AUTOMATING PHYSICAL REORGANIZATIONAL REQUIREMENTS AT THE ACCESS PATH LEVEL OF A RELATIONAL DATABASE MANAGEMENT SYSTEM

by

GRANT EDWIN WEDDELL

B.Sc., University of British Columbia, 1976

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

in

THE FACULTY OF GRADUATE STUDIES

(Department of Computer Science)

We accept this thesis as conforming
to the required standard.

THE UNIVERSITY OF BRITISH COLUMBIA

February, 1980

In presenting this thesis in partial fulfilment of the requirements for an advanced degree at the University of British Columbia, I agree that the Library shall make it freely available for reference and study. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by the Head of my Department or by his representatives. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Department of Computer Science

The University of British Columbia
2075 Wesbrook Place
Vancouver, Canada
V6T 1W5

Date March 3rd 1980

Abstract

Any design of an access path level of a database management system must make allowance for physical reorganization requirements. The facilities provided for such requirements at the access path level have so far been primitive in nature (almost always, in fact, requiring complicated human intervention). This thesis begins to explore the notion of increasing the degree of automation of such requirements at the access path level; to consider the practical basis for self-adapting or self-organizing data management systems. Consideration is first given to the motivation (justification) of such a notion. Then, based on a review of the relevent aspects of a number of existing data management systems, we present a complete design specification and outline for a proposed access path level. Regarding this system we consider in detail the automation of two major aspects of physical organization: the clustering of records on mass storage media and the selection of secondary indices. The results of our analysis of these problems provides a basis for the ultimate demonstration of feasibility of such automation.

Table of Contents

List of Tables

List of Figures

Acknowledgements

I would like to thank my supervisors Paul Gilmore and Al Fowler for their invaluable guidance and support. I would also like to thank Dave Kirkpatrick and again Paul Gilmore for their collaboration on the work resulting in Section C of Chapter III. Many thanks are also due Randy Goebel and the rest of my fellow graduates for many lively discussions regarding the thesis. Finally I would like to thank Paul and the Natural Science and Engineering Research Council for their financial support.

Chapter 1. Introduction

Reference will be made throughout this thesis to the "access path level" of a relational database[1] management system (RDMS). This level is defined by Tsichritzis and Lochovsky[2] as the level responsible for providing efficient access on relations. Their definition of access path level presumes the existence of a "file system" and, in turn, provides for the implementation of the "relational language" level (this latter level represents the lowest level user interface).

In more abstract terms the access path level represents the lowest level interface between the conceptual schema [Nijssen 76a] and physical schema of a RDMS. It provides the first level of information representation abstraction. Clearly, the design of this level is determined almost entirely by the requirements of the RDMS data management language (DML) and data definition language (or DDL, both of which comprise the user interface to the relational language level).

Tsichritzis and Lochovsky are very specific about the responsibilities of the access path level for an RDMS. Essentially two facilities are called for. The first involves

---

[1]The single word "database" instead of "data base" or "data-base" is used throughout the thesis.

[2]See pages 243-245 of [Tsichritzis and Lochovsky 77]

efficient access within a relation. This, in essence, provides for associative access to records. The second involves efficient access between relations and is realized, usually, with a linking convention. Such facilities might, for example, be used to implement, respectively, the selection and join operators of the relational algebra [Codd 70].

In Chapter II (where, among other things, a review will be made of some existing relational database systems) we will find the responsibilities of some existing access path levels somewhat more extensive than selection and link facilities. Additional duties have included:[1]

(a) management of locking protocols,

(b) provision for a variety of recovery methodology,

(c) automatic transaction deadlock detection and recovery and even

(d) complex query evaluation.

To a very limited extent provision at the access path level has also been made for:

(e) automatic physical clustering of records on secondary storage.

We can see, therefore, that the definition of access path level is by no means clearcut. Nevertheless, many existing relational database systems (e.g. System R [Astrahan et al.

---

[1]The reader should not think from this list that little remains to be handled by the "relational language" level. The Relational Data System (RDS) of System R [Astrahan et al. 76] provides for evolving views (or logical subschema), facilities for integrity assertions and an automatic transaction triggering facility.

76], ZETA [Brodie et al. 75], OMEGA [Schmid et al. 75] and INGRES [Stonebraker et al. 76]) conform closely to the hierarchy of levels defined by Tsichritzis and Lochovsky.

