganization of a relation, it is necessary to lock out all access to the relation being reorganized. This implies that relations that reference the relation being reorganized must also be locked.

Normally, the complete reorganization of a relation would rarely be necessary. Since the storage system maximizes the amount of data stored on each physical page (by storing small logical pages on the same physical page) and minimizes the number of page accesses required (through the use of logical page segments and the storage-management tables), the complete reorganization of a relation would not

necessarily improve the storage structure of the relation. However, a complete reorganization of a relation is necessary if the definition of the primary key of the relation is changed and the TID is to reflect the new primary-key order. During such a reorganization, the relation must be sorted by the new primary key before it is reloaded.

3.12 Special Relations

A type of data that has not yet been examined is the description of relations and their component domains. Each relation in the system is described and its description is stored in a relation that contains a description of all relations: the "relation relation". Each tuple in this relation contains the description of a relation. The description of a relation consists of information such as the relation name, the names of the data sets on which the relation is stored, the names of the domains which make up the relation, etc. The tuple describing a relation also contains system information such as the date stamp of the current DBP, the number of tuples in MOD pages, the number of tuples in BASE pages, the number of extra tuples in MOD pages, the relation-consistency flag, etc. Similarly, the description of each domain is stored in a "domain relation". The casual user is not permitted to access these relations but the DBA and the RDBMS itself can access these relations in order to add, delete, and modify relations and to extract

the definition of a relation. These data can be processed using the normal tuple-manipulation routines, so it is not necessary to write special routines in order to be able to process the relation and domain descriptions. A consequence of storing relation descriptions in a relation is that each relation can (internally) be uniquely identified: by its TID in the relation relation.

3.13 Storage-Management Tables

In this section, we summarize the information that is stored in the storage-management tables since the tables are of major importance in the storage system.

For each page, the storage-management table entry contains: the page reference number of the BASE page, the page reference number of the corresponding MOD page (which is zero if there is no such page), the TID filter, and the maximum recovery-unit sequence number within the current direct recovery period for the MOD page. The format of each entry in the SMT is illustrated in Figure 3.5.

Figure 3.5 SMT Entry Format

(In Chapter 4, we describe additional information that may be added to the storage-management tables in order to permit

the efficient prccessing of tuples by primary key.)

If a logical page is stored in more than one segment, the PRN of the page is replaced by a pointer into a list of entries. There is one entry for each logical page segment and each entry contains the maximum TID on the segment and the PRN of the segment.

For each relatien, the number of entries on each page of the storage-management tables and the size of each table entry are fixed. Thus, the logical page number can be used as a subscript into the storage-management tables. The size of each field in each SMT entry for a particular relation is fixed but the size may vary from relation to relation. Thus, in small files, the PRN's may be represented in two bytes while in larger files, three or four bytes may be required for each PRN. If necessary, the storage-management tables can be reorganzied in order to change the size of an entry or to change the number of entries on each page in the storage-management tables.

With the definition of the storage-management tables, we have created a second level of indirect page addresses since the DMT (device management table) used by the device system also uses indirect page addresses. However, in the interests of efficiency, the SMT and the permanent DMT for a relatien are merged intc one table. The PRN's in the SMT are replaced by the device addresses in the DMT. The

storage system then requests pages by device address instead of PRN. It is still be possible for the device address to point to an entry in a temporary DMT instead of to an actual device. However, since both the device system and the storage system access the merged table, the two systems are no longer independent.

Chapter 4: Access-Path System

4.1 Introduction

In this chapter, we develop the access-path system of the data-management system. First, we examine access paths in general. Then, we indicate which access paths provide the most power for relational data bases.

4.2 Access Paths

The purpose of an access path is to provide access to tuples in relations. For example, if a user wishes to know which students received "A" grades, one or more access paths are used to determine which tuples satisfy this qualification.

A key domain is a domain by which access to tuples is permitted, and access paths are defined for key domains. (Theoretically, in the relational data model, all domains are key domains, but, for practical applications, it may be too expensive to permit access to tuples by all domains. Thus, a subset of the domains may be designated as key domains.) Key domains are either unique or not unique within a relation. Candidate keys are unique within a relation since they uniquely identify each tuple. Secondary keys may be either unique or not unique.

Hsiao and Harary [HSIA70] define three basic types of

access paths: sequential scans, links, and directories. Other access paths are usually combinations of these three basic paths. A sequential scan involves examining each tuple in a subset of a relation. The sequential scan is quite fast if the tuples examined are all on the same page or a very small number of pages. The scan becomes expensive if it involves accessing many pages in the relation.

A link is a pointer from one tuple to another. The two tuples may be in the same or different relations. A link may be used to create a chain of related tuples: the first tuple in the chain is linked to the second tuple, the second tuple is linked to the third tuple, etc. These related tuples can be accessed quickly and efficiently by the storage system.

A directory (or index) contains qualifiers and a list of the tuples which satisfy each qualifier. Each qualifier contains a key value by which tuples are accessed. The qualifiers are always disjoint (the attribute value in a specific tuple can satisfy only one of the qualifiers). The qualifiers are either "dense" (all key-values currently in use are stored in the directory) or "non-dense" (only some of the key values are stored in the directory). A directory is used because it is normally much smaller than the associated relation, and so can be searched much faster than the relation.

One of the major considerations when defining the access paths to be used in a RDBMS is that, for a particular relation, the access paths will probably have to be modified at some time. There are two reasons for this. The first is that initially the wrong choice of access paths may be made. Only after usage patterns emerge can it be determined whether or not the initial choice was a good one. Secondly, usage patterns themselves change and access paths that once were suitable may become unsuitable. Thus, it is important that the DBA be able to add, delete, and modify access paths as necessary without having to unload and reload the relation itself.

4.3 Single-Attribute Access Paths

In this section, we examine "single attribute" access paths, that is, access paths that are used when processing queries with only one attribute in the qualifier. For example, qualifiers of the form

(PART-NUMBER = 'WRENCH')

can be processed using a single-attribute access path.

4.3.1 Primary-Key Index

A primary-key index contains entries in the index for some or all of the primary keys in a relation. If the relation is ordered by primary key, then the index need only contain entries of the form:

<KEY, PAGE#>

where KEY is the highest primary key on a page and PAGE# is the logical page number of that page. When searching such a non-dense primary-key index, the search is continued until an index entry with a key that is greater than or equal to the required key is found. Then, the correponding page is read and is searched for the tuple. This type of access path uses both a directory and a sequential scan. The advantage of the non-dense primary-key index is that there is only one index entry per page instead of one entry per tuple. This reduction of the number of index entries required results in a much smaller index than is otherwise possible. The major disadvantage of primary-key indexes is that the relation itself must be examined in order to determine whether or not a particular tuple exists.

4.3.2 Secondary-Key Indexes

There are many different types of secondary-key indexes. We shall examine the two most common types of secondary-key index: the multilist and the inverted list.

In the (basic) multilist, there is an index entry for each unique secondary-key value in a relation. Each index entry is of the form

<KEY, TID> .

If a particular secondary key is not unique, then the tuples with that secondary key are linked together in a chain which

starts at the index entry. The format of a multilist is illustrated in Figure 4.1.

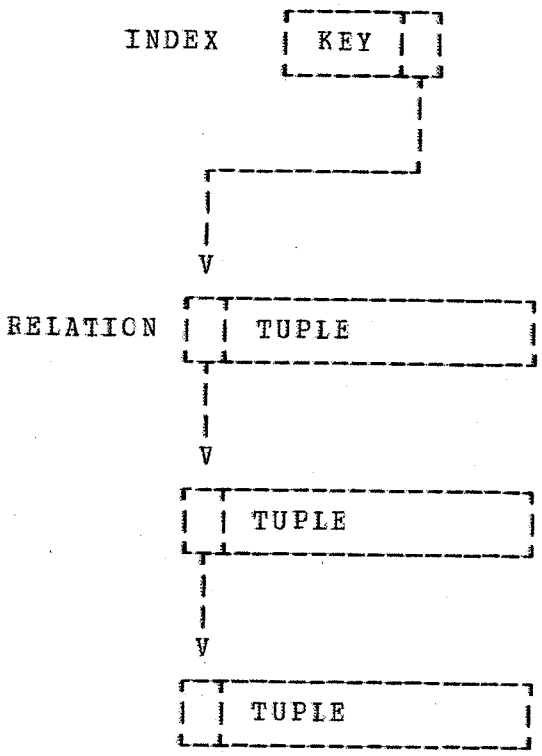


Figure 4.1 Multilist

This type of index uses a directory and a link. The major disadvantage of the multilist is that links are stored in the relation. Thus, in order to obtain the TID's of all tuples with a specific secondary key, it is necessary to access both the index and the relation. This process becomes very expensive if several such chains must be followed in order to find the intersection of the lists. Also, in order to update the index, it may be necessary to

modify part of the relation. Having to modify both the index and the relation creates consistency problems. For example, if the index is modified but the relation is not modified (due to a system error), then tuples that should be included may be missing from a chain or tuples that should not be included may be present on a chain. In order to guarantee the integrity of the chains themselves, it is normally necessary to maintain both forward and backward links so that if one link is destroyed, the chain can be recreated by processing it from the other direction [MART77].

There are many other versions of the basic multilist, such as the cellular multilist, etc., but they all share these fundamental disadvantages.

The inverted list is another type of secondary-key index. Like the basic multilist, the inverted list also contains one index entry for each unique secondary-key value in a relation. However, each index entry is of the form

<KEY, TID-LIST>

where TID-LIST is a variable-length list of the TID's of all tuples with the given secondary-key value. The format of an inverted list is illustrated in Figure 4.2.