Our concern in this thesis is to extend the functionality of the access path level to the point of automating, as far as possible, the physical reorganization requirements of a RDMS. Such requirements (in order of priority) include:

(a) mass storage space management. It is a surprising fact that some of the most popular commercial database management systems (DMS) available at the time of this writing[1] require the explicit allocation and control of fixed size disk "extents" to their most abstract conceptual units.

(b) physical reorganization implied by addition and deletion of tuples and relations. In some commercial DMSs it is the user's responsibility to periodically schedule "file compaction" utilities for the purpose of reclaiming space occupied by "deleted" records.

(c) physical reorganization implied by changing patterns of usage of existing relations.

(d) physical reorganization implied by augmentation or alteration of the data model itself.

(e) physical reorganization implied by the detection of faults in the physical (as opposed to logical) organization of the tuples and relations.

I am sure the reader will agree that (a) and (b) above can and should be automated at some level of an RDMS. A need to automate (c), (d) and (e), however, is not as evident. Consider in more detail what is implied by (c). We can include

---

[1]For example, TOTAL - a product of Cincom Systems, Inc. [Datapro 79] in use at more than 2000 installations.

in this area decisions with regard to the archiving of information, all decisions regarding the clustering of records and even decisions with regard to the selection of such things as indices.[1] [2]

The reader may be unclear about the implications of (d). To clarify; such physical reorganizations might be implied by the addition of an attribute to a relation or the addition of a new relation. The automation of such physical reorganization would further contribute to data independence.[3]

There are some interesting thoughts with regard to item (e). Experience has shown that no matter what the system or hardware something will at some point in time become very wrong with the database.[4] It is quite safe to say that significant resource in the form of full time technical personnel is spent in detecting and correcting such problems in almost all commercial environments using a DMS

---

[1] Recall that the access path level must at least provide for automatic maintenance of such things.

[2] At the time of writing the author is not aware of any existing DMS (experimental or commercial, relational or otherwise) that fully automates (or even comes close) to physical reorganization requirements of item (c) alone.

[3] Not having to recompile (rewrite!) all application programs unaffected by the change in the logical model is certainly nice. Even more desirable is relief from the necessity of reformatting and reloading vast amounts of data due to a minor alteration in the logical model.

[4] I speak from experience when I say that the mysterious (ie. unsolved) disappearance of entire tracks of data from a disk can and will occur.

(successfully). Considering the growing complexity of the data structures involved at the access path level of DMSs it is clear that automating recovery of physical data integrity is very desirable.

There is, however, an overriding motivation for such degrees of automation. This concerns the growing tendency of operating systems to assume the responsibilities of database management systems [Rodriguez and Echhouse 77]. Indeed many feel that file systems are, in fact, nothing more than a primitive DMS. We regard the desire of a "RDMS at the file system level" ("phase four" in [Berryman and Fowler 79]) as the prime motivation (justification) for the increased automation of physical reorganization requirements (the first section of Chapter II discusses this more fully).

Now that we understand the notions "access path level" and "physical reorganization requirements" we can categorize the degree of automation of a given access path level:

L1: no reorganizational effort is ever necessary.

L2: reorganization never is absolutely necessary and can nevertheless always be accomplished concurrently with retrieval and alteration.

L3: reorganization is never absolutely necessary but, once initiated, requires exclusive access to any affected structures and data (i.e. no concurrency).

L4: reorganization can become absolutely necessary but is entirely predictable and can always be accomplished concurrently with retrieval and alteration.

L5: reorganization can become absolutely necessary and is entirely predictable, but once initiated, requires exclusive access to any affected

structures and data (almost all commercial DMSs are no higher than this level).

L6: reorganization can become absolutely necessary and is not predictable and will not permit any degree of concurrency of use of any affected structures and data (a garbage collector for a lisp interpreter is in this category).

It is not clear which of L3 or L4 is more desirable. With respect to this 'hierarchy' it is clear that an implementation should attain a level as close to L1 as possible.[1] Indeed, since level L6 above is clearly unacceptable, an implementation must achieve a level at least as good as L5.

The design specification for the access path level presented in this thesis accomplishes physical reorganizational requirements concerning a type of record clustering (i.e. "horizontal" clustering[2]) and index selection. These are realized at a degree of automation corresponding to that of L2 above. The system, itself, is called SORAAM (for Self-Organizing Relational Associative Access Method).

Chapter II is concerned, initially, with a more detailed discussion of the motivations and justifications for such automation. A review follows concerning the relevent aspects

---

[1]There is good reason to believe that achieving level L1 itself is impossible. As shall be shown, the "optimal" arrangement of records within pages makes sense only with regard to a specific characterization of table or relation use (or query history). In a real world environment such a characterization itself may very from one day to the next.

[2]see "Detailed Problem Definition" in Chapter II.

of a number of existing RDMSs. Some detailed problem definitions for our access path level are then presented followed by an outline of the simplifying assumptions.

Chapter III is concerned with the design of SORAAM. An initial section outlines the design specifications and presents an overall design of the system. The next two sections are concerned with the primary indexing and the last two with the secondary indexing and search strategy. Such considerations of SORAAM relate to the two major aspects of physical organization implied by patterns of usage of base relations: tuple clustering or blocking and secondary index selection.

The first section of Chapter IV presents an outline of the "next step" towards a demonstration of feasibility: the acquisition of some specific empirical evidence and the testing environment implied. The thesis concludes with an overall assessment of SORAAM. In particular, advantages realized over other systems with regard to self-organization and automatic index selection are given. Unresolved issues and suggestions for future research comprise the last part of the assessment.

Chapter II: Preliminaries

A. Justification for Automation

Ample evidence exists that suggests a tendency of operating systems to assume more direct responsibility of such things as DMSs. Consider, for example, the recent introduction of System/38 from IBM [Henry 78]. In this system, support for a DMS was thought important enough to warrant the microcoding of a significant percentage of the required software [Watson and Aberle 78]. Nevertheless, one may ask what we can expect of a DMS that a more traditional file system (possibly including access method utilities) does not provide.

The obvious advantage of a DMS is its greatly enhanced capacity for "direct"[1] information representation. With respect to this an interesting analogy can be drawn between DMSs and file systems on one side and high level and low level programming languages on the other. One may even claim that, where operating system facilities for procedural specification (in the form of command languages/interpreters) have evolved enormously, corresponding facilities for information representation and manipulation have remained at a primitive level.

---

[1]The degree of "direct"ness of various DMS's "conceptual schema" [Nijssen 76a] is often used, in fact, as the most significant measure of adequacy (for example, evaluations concerning "relational" verses "network" abstractions in [McGee 76]).

In a DMS, facilities for integrity and security constraint specification are far more elaborate. Traditional file systems have provided for a degree of security but only at a gross file level. Direct support for the maintenance of physical integrity alone in most file systems is minimal.

In enforcing a conceptual schema at the operating system level a greater degree of device independence of "all above the OS" is achieved. Users are far less capable of taking advantage of device dependent knowledge in implementing applications (this, of course, being the concern of the DMS implementors). This results in an environment far less susceptible to such things as hardware upgrades and even contributes significantly to the overall portability of all the applications.[1]

Recovery facilities provided by a DMS are almost always more extensive than those of file systems. The latter, for example, have, in almost all cases, no support for recovery at the transaction level. A DMS provides greater degrees of recovery at larger levels of concurrency to finer grains of data.

There is a more subtle but extremely important advantage to the enforcement of a conceptual schema at the operating system level (i.e. a DMS instead of a file system). This

---

[1] Should operating system primitives take advantage of a DMS to perform some of their duties then such a DMS would contribute directly to the portability of the operating system itself.

concerns the resulting integration of all information representation used by all applications. Such a result would benefit virtually all levels of operating system support. Consider, for example, that the normal proliferation of special purpose recovery utilities would essentially be checked.[1] A good case exists that contends such an integration to be prerequisite to the attainment of higher levels of computing environments.[2]

Naturally, a DMS subsuming the role of file system must be able to support a variety of "traditional" operating system facilities. These include:

(a) spooling systems,

(b) message passing systems,

(c) device and network interfaces and even

(d) compiler source code management.

A remarkable design characteristic of the System/38 from IBM [IBM 78] is that the DMS of that machine provides the basic support for all of the above. In [McDonell 77] many arguments are presented in favour of a "homogeneous secondary storage input-output interface" (based upon the CODASYL DDL/DML, [CODASYL 71], [CODASYL 73]). In an appendix to his thesis we are presented with an entire implementation design for a

---

[1] This is true since all that would normally be required by most applications would already be provided by the DMS.

[2] A complete demonstration of this, however, is considered to be outside the scope of the thesis.

spooling subsystem based on this interface. In his thesis, [Cargill 79] goes to great lengths in directly applying the information modeling capability of the UNIX file system for the purposes of compiler source code maintenance.[1]

If we accept that a DMS instead of a file system is both desirable and non-restrictive then obvious justification exists for achieving the highest degree of automation possible for physical reorganization requirements. Clearly, one cannot allow a situation in which an entire computing system is frequently unavailable because of time-consuming reformatting or rebuilding phases. We see, therefore, the design of our access path level appropriate in this more global environment.