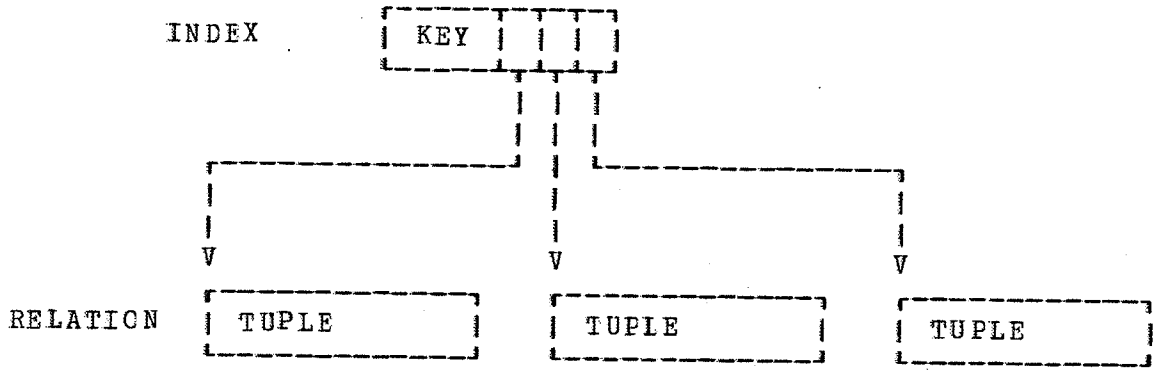


Figure 4.2 Inverted List

Because the inverted list contains more information than the multilist, the inverted-list index is larger and a search of the index requires more time than a search of the corresponding multilist. However, the total amount of (index and relation) space required is approximately the same for both the multilist and the inverted list. The major advantage of the inverted list is that the access path and the data are stored separately so that it is not necessary to access the relation itself when retrieving the list of tuples with a given secondary key or when modifying the inverted list.

The separation of access path from the data makes it easier to modify the data structure as old access paths become unnecessary and must be replaced by new access paths. This type of logical reorganization is difficult when the access paths are stored with the data. For example, in IMS [IBM74c], to delete an access path (logical-child segment)

from one data base to another data base and replace it with another access path, it is necessary to unload the data base and reload it with the new access path. For a large data base, this process is very expensive, not only because of the CPU time required to perform the operation, but also because the data base is not available to users during the operation.

4.4 Multi-Attribute Access Paths

Primary-key indexes and secondary-key indexes provide fast access to tuples when a query refers to only one attribute. However, if a query refers to more than one attribute, then the processing becomes more complicated. For example, if single-attribute access paths are used to process

(PART-NUMBER='WRENCH' AND COLOUR='BLUE')

it is necessary to search the PART-NAME access path and the COLOUR access path, and then take the intersection of the two access-path TID lists. This process is expensive if the individual TID lists are long but there are only a few tuples in the intersection of the lists. As the number of attributes specified in a query increases, the processing becomes even more complicated.

In this section, we examine "multi-attribute" access paths: access paths that can be used to process queries which reference several attributes.

4.4.1 Combined Indexes

A combined index is a collection of indexes (inverted lists) for a set of attributes [LUM70]. Each index contains a different ordering of the attributes so that all possible attribute combinations occur at the left of one of the indexes. For example, if a relation contains three domains A, B, and C, which are all frequently used in the same qualification, then a combined index could be defined for the three attributes. One of the indexes would contain entries of the form

<A, B, C: TID-LIST> .

There is one entry in the index for each combination of the three attributes. These entries are ordered first on the "A" attribute, then on the "B" attribute, and finally on the "C" attribute. Queries of the form

(A = 'x' and B = 'y' and C = 'z')

can be answered easily by searching this index. The index can also be used to answer queries involving the attributes A and B, and queries involving only the attribute A. However, in order to be able to answer queries about the other attribute combinations (A and C, B and C, B, and C), it is necessary to define two more indexes:

<B, C, A: TID-LIST> and <C, A, B: TID-LIST> .

The three indexes comprise the combined index. The three versions of the index are different only in the ordering of

the index entries; there is nc logical difference in their contents. With these three indexes, we can answer queries involving any combination of the attributes A, B, and C. In general, if we wish to be able to index on N attributes, then the combined index contains C(N,K) (the number of combinations of N elements, taking K elements at a time, where K is the smallest integer greater than or equal to N/2) indexes [LUM70].

The advantage of a combined index is that only one index must be searched in crder to evaluate a query involving up to and including the N indexed attributes. However, there are two major disadvantages to using a combined index. The obvious one is that a large amount of storage is used to store the different versions of the index. A second disadvantage is that when an attribute value is modified, the change must be made in each version of the indexes. This is a major problem if attribute values are modified frequently.

4.4.2 Modified Combined Index

In this section, we present a modification cf the combined index which eliminates the problem of having to update each version of the index each time that a change is made. In the modified combined index, there are still several versions of the index keys but there is only one copy of the TID list for each attribute combination. Each

index entry contains the attribute values and a pointer to a "bucket". The bucket contains the list of TID's for that attribute combination. Thus, all indexes share a common set of buckets. The format of the modified combined index is shown in Figure 4.3.

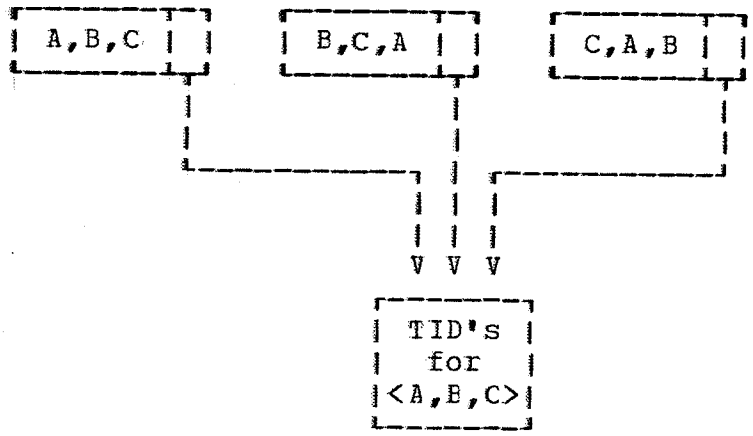


Figure 4.3 Modified Combined Index

Since most modifications to the index involve only changes to the buckets, not to the index itself, only one change must be made to the modified index. The only time that the indexes themselves are changed is when a new attribute combination is added to the relation. Then, the new attribute combination must be added to each index. When attribute combinations are deleted from a relation, instead of removing the index entries immediately, the index entries are kept in the index and the bucket contains a null list of TID's. Later, at a convenient time, these unused entries

are deleted from all of the indexes.

At the expense of one extra pointer for each attribute combination in each index, the overall size of the indexes is reduced (since the TID lists are stored in only one place) and the problem of having to modify all versions of the index each time that a change is made is eliminated.

4.4.3 Boolean Algebra Atoms

Wong and Chiang propose an index that consists of disjoint atoms that "cover" a relation [WONG71]. The atoms are Boolean expressions (involving the N attributes) which need never be broken down. Associated with each atom is a list of the tuples for which the expression in the atom is true. Queries expressed in Boolean algebra are then broken down into the corresponding queries involving the atoms in the index. Since the atoms are disjoint, it is not necessary to take the intersection of any of the TID lists in the index.

A major advantage of this method is that the Boolean algebra atoms can be tailored to each relation. For example, if certain attribute combinations always occur together, then the combinations could be defined as atoms. A disadvantage is that taking an arbitrary Boolean expression and breaking it down into the corresponding atoms is not necessarily trivial.

4.4.4 Multi-Attribute Hashing

Rothnie and Lozano define a method for multi-attribute retrieval which is based on hashing [ROTH74]. For each tuple in a relation, each of the keyed attributes is hashed with a (different) hashing function. The hashing produces a set of values called the "characteristic tuple". All tuples with the same characteristic tuple are then stored in a "cluster". (In the storage system, a cluster would be a logical page.) By using appropriate hashing functions, tuples which are likely to be accessed together can be stored in the same or adjacent clusters.

Two disadvantages of this technique are that the tuples are not ordered and that the storage of tuples is based on the hashing functions. If the hashing functions are changed, the entire relation must be unloaded and then reloaded.

4.4.5 Partitioning of Index Entries

There have been many other proposals in which the problems of multi-attribute queries are examined: Bentley and Finkel proposed quad trees and multidimensional search trees [BENT74]; Huang proposed data base graphs [HUAN73]; Michaels proposed partitioned multi-attribute indexes [MICH76]; Yao proposed random 2-3 trees [YAO78]. These schemes and the others already examined (modified combined index, multi-attribute hashing, boolean algebra atoms) that

use only one bucket for each attribute combination have a major problem: how to partition the buckets so that access by fewer than N attributes does not require a large number of index I/O requests. For example, suppose that we wish to index on two attributes A (with attribute values Ai: i=1,2,...,N) and B (with attribute values Bj: j=1,2,...,M). If there is room on each page for K buckets, then P=N*M/K bucket pages are required. (We assume that most of the N*M combinations occur. For practical purposes, the number of actual combinations may be much less than N*M.) Even though the number of pages may be quite large, access by both Ai and Bj values is quite fast because there is only one bucket for each (Ai,Bj) combination. If the buckets are ordered on the Ai values, then access by Ai alone is also fast since the Ai buckets are on the same or adjacent pages. However, if we try to access by a Bj value alone, the Bj buckets are spread over the P pages and access by a Bj value involves accessing all (or almost all) P pages.