One last point we shall deal with in this section concerns the idea of arriving at our "dream" by simply implementing our DMS on top of a currently existing file system. Experience has shown that such an approach eventually involves either significantly altering the operating system itself or adopting substantial design compromises. The System R people [Astrahan et al. 76] found it necessary, for example, to make several extensions to VM/370 (IBM's large-scale virtual machine facility). Others concur [Berryman and Fowler 79], [Gray 78] that this is to be expected. On the other hand, consider a decision made by the designers of INGRES:

---

[1] One cannot help feel, when reading the thesis, that his life would have been made much easier could he have taken advantage of a more powerful information modeling capability.

> "In keeping with the design decision of not modifying UNIX these considerations[1] were incorporated in the design decision not to support clustering."[2]

We shall consider ourselves, therefore, to be unconstrained in the design of our access path level by lower levels of software. Likewise, we shall be forever wary of our commitment to the relational model of a conceptual schema.

B. Previous Systems

A number of RDMSs have been developed at various research centers. This section first presents a general overview of some of these. Emphasis in this overview is placed on their respective access path levels (as we shall see the distinction of an access path level is surprisingly well-defined in most of these systems). The physical reorganizational aspects of the access path level is of particular concern.

There has been some work concerning self-organization and the problem of record (tuple) clustering. The second part of this section reviews this work in relation to our notion of physical reorganizational requirements.

---

[1] "these considerations" involve the facts that files are realized in UNIX by small (512 byte) possibly randomly located physical blocks (certainly an environment inhibiting consideration of clustering).

[2] This passage is taken from page 202 of [Stonebraker et al. 76].

ZETA [Brodie et al. 75] is a RDMS (relational database management system) being developed at the University of Toronto to run on IBM 370 type machines. The system is composed of three principal levels termed MINIZ, the EXECUTOR and the "language facilities".

The first of these levels, MINIZ, performs the duties required by our notion of an access path level. This component implements relations directly as files of data. That is, each relation in MINIZ is represented as a single file.[1] Besides this initial level of abstraction MINIZ can accomplish single relation retrievals (requiring no more than one pass) and tuple-at-a-time modifications. Responsibilities also include a recovery facility and maintenance of four system relations. These system relations store all information describing all relations, domains, users and the physical extent and blocking of relations (the latter is used only by the file system).

MINIZ is managed by the EXECUTOR. This second level forms part of the relational language level of ZETA and is responsible for "snapshot"[2] relations and multiple relation queries.

A "host programming language system" (HLS) and "self-contained language system" (SLS) comprise the language

---

[1]Almost all RDMSs we consider take this approach.

[2]This should not be confused with derived "views" or subschema. A snapshot relation is independent of the relations from which it is created.

facilities (third level) of ZETA. The latter is interesting in that it is actually a "query language generator system" (QLS) which is used to generate a tailor-made SLS.

The base relations of ZETA must each be assigned a fixed sized disk extent (which, of course, implies a maximum number of tuples allowed in any such base relation).[1] Furthermore, all tuples are added at the end of a relation (tuple deletion results in a simple marking). To build an appropriately clustered relation, therefore, a user must first find and allocate a physical extent (he must anticipate relation size) then appropriately order his tuples prior to insertion. To reclaim storage used by deleted tuples the user must essentially rebuild the relation.

Reorganization appears to be predictable in ZETA. However, virtually no automation of physical reorganization requirements is provided. Also, the search facilities used by MINIZ are not as sophisticated as other systems (most new queries involve sequencial scanning of base relations).

Another RDMS developed at the University of Toronto is the OMEGA system [Schmid et al. 76]. OMEGA was developed to run under the control of the UNIX operating system on a PDP 11/45 minicomputer. The levels which comprise OMEGA (excluding the one represented by the UNIX file system) are entirely analogous to the ZETA system.

---

[1] In fact, an OS JCL DD card must be provided for each base relation for any job using ZETA.

Operations implemented by MINI-OMEGA and by the "data structures system" comprise the first or access path level. MINI-OMEGA provides the initial relational abstraction communicating directly with the UNIX file system. The data structure system maintains inverted files and "basic elements". The latter consists of two varieties corresponding essentially to internal temporary results and snapshot relations.

The second level of OMEGA (termed the "access structure level" by the implementors), like ZETA, forms part of what we understand as the relational language level. This access structure level is responsible for the optimization and interpretation of a query language, the consistency of existing access structures and their creation and destruction. The query language is called the "link and selector language" (LSL) and must be the result produced by all relational query languages comprising the user interfaces at the third level.[1]

As in ZETA the level of automation of physical reorganizational requirements is relatively low. There are, however, a number of advances over ZETA. The automatic maintenance of secondary indices is one example (although the creation and deletion of indices is still the responsibility

---

[1]An expression in LSL, as seen at the second level, is tree structured. The current OMEGA implementation provides for a relational query language that maps directly to this tree structure.

of a human[1]. Also, the initial tuples of a relation are stored in a primary area sorted by key value. If access to tuples is based primarily on the key domains of the relation this results in a more desirable (but not necessarily optimal) clustering of tuples within disk pages. Tuples that are added following the initial setup are located in an overflow area which in turn implies a periodic physical reorganization requirement.

Another form of tuple clustering is also realized by OMEGA. This involves an ability to store a logical record (tuple) split-up into several physical records in separate physical files (i.e. the i-th physical records of a set of physical files comprise the i-th logical record). The advantage of this is that many more partial-tuples can now be stored in a single block of a physical file. Of course, such a "vertical" partitioning of tuples depends entirely on the nature of use of the relation. Again, the human is responsible for determining such a partitioning. Later on in this section we review a system (SODMS [Kollias et al. 77], [Stocker and Dearnley 74]) which automates this responsibility.

INGRES [Stonebraker et al. 76] is another RDMS developed to run under the control of the UNIX operating system. The structuring of the various components of INGRES, however, is more involved that either ZETA or OMEGA. Our analogy of these

---

[1]In fact, none of the systems we review provide for this degree of automation.

components to the "access path level" and "relational language" abstractions of [Tsichritzis and Lochovsky 77] is somewhat more strained.

Due mainly to the limited logical address space of PDP 11s (64k bytes) INGRES is structured as a four level hierarchy of four communicating processes. The first of these (top level) may be an interactive terminal monitor or an executing user process. The former facilitates direct use of INGRES by users and the latter access to INGRES by user programs.

The second level process (process 2) has responsibility for parsing queries, concurrency and security control and support for integrity control and derived views.[1] Security and integrity control and derived views are accomplished by a query modification procedure. Concurrency is realized by use of a locking protocol restrictive enough to not require transaction deadlock detection and recovery.

Level three of the process hierarchy is responsible for query evaluation. This is accomplished by a query decomposition procedure and a one-variable query processor (OVQP). Both are concerned to a large extent with optimization considerations. All other INGRES commands are interpreted by a set of overlaying utilities at process level four.

A set of functions used in common by processes two, three

---

[1]Derived views differ from snapshot relations in that they always reflect the current state of the base relations from which they are derived.

and four is called the "Access Methods Interface" (AMI). This set of functions communicates directly with the UNIX file system to implement the initial relational abstraction. It is the AMI together with process levels three (or at least OVQP) and four that corresponds to our notion of an access path level. The relational language level is provided by processes one and two as a query language called QUEL and an embedded[1] query language called EQUEL.

The AMI supports five different physical organizations of base relations. These correspond to a "heap" storage scheme and "hashed", "compressed hash", "ISAM" and "compressed ISAM" indexed storage schemes. The first of these is considered appropriate for very small relations, transitional storage and temporary internal relations. If access to a relation is primarily via a partial match on its key attributes then the hashed scheme is appropriate. In this case the physical page location of a tuple is determined by a hash function applied to an ordered list of values of its key attributes. The ISAM scheme is to be prefered in situations where access to the relation has the form of a partial range query on its key attributes (i.e. a search condition specifying a range of values for each key attribute of the relation). This is because the physical page location of a tuple is determined by a key ordering. As in OMEGA, this results is a clustering (on

---

[1] Embedded, that is, in a general purpose programming language called "C".

the key attribute values) of tuples within disk pages.