If the (Ai,Bj) combinations are distributed over the bucket pages in the form of a Latin square (Latin squares are described, for example, by Street and Wallis [STRE77]), then it appears that most requests involving Ai or Bj involve accessing SQRT(P) bucket pages. This number of I/O requests is still unacceptably large.

The problem of partitioning the buckets becomes more

complex as the number of attributes increases, thus increasing the number of attribute combinations. As the number of bucket pages increases, the number of pages accessed while processing a query involving a subset of the possible attributes also increases.

4.4.6 Partial Combined Indexes

We now summarize multi-attribute query processing and indicate a preferred type of index for processing multi-attribute queries.

The use of single-attribute indexes for each key attribute is expensive if several keys are frequently specified in one query since the intersection of the TID lists must be determined. The advantage of single-attribute indexes is that each key appears in only one index so that only one index must be modified when changing a key value.

In the multi-attribute indexes which use only one set of buckets, the index provides fast retrieval when a query is of the same form as the index entries. However, if a query involves a subset of the indexed attributes, then many index entries may have to be accessed in order to satisfy a query and an excessive number of page transfers may be required if the index entries are not partitioned appropriately. This type of index does provide good facilities for update since a particular tuple appears in the TID list of only one index entry.

The combined index provides fast retrieval whether all attributes are specified or a subset of the attributes is specified. In the case of partial attribute retrieval, the TID lists are stored consecutively on the same or adjacent pages thus minimizing the number of pages accessed. The disadvantage of using combined indexes is that when an attribute value is changed, added, or deleted, all versions of the index must be appropiately modified.

In order to provide fast access to tuples while avoiding excessive update costs, the following hybrid scheme is used in the access-path system to provide a multi-attribute query capability. Instead of using a full combined index with all required attributes, a "partial combined index", several indexes each with a subset of the required attributes is maintained. This idea was proposed by Mullin [MULL71] and extended by Stonebraker [STON74] and Berra and Anderson [BERR77].

For example, suppose that we require an index on the 4 attributes A, B, C, and D. The full combined index contains an index for each of the combinations:

<A, B, C, D> <B, C, D, A> <C, D, A, B>
<D, A, B, C> <A, C, B, D> <B, D, A, C> .

Any query involving any of the four keys can be resolved using only one index access, but an update requires six

index accesses. Instead of using the full combined index, the following partial combined index could be used.

<A, B> <B, A> <C, D> <D, C>

This index provides retrieval with either one index access or two index accesses plus the intersection of two TID lists (an average of 1.60 accesses), and update of a single attribute value with only two index accesses. Thus, the power of the full combined index is available with little extra work when retrieving values and at a significant saving when the index is modified.

The partial combined index can be tailored to each relation and can be recreated as usage patterns change. For example, the index sets

<A, B> <B, C> <C, D> <D, A>

(with averages of 1.47 accesses for retrieval and 2.0 accesses for update) and the index sets

<A, B, C> <B, C, D> <C, A> <D>

(with averages of 1.40 accesses for retrieval and 2.25 accesses for update) both provide the necessary facilities to index on any combination of the attributes A, B, C, and D, but in a slightly different manner than the full combined index. By maintaining statistics on the combinations of attributes used in queries, the DBA can rearrange the indexes as necessary in order to minimize the total number of index accesses required. For example, if most index

requests are retrieval requests, then a full combined index could be used; while if most index requests are update requests, then single-attribute indexes on each attribute could be used; otherwise, an appropriate partial index could be used.

4.5 Multiple-Relation Access-Paths

In the previous sections, we examined "single-relation" access paths: access paths that provide access to data within one relation. In this section, we indicate how access paths for several relations can be combined into one access path, the "multiple-relation" access path, in order to provide a more efficient access path. (A similar but more restrictive version of the multiple-relation access path has independently been defined by Haerder [HAER78].)

Access paths are normally defined for individual relations. However, if several relations in the same data base have access paths defined for the same attribute, then the access paths can be combined into one access path. For a particular attribute, instead of defining one access path for each relation in which the attribute is defined, a "multiple-relation" access path is defined for the data base. The multiple-relation access path contains the information that is normally distributed over the individual access paths and this access path is shared by the various relations. For example, if it is necessary to define an

access path for S# in both the relation S and the relation SC, a multiple relation access path can be defined. Each entry in the multiple-relation access path contains the following information:

<S#: S-TID-LIST; SC-TID-LIST> .

The first list contains the TID's of tuples in the relation S with the given value of S#, and the second list contains the TID's of tuples in the relation SC with the given value of S#. Both the lists of TID's may vary in length (however, if S# is the primary key for the relation S, then there is only one TID in S-TID-LIST for each value of S#). The resulting access path is smaller than the sum of the two individual access paths for S# since the keys (S# values) are specified only once. However, the time required to search the access path may be slightly longer because the access path contains more information.

An important reason for defining multiple paths within an access path is the information that can be inferred from the access path. For example, the access path on S# for the relations S and SC permits the RDBMS to determine which students are or are not enrolled in at least one course. The access path also defines the relational algebra "join" of the relations S and SC on the domain S#. (The relational algebra join of two relations is described by Date [DATE77].)

In general, we define a multiple-relation access path on a "major key", one or more attributes in a data base for which access paths are required. We can then define individual paths to specific relations within the access path. For each path, a "minor key" may also be defined. A minor key is one or more attributes which are used to subdivide the list of TID's for each major-key value. Each access-path entry has the format:

<major-key-value: path-entry1; path-entry2; ...>

where each path entry has the format:

TID-LIST

or

minor-key-value: TID-LIST; ...

Tuples may be retrieved by specifying any number of the left-most attributes in the major key and, optionally, the minor key. If both the major key and the minor key are specified, then the list of TID's associated with the given major key/minor key pair is returned. If only the major key is specified, then the lists of TID's associated with all of the minor keys for the given major key are returned. If only the leftmost domains in the major key are specified, then the lists of TID's from all of the access-path entries with the given major-key prefix are returned.

Depending on the number of entries with a particular major-key prefix, it may be possible to process queries in

which not all of the specified attributes are defined at the left of the key. For example, if the key

<K1,K2,K3,K4,K5>

is defined (the division of the key into major and minor components is not significant), then queries involving K1, K2, and K5 but not K3 or K4 can be evaluated by examining all access-path entries that begin with the specified K1 and K2 values. The required TID lists are no longer adjacent but as long as the number of entries with the given K1/K2 values is not large, the request can be satisfied quickly.

The multiple-relation access path can be used to implement a path in the partial index. For example, if a path is required for <A, B, C>, the attributes A, B, and C are divided to form a major, and, optionally, a minor key. Normally, A is chosen as the major key and B and C are the minor key; this ordering permits other paths to be defined in this access path for A. If A and B are chosen as the major key, the access path can be shared only by paths which also use A and B as the major key; paths which use only A as the major key must be defined in another access path.

The use of the multiple-relation access path does not avoid the problem of having to maintain the different combinations of the combined index. The access path only permits the merging of several access paths with a common attribute prefix into one access path.

The multiple-relation access path could be extended to a "data base" access path that contains all access paths for a particular data base. By adding a "type indicator" to each access-path entry, the entries for all access paths in a data base could be merged into one large access path. Access-path entries would then be accessed by key value and type. The data base access path would contain fewer pages than the associated multiple-relation access paths but would probably cause the number of access-path pages examined during the processing of a given request to increase.

4.6 Access-Path Structure

In this section, we describe the structure of the access path which provides the foundation for the access-path system. The access path used by the access-path system is a directory that is designed so that it could be used for any of the access paths described earlier in this chapter. One general set of routines is needed to extract the contents of an access-path entry while specialized routines for each of the desired access paths are used to interpret the access-path contents.

In order to reduce the number of access-path pages examined while searching the access path, the access path is structured in the form of a B-tree [KNUT73]. A B-tree, as illustrated in Figure 4.4, is a multilevel tree. The higher levels of the tree are used only to reduce the number of

pages accessed before the desired page in the lowest level of the tree is found.

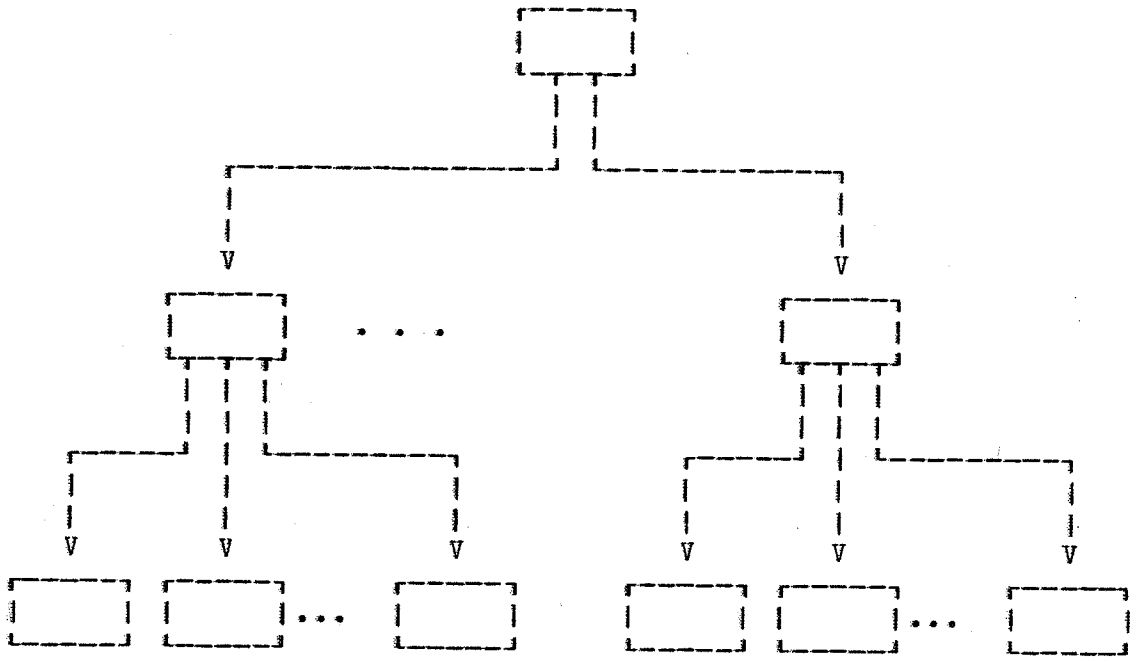


Figure 4.4 B-Tree

The lowest level of the access path contains the actual access-path entries: these entries contain major-key values and the associated minor-key values and variable-length lists of TID's. With this access-path format, it is possible to define primary-key indexes, inverted lists, combined indexes, etc. Each access-path entry is stored in a tuple in an access-path relation. By storing entries in tuples, the storage system is used to retrieve, store, delete, and insert entries for the access-path system. The access-path entries are stored in ascending order of major

key.

If the lowest level of the access path is stored on more than one page, then one or more higher levels are added to the access path to form a B-tree. Each entry in the higher levels of the access path contains the highest key on a page in the next lower level of the access path and a pointer to that page. The process of adding higher levels to the access path is continued until all the access-path entries at the highest level can be stored in one page.

In general, if there are N data pages in a relation and K access-path entries can be stored in each access-path page, then the access path contains approximately

LOG(N) / LOG(K)

levels (including the lowest level of the access path). The number of levels in the access path indicates the number of access-path pages that must be examined when retrieving a tuple. For example, if we assume that a relation contains 500,000 tuples, that 50 tuples can be stored on each data page, and that 100 access-path entries can be stored on each access-path page, then the access path is only two levels deep and so a tuple can be located after examining only two pages in the access path.

The format of the access-path entries in the higher levels of the access path is

<KEY: TID> .

The TID in the access-path entry points to the first access-path entry on the corresponding page at the next lower level in the access path. The keys in the higher-level access-path entries can be compressed both at the beginning and at the end as discussed in [BAYE77] and [IBM73b]. For example, if the highest key on a page is

<74120, 100, T10>

and the lowest key on the next page is

<74206, 200, T5>

then the key "74120,100,T10" can be compressed on the right to "741". If required, this key can also be compressed on the left to remove the characters "74". However, the number of characters removed from the front of the key must be included with the key in order to be able to compare keys correctly. This key is sufficient to distinguish between the two lower-level pages. At the lowest level of the access path, key compression is not used since it is necessary to know the exact value of the key.

The access-path retrieval routine returns the list of tuples that contain a particular attribute combination. If a request includes only the leading K keys, where K < N (N is the number of attributes for which the access path is defined), then the TID lists from several access-path entries are merged and returned. For example, if access-path entries contain:

<74120, 100, I1: TID3, TID10>

<74120, 100, I3: TID5, TID7, TID9>

<74120, 200, I1: TID6>

then a request invclving the keys

<74120, 100>

returns the TID list

TID3, TID5, TID7, TID9, TID10

and a request involving the the key

<74120>

returns the TID list

TID3, TID5, TID6, TID7, TID9, TID10 .

An extension that could be made to the multi-level access-path is the "generalized index" proposed by Held [HELD75]. The generalized index is a combination of the multi-level index and order-preserving functions. At each level in the multi-level index, there is one index entry for each page at the next lower level of the index. With an order-preserving function, there is only one index entry for the entire lower level (that index entry is the definition of the function). Normally, it is not possible to define one order-preserving function that can eliminate an entire level in the index; so the generalized index combines order-preserving functions and the multi-level index. By defining the appropriate functions, it should be possible to

reduce the number of higher-level index entries required. In the worst case, the generalized index would be the same as the equivalent multi-level index. In the best case, the generalized index would contain only one order-preserving function.

4.7 Maintenance of Access Paths

We now examine the effects of modifying tuples in relations. When a tuple is inserted in a relation, all access paths for that relation must be modified. This involves either inserting a new access-path entry or adding the TID of the new tuple to an existing access-path entry. If an attribute value is modified, then the old attribute value must be removed from all of its access-path entries and the new value must be added to the corresponding access-path entries. (It should be noted that it may be too expensive to keep all access paths up to date. If an access path is used infrequently, instead of updating it, it could be marked as no longer up to date. Then when next needed, it is recreated.)

Since access paths are actually manipulated as relations, BASE and MOD files and DRP's can be used to maintain the integrity of access paths. Thus, a master copy of each access path is stored in the BASE file and all changes are made to the MOD file. When changes are made to a MOD page, all copies of the access-path entry within the current

direct-recovery period are kept in the page. If it is necessary to back out some of the changes in a relation, the corresponding access-path changes can also be backed out of the access path quite easily. If a particular access path is quite volatile, then it could be stored in only a MOD file with a small (or no) direct-recovery period; the extra storage required the multiple copies of tuples in the MOD file is eliminated.

In a large B-tree, if many changes are made to one area of the tree, it may be necessary to move some of the access-path entries to unused pages in order to create the space needed for the changes. The moving of tuples can cause the tree to become unbalanced, with some of the paths in the tree being longer than others. Also, when entries are moved from one page to another, one or more higher levels in the tree must be modified to reflect the new location of the moved entries and this may cause other modifications to the tree. However, in the access-path system, the depth of the access path never changes. When an access-path entry is added or modified and there is not enough free space for the change, the access-path page is automatically extended by the storage system (as described in Chapter 3). Thus, the routines in the access-path system are less complex than most directory-manipulating routines since they do not have to perform reorganizations caused by

changes "rippling" up to higher levels. The advantages gained are similar to those gained in Held and Stonebraker's "static index" [HELD78] but without the disadvantage of not being able to modify the static index.

Since there are no pointers into an access path, it is possible to perform a complete reorganization of an access path without affecting other relations. This type of reorganization is performed whenever it is convenient instead of when it is necessary as in B-trees.

4.8 Primary-Key Access

Primary-key access (using a non-dense index or a hashing function) creates additional problems for the storage system. The index or hashing function generates only the number of the logical page on which the tuple is stored, not the TID of the tuple. In order to find the tuple with the required primary key, the storage system must sequentially scan the tuples on the MOD page (which may be stored in more than one logical-page segment), and, if the tuple is not found, the storage system must then scan the tuples on the BASE page (which may also be stored in more than one logical-page segment). In order to reduce the number of page accesses required, the following information could be added to the storage-management tables to support efficient primary-key access.

To avoid accessing both the MOD file and then the BASE

file, a primary-key filter could be added to the storage-management tables. This filter is used in the same way that the TID filter is used: the primary key (or a portion of it) is hashed and the resulting bit string is compared with the filter value. If the two values match, then the tuple may be in the MOD page; if they do not match, then the tuple (if it exists) must be in the BASE page. There would be one primary-key filter in the SMT for each MOD page.

To determine which logical-page segment contains the required tuple, when a page is extended, the highest primary key on each logical-page segment could be included in the SMT. Thus, splitting a page into segments does not affect the efficiency of primary-key access since the required logical-page segment can be accessed with only one data I/O request. When hashed access is used, even though a large number of tuples may hash to the same logical page, the storage system ensures that access to the required logical-page segment is still efficient.

The storage system could also include the highest primary key of each group of tuples in the tuple index stored at the end of each logical page. This extra information would reduce the time required to find a tuple in a page when accessing the tuple by primary key.

The ability of the storage system to perform an efficient search of a page by primary key can be used

effectively by the access-path system to retrieve access-path entries. Each access-path entry is stored in a tuple with the major key of the entry used as the primary key. Routines to search a page for a particular access-path entry are not needed in the access-path system since the storage system already provides the required function.

Chapter 5: Retrieval System

5.1 Introduction

In this chapter, we develop the retrieval system of the data-management system. First, we define a data-manipulation language (DML) which can be used to provide associative access to the tuples in a relation. The language is sufficiently powerful to be used by itself when processing one relation at a time; however, it is designed for use in implementing DML's such as the relational calculus or relational algebra. We also examine how the retrieval-system requests are translated into the necessary access-path-system and storage-system requests.

5.2 Associative Access

One of the goals of the retrieval system is to free the user from having to know the details of how relations are stored and the access paths that are available. The user specifies a query, defining what he wants, and the retrieval system attempts to find the "best" (or only) method of processing the query. There are several reasons for using associative access to data. The major reason is that since access paths are dynamic, the user probably does not know (and should not be expected to know) all of the access paths that currently exist, and, may make a poor choice if allowed

to choose the access paths directly. It is expected that many users of the DML are higher-level systems and these systems should not have to know which access paths are available for a given relation.

It is possible that some requests can not be processed without an extremely large amount of work on the part of the retrieval system. For example, it may be necessary to scan a large relation one or more times in order to process a particular query. It should be the responsibility of the retrieval system to determine which requests are "reasonable" and which are not; this may require some knowledge of the request or the user. For example, a very slow request that is run only once a month is acceptable while the same request is rejected (until extra access paths are added) if it is run several times a day.

5.3 Relation Retrieval

In this section, we describe the facilities available for the associative retrieval of tuples in a single relation. The syntax of the retrieval-system commands is given in Appendix II.

The general form of the retrieval statement is

RETRIEVE WHERE(qualifier) .

(In order to keep the syntax of the statements as simple as possible, it is assumed that the relation and domains being processed have already been identified to the system.)

There are two types of qualifiers: the "single" qualifier is a qualifier which is processed by examining the tuples in a relation one tuple at a time; and the "multiple" qualifier is a qualifier which is processed by examining groups of tuples in a relation. In the following sections, we examine the formats of these two types of qualifiers.

5.3.1 Single-Tuple Processing

We now examine queries that can be processed by examining each tuple individually. The simplest form of the single qualifier is

Di relop 'V'

where "Di" is the domain being referenced and "relop" is one of the relaticnal operators. "V" is the value with which the domain is compared; we refer to "V" as a "simple value". In most of the following examples, we use the relational operator "=" since it is the operator most commonly used.

Qualifiers can contain several expressions joined together by the standard logical operators (AND, OR, NOT), and the order in which expressions are to be evaluated can be indicated by the use of parentheses. Thus, the qualifier

Di='V1' AND Dj='V2'

is true if both subexpressions are true.

The user may also specify a "range value" for a domain. A range value is defined by specifying the minimum value in the range and the maximum value in the range. For example,

in the qualifier

Di = 'V1':'V2'

the tuple is accepted if the domain value is between "V1" and "V2" or equal to either value.

In order to make ranges as general as possible, we adopt the following conventions: if the lower range value is not supplied, the smallest possible value is assumed; if the upper range value is not supplied, the largest possible value is assumed. Thus, the qualifier

Di = :'V2'

has the same effect as the qualifier

Di ≤ 'V2'

and the qualifier

Di = 'V1':

has the same effect as the qualifier

Di ≥ 'V1' .

In order to select tuples with one of several attribute values, the user can include a domain "value list" in an expression. A value list consists of any combination of simple values and/or range values, separated by commas if there is more than one value. (We assume that all elements in the value list are mutually exclusive. If some elements are not mutually exclusive, this may cause results to be unpredictable.) For example, the qualifier

Di = 'V1', 'V2'

causes the tuple to be selected if Di is equal to either of the values specified. The qualifier

Di = 'V1':'V2', 'V3', 'V4':'V5', 'V6'

causes the tuple to be selected if Di is equal to any one of the expressions listed.

If two or more domains are frequently referenced together, they can be concatenated. The qualifier

Di,Dj = 'V1','V2'

is equivalent to the qualifier

Di='V1' AND Dj='V2' .

When domains are concatenated, each entry in the value list must contain the appropriate number of concatentated values. The concatenation of domains makes it much easier to specify value lists. For example, the qualifier

Di,Dj = 'Vi1','Vj1' : 'Vi2','Vj2', 'Vi3','Vj3'

is much simpler than the equivalent qualifier without the concatenation.

5.3.2 Multiple-Tuple Processing

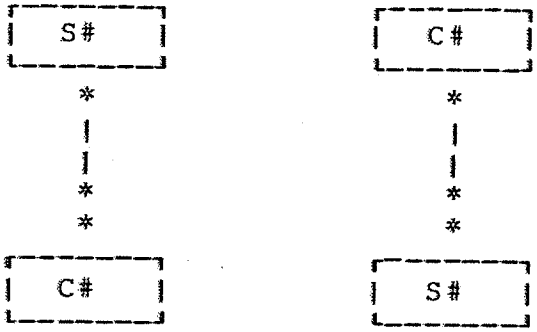
The facilities described in the preceding section can be provided by the retrieval system by examining tuples individually. In this section, we examine multiple qualifiers, qualifiers that can be used to examine sets of tuples.

Frequently, the tuples in a relation contain an attribute value that occurs in more than one tuple. At times, it

is convenient for the user to view such a relation as a hierarchical structure. For example, the relation

SC(S#, C#, GRADE)

could be viewed as



(These are not the only hierarchical relationships that can be defined for this relation.)

When viewing a relation as a hierarchy, the user can ask such questions as "Which students are enrolled in courses 74307 and 74308?". It is not possible to write one single qualifier which permits the user to ask such questions. In general, the user may wish to know which parents contain certain children. The notation used to define such qualifiers is

Dp.Dc = {child value set}

where "Dp" is the parent domain, "Dc" is the child domain, and the "child value set" is a value list. Both Dp and Dc can be concatenated domains. The individual values in the child value set can again be any combination of simple values and range values. The qualifier which corresponds to

"Which students are enrolled in both 74307 and 74308?" is

S#.C# = {'74307', '74308'} .

The qualifier

S#.C# = {'74200':'74299', '74300':'74399'}

illustrates the use of ranges in the child value set. The qualifier is used to determine which students are taking at least one second-year course and at least one third-year course in department "74".

When a child value set is processed, all of the tuples in the child value set are returned for each qualifying parent. For example, given the tuples

100 74306
100 74307
100 74308
100 74410

the qualifier

S#.C# = {'74307', '74308'}

causes the tuples

100 74307
100 74308

to be returned.

In general, the evaluation of a qualifier which contains a child set involves examining each parent domain and determining whether or not all the children defined in the value list occur under the parent.

5.3.3 Quotas

In order to extend the power of the DML, we now introduce "quotas". A quota is used to specify the exact number of subexpressions in a value list that must be true for the entire expression to be true. A qualifier written with a quota can always be written in an equivalent form without a quota, but the use of quotas frequently simplifies the writting of qualifiers. The general form of a quota is

Q(value list) .

The value list consists of any combination of simple values and range values. The following qualifiers illustrate the use of quotas. For the qualifier

Q(2) (S#='100', C#='74307')

to be true, both subexpressions must be true. This qualifier could be rewritten as

S#='100' AND C#='74307' .

If the qualifier is changed to

Q(1) (S#='100', C#='74307')

then the expression is true if either subexpression is true but both subexpressions are not true. To get the same effect as using the logical operator OR, the quota must be changed to Q(1:2) or to Q(1:). The expression is then true if either subexpression is true or if both subexpressions are true.

The use of quotas is especially convenient in lists

containing three or more subexpressions. For example, the qualifier

```
Q(2:)(S#='100', C#='74307', GRADE='A')
```

is expressed quite simply using quotas. The equivalent expression without quotas becomes much more complex.

```
(S#='100' AND (C#='74307' OR GRADE='A'))
   OR (C#='74307' AND GRADE='A') .
```

If the quota Q(2:) in the previous qualifier is changed to Q(2), the equivalent expression without a quota becomes even more complex (and unreadable), requiring three logical expressions, each containing three subexpressions.

The use of quotas is also permitted in multiple processing. The qualifier

```
S#.C# = {'74307', '74308'}
```

can be rewritten as

```
S#.C# = Q(2){'74307', '74308'} .
```

The qualifier

```
S#.C# = Q(2:){'74307', '74308', '74410'}
```

specifies that a student is selected if he is enrolled in at least two of the three courses specified.

In general, quotas used with child value sets are evaluated as follows. A counter is maintained for each of the subexpressions in the value list. For each parent, each child is examined; if a child satisfies a subexpression, the counter for that subexpression is incremented. (We assumed

that all subexpressions in a value list are mutually exclusive so a child would satisfy only one subexpression and the evaluation of an expression would terminate as soon as one subexpression is found to be true. Thus, if subexpressions are not mutually exclusive, only the counter for the first true subexpression is incremented.) After the last child for a particular parent is processed, the number of counters that are non-zero is determined. This value is then compared with the quota value list, and, if the value matches one of the quota value list subexpressions, the result of the expression is "true".

5.3.4 Counts

The last addition to qualifiers in the DML is the "count" parameter. Counts are used to define child-set queries which involve the number of children under each parent. Queries such as "Which students received only A grades?" can not be answered with the standard child-set qualifiers or with quotas.

The general form of the count parameter is

C(true value list; false value list) .

Both the "true value list" and the "false value list" are value lists which can contain any combination of simple values and range values. The true list is used to specify the number of children that must satisfy the corresponding domain value list for the expression to be true, and the

false list specifies the number of children that need not satisfy the domain value list for the expression still to be true. We shall assume that if the true list or the false list is not specified, then the count for that list may be any value.

We now examine some qualifiers that use the count parameter. The qualifier

S#.GRADE = C(1:) {'A'}

can be used to determine which students received at least one A grade and any number of grades that are not A's. The qualifier

S#.GRADE = C(1: ;0) {'A'}

can be used to determine which students received only A grades. The qualifier

S#.GRADE = C(;0) {'F'}

can be used to determine which students have not received any F grades.

The use of counts with sets has an interesting side result. If a user wishes to select a parent based on the number of children the parent has, then a count can be used with a null child set if we adopt the convention that a null child set is always true. For example, the qualifier

S#.C# = C(5:) { }

can be used to select students enrolled in at least 5 courses.

The processing of the count parameter is performed as follows. For each parent, each child is examined. For each child that satisfies the domain value list, a "true counter" is incremented. For each child that does not satisfy the domain value list, a "false counter" is incremented. After the last child is processed, the true count is compared with the true value list and the false count is compared with the false value list. If both values are within the defined limits, then the expression is true; otherwise, the expression is false.

5.4 Relation Modification

In the previous sections, we examined the retrieval of tuples in a relation; in this section, we show how tuples can be inserted, deleted, and modified.

5.4.1 Tuple Insertion

The general form of the statement used to insert tuples is

INSERT WITH (modifier)

where the "modifier" has the same basic form as the qualifier defined for tuple retrieval. For example, to insert a tuple for student 100 in the SC relation, the modifier

S#='100' AND C#='74307' AND GRADE='A'

can be used.

When inserting a tuple, it is necessary to include at least the primary-key domains since they uniquely identify the tuple. For example, the modifier

S# = '100'

is not complete when inserting a tuple in the relation SC because the course number is not specified. (Codd has examined the problem of insertions into a relation and concludes that permitting insertions which specify a candidate key that is not the primary key may permit duplicate tuples to be inserted [CODD75].)

If several tuples are to be inserted at once, a child value set can be included in the modifier. For example, the modifier

S#='100' AND S#.C#={'74307', '74308'}

can be used to insert two tuples in SC.

The use of quotas and counts is not permitted in a modifier since those parameters are used only when processing tuples that already exist.

Domains which are not given values in the insertion statement are assigned "default values". (A default value for each domain is defined when the relation is defined.)

5.4.2 Tuple Deletion

The general form of the statement used to delete tuples is

DELETE WHERE(qualifier)

where the qualifier is the same as the qualifier defined for tuple retrieval. For example, the qualifier

S#='100' AND C#='74307'

can be used to delete a tuple in the SC relation.

When deleting tuples, the primary-key domains do not need to be specified. For example, the qualifier

GRADE = 'F'

can be used to delete all tuples in SC with a grade of F.

If a child value set is included in the qualifier of a delete statement, for qualifying parents, the tuples in the child value set are deleted. For example, the qualifier

S#.C# = Q(2){'74307', '74308'}

causes the tuples 74307 and 74308 to be deleted from students enrolled in both courses. The qualifier

S#.C# = Q(1:){'74307', '74308'}

causes 74307 and 74308 tuples to be deleted from students enrolled in either course.

5.4.3 Tuple Modification

The general form of the statement used to modify tuples is

REPLACE WHERE(qualifier) WITH(modifier)

where the qualifier indicates the tuples to be modified and the modifier specifies the new domain values for the tuples to be modified. For example, the statement

REPLACE WHERE(S#='100' AND C#='74307') WITH(GRADE='A')

can be used to change a student's grade.

If not all of the primary-key domains are specified, then several tuples may be modified. For example, the statement

REPLACE WHERE(S#='100') WITH(GRADE='A')

causes all tuples for student 100 to be modified.

If a primary-key domain is modified, then the old tuple is deleted and a new tuple is inserted. For example, the statement

REPLACE WHERE(S#='100' AND C#='74307') WITH(C#='74308')

causes the tuple "100,74307" to be deleted and the tuple "100,74308" to be inserted. The domains not specified in the modifier are copied from the deleted tuple to the new tuple.

If a child value set is included in the qualifier of a replace statement, for qualifying parents, all tuples in the child value set are modified. For example, the statement

REPLACE WHERE(S#.C#={'74307','74308'}) WITH(GRADE='A')

causes all students enrolled in both 74307 and 74308 to be given A grades in both courses.

5.5 Strategy Relation

The most important part of the retrieval system is the selection of efficient access paths for each request. The selection of access paths should be dynamic because the type and number of access paths vary with time. However, trying

to determine dynamically the best set of access paths for a particular request can itself be time consuming. So, in the retrieval system, a compromise strategy is used. For each relation, a special relation, called the strategy relation, is maintained by the retrieval system. Each tuple in the strategy relation contains a list of domain names and a set of "paths". Each path contains the internal identifier of an access path, the type of the access path, and the names of the domains used with the access path. When processing a request, the retrieval system reduces the domains in the request to a canonical form and then retrieves the associated tuple from the strategy relation. The paths defined in the strategy-relation tuple are processed, from left to right, until the desired tuples are retrieved. For example, to process the request

```
RETRIEVE WHERE(S#='100' AND GRADE='A')
```

the strategy relation might indicate that a directory for S# can be used to establish a position at the first tuple for student 100 and that a subsequent sequential scan can be used to locate all of the tuples with an A grade.

The use of the strategy relation permits fast access to tuples by predetermined paths; this technique is more efficient than dynamically determining the best path for each request. Yet, access paths can be added, deleted, and modified at any time as long as the strategy relation is

also modified to reflect the new access paths. The speed of predetermined paths is provided with the flexibility of being able to change access paths at any time.

The strategy relation can also be used to maintain information on the frequency with which the various domain combinations are used if some extra domains are stored in each strategy-relation tuple. One domain contains the date on which the tuple was created and other domains contain the dates on which the tuple was most recently accessed for retrieval, insertion, deletion, and modification, and the number of times that the tuple was accessed for retrieval, insertion, deletion, and modification. These domains can be used when evaluating the access patterns for a relation.

When processing a request, if the necessary entry is not found in the strategy relation, then the request is rejected. Thus, the use of the strategy relation permits the DBA to prohibit certain requests by not defining the necessary entries in the strategy relation. Similarly, by adding to the strategy relation a domain which defines the minimum number of days that must pass before a request can be issued again, the DBA can control the frequency with which expensive requests are issued.

Chapter 6: Future Research and Conclusions

6.1 Future Research

In this thesis, the foundation for a generalized data-management system has been defined. There are, however, several functions of data-management systems that were mentioned but not examined in detail. In this section, we indicate some areas which deserve further examination.

We have assumed that the tuning of a data base is performed by the data base administrator. However, if the necessary statistics are maintained, it should be possible to have the data-management system itself perform much of the tuning. For example, pages that are frequently accessed together could be stored on the same data set; the length of the direct-recovery period of a relation could be modified as the rate of access to the relation increases or decreases; the access paths for a data base could be reorganized to suit current access patterns. The facility to add and delete access paths based on predefined future needs could also be supplied. (For example, adding temporary access paths at the end of each year could reduce the access-path processing required by year-end summary programs.) By adding such a tuning system, the data-management system would be able to overcome poor initial access-path

choices and provide the user with more efficient data-management services.

Another problem that needs to be examined is that of concurrent access to the data in a data base. Current operating systems provide few facilities to aid the data-management system in sharing data among users. Consequently, the data-management system must contain a mechanism for locking portions of a data base as changes are made to the data base. The locking mechanism must maintain the integrity of the data base when several users attempt to make changes to the same area of the data base without prohibiting access by users who wish to access a different area of the data base. Such a locking mechanism is required in order to be able to use data bases effectively. As more data are added to each data base, the number of users who require access to the data increases. It is the responsibility of the data-management system (and the operating system) to ensure that shared-data integrity is maintained.

6.2 Conclusions

In this thesis, we have defined a data-management system that provides addressed access to pages and tuples, keyed access to tuples using pre-defined access paths, and associative access to tuples. The device system manages the pages in each relation for the storage system. Through the use of the device-management tables, pages may be moved from

one data set to another without affecting the user. The ability to change the location of a page is important in large data bases where it is not economically practical to store all pages on fast devices. Instead, the access frequency of a page can be used to determine where the page is stored. The use of temporary device-management tables provides a simple method of recording the current location of a page as the page is moved from one location to another during its processing.

The storage system manipulates the tuples stored in relations. The storage system reduces the secondary storage required for a relation by storing the majority of the data in a relation in a BASE file (with no free space) and storing changes to the relation in a MOD file. The storage-management tables are used to define the current location of the BASE and MOD pages. Through the use of direct-recovery periods, the storage system also provides the facility to roll back groups of changes to a relation. This facility is important in a multi-user environment where changes made by one user may have to be backed out in order to preserve the integrity of the data base. The storage system is designed so that the use of BASE and MOD files and DRP's can be modified with a minimal amount of reorganization of the data base. Thus, as the users' requirements change, the facilities used may be changed.

The access-path system provides a powerful access-path structure, the multiple-relation access path, to support keyed access to data. The multiple-relation access path requires less storage and provides more information than the equivalent single-relation access paths. By not storing access-path links in the data portion of a relation, it is easy to add, delete, and reorganize access paths.

The retrieval-system DML provides associative access to the tuples in a relation. The use of the strategy relation provides a convenient method for translating associative queries into the necessary keyed requests to the access-path system. The statistics that are kept in the strategy relation are used when changes must be made to the access-path structure to improve access to the data base.

The major goals in the design of the system were to define a data-management system which provided both efficient storage of data and rapid retrieval of data and to permit the user to change data-management facilities used as his data-management needs change. Portions of the data-management system described in this thesis have been implemented and have been found to provide facilities at least as powerful as (and in many cases, more powerful than) the facilities provided by current data-management systems.

Appendix I: Variable-Length Values

In this appendix, we indicate how tuples, domains, and TID's can be stored efficiently as variable-length character strings.

In most systems, variable-length character strings contain a fixed-length prefix which contains the length of the character string. Such fixed-length prefixes either restrict the maximum length of a character string unnecessarily when too small or waste storage space when made larger than is necessary in order to permit the manipulation of the occasional large character string. For example, if the prefix is one byte, the maximum length of the associated character string is 256 bytes. If the prefix is two bytes, the maximum length of the character string is 65,536 bytes; however, for any string that is smaller than 256 bytes, one byte of the prefix is wasted.

In order to utilize storage space as efficiently as possible, variable-length prefixes can be used. The high-order N bits of the prefix can be used to indicate the length of the prefix. If all N bits are 1's, then the prefix is stored in two bytes; otherwise, the prefix is stored in one byte. Table I.1 indicates the maximum lengths that can be represented in one- and two-byte prefixes for various values of N.

N	1-byte length	2-byte length
1	128	32,768
2	192	16,384
3	224	8,192
4	240	4,096

Table I.1 Variable-Length Prefixes

With a variable-length prefix, the user is not restricted to domains with a small maximum length nor penalized because some domains may be quite long. The processing of the prefix requires the execution of extra instructions (approximately four Assembler language instructions on an IBM System 370) but this is a small price to pay considering the added flexibility provided.

It would be unusual to have tuples or domains whose lengths could not be represented in two bytes. However, if necessary, the size of the prefix can be increased to any size using the following strategy. The first N bits of the first byte of the prefix are divided into groups of bits with lengths M1, M2, ... If any of the first M1 bits are 0's, then the prefix is stored in one byte. If all of the first M1 bits are 1's and any of the next M2 bits are 0's, then the prefix is stored in two bytes. If the next M2 bits are also 1's but the next M3 are not all 1's, the prefix is stored in three bytes. This process can be continued for as

many bytes as necessary.

The TID can not be manipulated as a simple variable-length character string since the TID actually contains three values: the page number, the tuple number, and the insert number. Instead, these values are manipulated individually as variable-length values.

The page number and the tuple number can be stored with the high-order N bits indicating the number of bytes used to represent the value. The remainder of the first byte and any remaining bytes contain the actual value of the page or tuple number, not the length of the value. Thus, page numbers and tuple numbers of any magnitude can be represented efficiently.

An advantage of this representation of the page number and the tuple number is that two page numbers or tuple numbers can be compared by comparing the stored representation of the numbers instead of having to extract the represented values. For example, when a two-byte page number is compared with a one-byte page number, since the leading N bits of the two-byte value are all 1's while the leading N bits of the one-byte value are not all 1's, the two-byte value is designated as the larger value after only the first N (or fewer) bits are examined.

If the insert number is to be compared in its stored representation, it should not be stored in the same manner

as the page and tuple numbers since the insert number is a left-justified value while the page and tuple numbers are right-justified. (The fact that one insert number is stored in more bytes than another insert number does not mean that the first value is greater than the second value.) Thus, for two insert numbers to be compared efficiently, the length of the value must not be at the front of the value. Instead, the last bit of each byte can be used to indicate whether or not there is another byte following the current byte: if the last bit is not 1, then the current byte is the last byte used to represent the value; otherwise, there is at least one more byte in the value. Using this method of representation, insert numbers of any magnitude are stored efficiently and can be compared in their stored representation.

By storing the components of a TID adjacent to each other in their natural order (page number, tuple number, insert number), it is possible to compare TID's as character strings without having to extract and compare each component of the TID. This ease of comparison is important in both the storage system and the access-path system.

Appendix II: Syntax

```
<statement>     ::=  RETRIEVE WHERE( <qualifier> )
                   | DELETE WHERE( <qualifier> )
                   | INSERT WITH( <modifier> )
                   | REPLACE WHERE( <qualifier> )
                             WITH( <modifier> )

<qualifier>     ::=  <stexp2>
                   | <mtexp2>
                   | <mtexp2> AND <stexp2>

<modifier>      ::=  <stexp>
                   | <mtexp>
                   | <mtexp> AND <stexp>

<mtexp2>        ::=  <parent-child> <relop> <mtlist>
                   | ( <mtexp2> )

<mtlist>        ::=  { <value-list> }
                   | <count> { <value-list> }
                   | <count> { }
                   | <quota> { <value-list> }

<mtexp>         ::=  <parent-child> <relop> { <value-list> }
                   | ( <mtexp> )

<stexp2>        ::=  <stcl2>
                   | <stexp2> AND <stcl2>
                   | <stexp2> OR <stcl2>
                   | NOT <stexp2>
                   | <quota> ( <explist> )

<stexp>         ::=  <stcl>
                   | <stexp> AND <stcl>

<explist>       ::=  <stcl2>
                   | <explist> , <stcl2>
```

```
<stcl2>          ::=  <stcl>
                    |  ( <stexp2> )

<stcl>           ::=  <domain-list> <relop> <value-list>
                    |  ( <stexp> )

<count>          ::=  C( <value-list> ; <value-list> )
                    |  C( <value-list> ; )
                    |  C( ; <value-list> )

<quota>          ::=  Q( <value-list> )

<parent-child>   ::=  <domain-list> . <domain-list>

<domain-list>    ::=  <domain-name>
                    |  <domain-list> , <domain-name>

<value-list>     ::=  <value>
                    |  <value-list> , <value>

<value>          ::=  <domain-value>
                    |  <range-value>

<range-value>    ::=  <domain-value> : <domain-value>
                    |  <domain-value> :
                    |    : <domain-value>

<domain-value>   ::=  <simple-value>
                    |  <domain-value> , <simple-value>

<simple-value>   ::=  ' <character-string> '

<relop>          ::=  =
                    |  <
                    |  >
                    |  <=
                    |  >=
                    |  ¬=

<domain-name>    ::=  <character-string>
```

References

ANSI75 ANSI/X3/SPARC, "Study Group on Data Base Management Systems", Interim Report, ACM-SIGMOD FDT, Volume 7, Number 2, 1975.

ASTR76 Astrahan, M. M., et al., "System R: Relational Approach to Database Management", ACM-TODS, Volume 1, Number 2, pp. 97-137, June 1976.

BACH73 Bachman, C. W., "The Programmer as Navigator", CACM, Volume 16, Number 11, pp. 653-658, November 1973.

BAYE77 Bayer, R. and Unterauer, K., "Prefix B-Trees", ACM-TODS, Volume 2, Number 1, pp. 11-26, March 1977.

BENT74 Bentley, J. L. and Finkel, R. A., "Quad Trees A Data Structure For Retrieval on Composite Keys", Acta Informatica, Volume 4, pp. 1-9, Springer-Verlag, New York, New York, 1974.

BERR77 Berra, P. B. and Anderson, H. D., "Minimum Cost Selection of Secondary Indexes For Formatted Files", ACM-TODS, Volume 2, Number 1, pp. 68-90, March 1977.

BLOO70 Bloom, B. H., "Space/Time Trade-offs in Hash Coding with Allowable Errors", CACM, Volume 13, Number 7, pp. 422-426, July 1970.

BROD75 Brodie, M. L. et al., "ZETA: A Prototype Relational Data Base Management System", Technical Report CSRG-51, Department of Computer Science, University of Toronto, Toronto, Ontario, 1975.

CINC74 CINCOM Systems, "TOTAL/7 Reference Manual", CINCOM Systems Inc., Cincinnati, Ohio, 1974.

CODA71 CODASYL Comittee, "CODADSYL Data Base Task Group Report", Conference on Data Systems Languages, ACM, New York, New York, April 1971.

CODD70 Codd, E. F., "A Relational Model of Data for Large Shared Data Banks", CACM, Volume 13, Number 6, pp. 377-387, June 1970.

CODD72 Codd, E. F., "Relational Completeness of Data Base Sublanguages", Courant Computer Science Series, Volume 6, Database Systems, Prentice-Hall, Toronto, Ontario, 1972.

CODD75 Codd, E. F., "Understanding Relations - Installment 6", ACM SIGFDT, Volume 7, Number 1, pp. 1-4, 1975.

DATE77 Date, C. J., "An Introduction to Database Systems" Second Edition, Addision-Wesley, Don Mills, Ontario, 1977.

FERC78a Ferch, H. J., Neufeld, G. W., and Zarnke, C. R., "Mantes User Manual", Department of Computer Science, University of Manitoba, Winnipeg, Manitoba, 1978.

FERC78b Ferch, H. J., "The Design and Implementation of a Structured Indexed File System", Ph.D. Dissertation, Department of Computer Science, University of Manitoba, Winnipeg, Manitoba, 1978.

HAER78 Haerder, T., "Implementing a Generalized Access-Path Structure for a Relational Database System", ACM-TODS, Volume 3, Number 3, pp. 285-298, September 1978.

HELD75 Held, G., "Storage Structures for Relational Data Base Management Systems", Ph.D. Dissertation, Electrical Engineering and Computer Science, University of California at Berkeley, Berkeley, California, 1975.

HELD78 Held, G. and Stonebraker, M., "B-Trees Reexamined", CACM, Volume 21, Number 2, pp. 139-143, February 1978.

HOFF75 Hoffer, J., "A Clustering Approach to the Generation of Subfiles for the Design of a Computer Data Base", Ph.D. Dissertation, Department of Operations Research, Cornell University, Ithaca, New York, 1975.

HSIA70 Hsiao, D. and Harary, F., "A Formal System for Information Retrieval from Files", CACM, Volume 13, Number 2, pp. 67-73, February 1970.

HUAN73 Huang, J. C., "A Note on Information Organization and Storage", CACM, Volume 16, Number 7, pp. 406-410, July 1973.

IBM71 IBM, "IBM System/360 Operating System: Indexed Sequential Access Method, Program Logic Manual", IBM Corp., GY28-6618, 1971.

IBM73a IBM, "OS/VS Virtual Storage Access Method: Programmer's Guide", IBM Corp., GC26-3818, 1973.

IBM73b IBM, "OS/VS Virtual Storage Access Method: Options for Advanced Applications", IBM Corp., GC26-3819, 1973.

IBM74a IBM, "IBM System/370 Model 158 Functional Characteristics", IBM Corp., GA22-7011, 1974.

IBM74b IBM, "Reference Manual for IBM 3830 Storage Control Model I and IBM 3330 Disk Storage", IBM Corp., GA26-1592, 1974.

IBM74c IBM, "Information Management System/Virtual Storage: System/Application Design Guide", IBM Corp., SH20-9025, 1974.

IBM75 IBM, "Information Management System/Virtual Storage: General Information Manual", IBM Corp., GH20-1260, 1975.

IBM76 IBM, "OS/VS2 MVS Data Management Services Guide", IBM Corp., GC26-3875, 1976.

IBM77 IBM, "Planning for Enhanced VSAM under OS/VS", IBM Corp., GC26-3842, 1977.

KNUT73 Knuth, D. E., "The Art of Computer Programming, Sorting and Searching", Volume 3, Addison-Wesley, Don Mills, Ontario, 1973.

KNUT75 Knuth, D. E., "The Art of Computer Programming, Fundamental Algorithms", Volume 1, Second Edition, Addison-Wesley, Don Mills, Ontario, 1975.

LORI74 Lorie, R. A., "XRM - An Extended (N-ary) Relational Memory", IBM Cambridge Scientific Center, Cambridge, Massachusetts, G320-2096, 1974.

LUM70 Lum, V. Y., "Multi-Attribute Retrieval With Combined Indexes", CACM, Volume 13, Number 11, pp. 660-665, November 1970.

MART77 Martin, J., "Computer Data-Base Organization" Second Edition, Prentice-Hall, Toronto, Ontario, 1977.

MICH76 Michaels, A., "Secondary Indexes as Access Models for Relational Data Base Systems", Ph.D. Dissertation, Department of Computer Science, Northwestern University, Evanston, Illinois, 1976.

MRI74 MRI Systems, "SYSTEM 2000 Reference Manual", MRI Systems Corp., Austin, Texas, 1974.

MULL71 Mullin, J. K., "Retrieval - Update Speed Tradeoffs Using Combined Indices", CACM, Volume 14, Number 12, pp. 775-776, December 1971.

ROTH74 Rothnie, J. B. Jr. and Lozano, T., "Attribute Based File Organization in a Paged Memory Environment", CACM, Volume 17, Number 2, pp. 63-69, February 1974.

SENK72 Senko, M. E., et al., "Concepts of a Data Independent Accessing Model", ACM-SIGFIDET Workshop on Data Description, Access and Control, pp. 349-362, Denver, Colorado, November 1972.

SENK75 Senko, M. E., "Specification of Stored Data Structures and Desired Output Results in DIAM II with FORAL", Proceedings of International Conference on Very Large Data Bases", pp. 557-571, Farmingham, Massachusetts, ACM, New York, New York, September 1975.

SENK76 Senko, M. E. and Altman, E. B., "DIAM II and Levels of Abstraction, The Physical Device Level: A General Model for Access Methods", Proceedings of the Second International Conference on Very Large Data Bases", pp. 79-94, Brussels, Belgium, North-Holland Publishing Co., New York, New York, September 1976.

SEVE76a Severence, D. G. and Lohman, G. M., "Differential Files: Their Application to the Maintenance of Large Databases", ACM-TODS, Volume 1, Number 3, pp. 256-267, September 1976.

SEVE76b Severence, D. G. and Eisner, M. J., "Mathematical Techniques for Efficient Record Segmentation in Large Shared Databases", JACM, Volume 23, Number 4, pp. 619-635, October 1976.

SOFT74 Software AG, "ADABAS Introduction", Software AG of North America, Reston, Virginia, 1974.

STON74 Stonebraker, M., "The Choice of Partial Inversions and Combined Indices", International Journal of Computer and Information Sciences, Volume 3, Number 2, pp. 167-188, Plenum Press, New York, New York, 1974.

STON76 Stonebraker, M., et al., "The Design and Implementation of INGRES", ACM-TODS, Volume 1, Number 3, pp. 189-222, September 1976.

STRE77 Street, A. and Wallis, W., "Combinatorial Theory: An Introduction", Charles Babbage Research Centre, St. Pierre, Manitoba, 1977.

TSIC74 Tsichritzis, D., "On Implementation of Relations", Technical Report CSRG-35, Department of Computer Science, University of Toronto, Toronto, Ontario, May 1974.

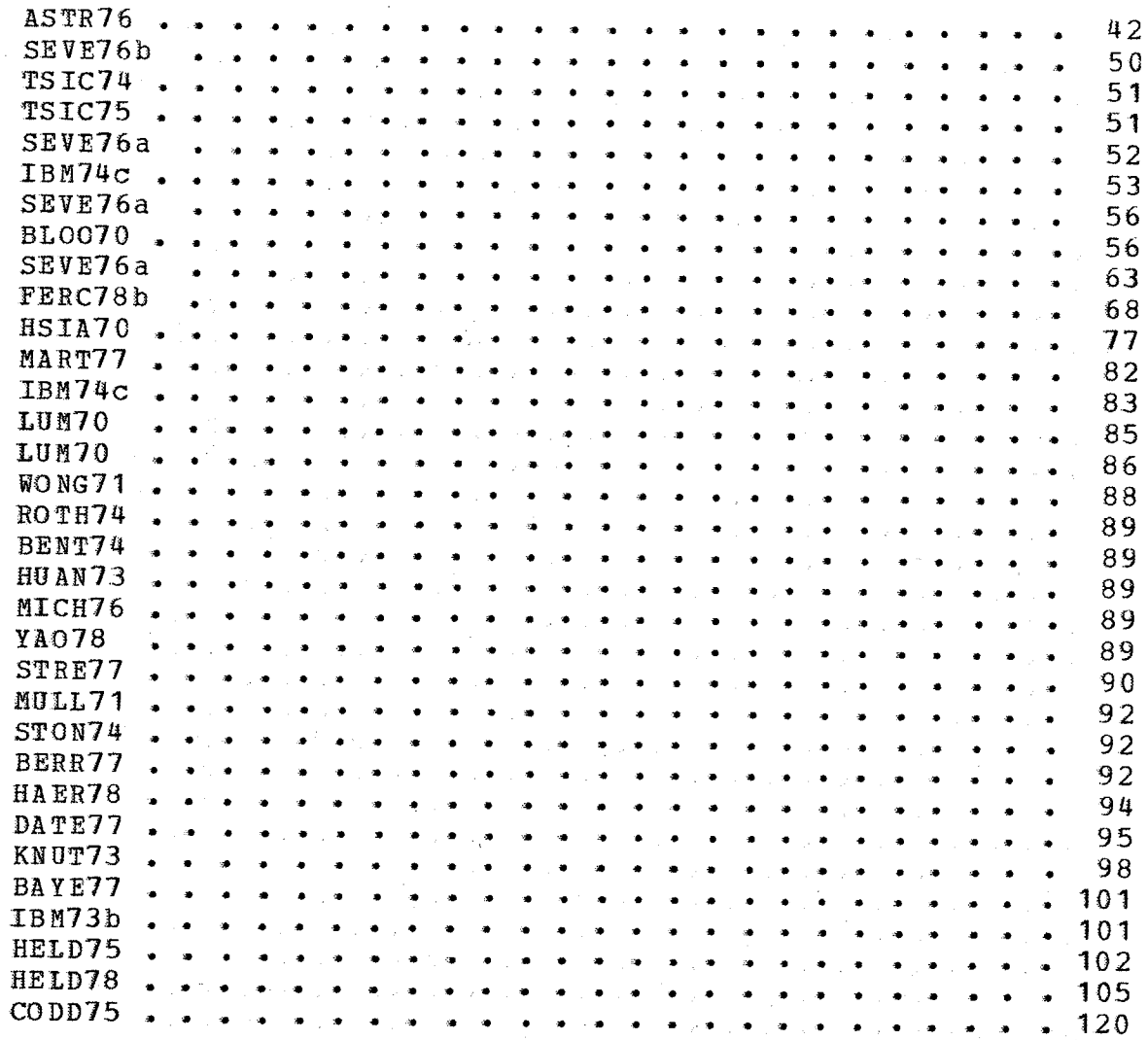
TSIC75 Tsichritzis, D., "LSL: A Link and Selector Language", Technical Report CSRG-61, Department of Computer Science, University of Toronto, Toronto, Ontario, November 1975.

WHIT74 Whitney, V. K., "Relational Data Management Implementation Techniques", Proceedings of ACM-SIGMOD Workshop on Data Description, Access and Control, pp. 321-348, Ann Arbor, Michigan, May 1974.

WONG71 Wong, E. and Chiang, T. C., "Canonical Structure in Attribute Based File Organization", CACM, Volume 14, Number 9, pp. 593-597, September 1971.

YAO78 Yao, A., "On Random 2-3 Trees", Acta Informatica, Volume 9, pp. 159-170, Springer-Verlag, New York, New York, 1978.

Table of References


ASTR76 42
SEVE76b 50
TSIC74 51
TSIC75 51
SEVE76a 52
IBM74c 53
SEVE76a 56
BLOO70 56
SEVE76a 63
FERC78b 68
HSIA70 77
MART77 82
IBM74c 83
LUM70 85
LUM70 86
WONG71 88
ROTH74 89
BENT74 89
HUAN73 89
MICH76 89
YAO78 89
STRE77 90
MULL71 92
STON74 92
BERR77 92
HAER78 94
DATE77 95
KNUT73 98
BAYE77 101
IBM73b 101
HELD75 102
HELD78 105
CODD75 120