INGRES, also like OMEGA, maintains overflow pages for its base relations (an overflow page is used whenever storage is exhausted in a primary page[1]). This again requires periodic physical reorganization. In INGRES, secondary indices are supported and are maintained in the same way as any of the base relations. Overall INGRES is analogous to OMEGA with respect to its degree of physical reorganization requirements

By far the most ambitious RDMS development to date is the System R project [Astrahan et al.76] undertaken by IBM at their San Jose research laboratory. System R represents an attempt to provide a complete relational database management system. In particular, support is provided for a large degree of concurrency and system recovery. A modified host operating system (VM/370) directly supports multithreading (concurrency) at the level of virtual machines which in turn allows a productive use of a multiprocessor environment. Extensive support for system recovery at the logical transaction level is provided for both soft and hard system failure. A system-wide monitor program has the responsibility for such recovery and for the detection and recovery from transaction deadlock (the level of concurrency in System R is such that deadlock can arise).

The System R part of each virtual machine is comprised of

---

[1]The number of primary pages of a base relation is fixed at the time of initial setup.

two major components. The Relational Storage System (RSS) together with its Relational Storage Interface (RSI) corresponds in function to our notion of access path level. The relational language level is realized by the Relational Data System (RDS) of System R together with its Relational Data Interface (RDI; lowest level user interface) the main component of which is the SEQUEL DML/DDL. All application programs communicate with System R through the RDI. Figure II-1 below illustrates the basic architecture of System R.

Architecture of System R

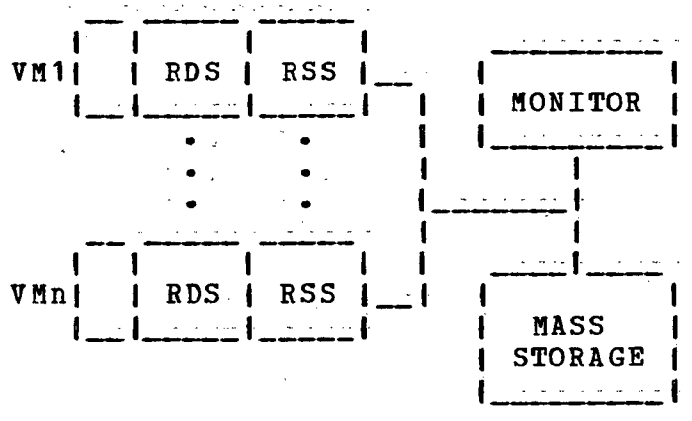


Figure II-1

A major component of the RDS is the optimizer. It is the responsibility of this component to find the cheapest possible means of executing SEQUEL statements given available links and indices defined on the base relations. The optimizer chooses from a variety of methods to evaluate both restriction on relations (equivalent to a one variable query in QUEL under

INGRES) and inter/intra-relation join operations (multiple variable queries in QUEL). This is accomplished primarily through the use of a variety of statistics maintained for each base relation.[1]

The RDS also has the responsibility for providing for user defined evolving views, integrity assertions, authorization and transaction "triggers". The latter facility is analogous to the antecedent theorem of PLANNER [Hewitt 72]. In addition the RDS must provide "advice" to the RSS as to where new tuples should be located. This is accomplished by passing a tentative disk address for the new tuples. All the system catalogue relations are also maintained by the RDS. They provide the information about all other base relations, views, images, links, integrity assertions and triggers.

Images and links are special data structures that permit efficient associative access to tuples in System R. Images are secondary indices in the form of B-trees [Bayer and McCreight 72] and are defined on an explicit ordered subset of attributes of single base relations. Links are a multiring data structure [Wiederhold 77] permitting a direct efficient means of representing one to many relationships among attributes within and between base relations. Links defined on a single attribute in a single relation imply a partial

---

[1] As far as I know, these statistics are not dynamically maintained but rather are specified by a user at initial relation definition/setup.

ordering of the tuples on that attribute. Binary links (i.e. links between attributes in separate relations) are analogous in concept to a variety of owner coupled set of the DBTG model (see [CODASYL 71]). Both images and links are maintained and used by the RSS component of System R.

In addition to base relations[1], users of System R also have the responsibility for specifying the RSS access paths (i.e. images and links). Research is currently underway, however, to automate this function. It should be stressed at this point that the DML component (including the query facility) of the RDI of System R in no way presumes or permits any knowledge of the existence of these access paths. Such a property of the DML is precisely what distinguishes System R as a RDMS (verses a hierarchic or network DMS).

The access path level of System R (the RSS) is the most ambitious and sophisticated we shall review. The RSS provides the support for both physical (segment level) and logical (transaction level) recovery from transaction deadlock and system failure. In addition the RSS supports all logical and physical locking requirements and the dynamic creation, maintenance and deletion of base relations, images and links. New fields may be added on the right of any existing base relation at any time without incurring the overhead of database reload or even the immediate update of the tuples

---

[1]Relations, in System R, are actually refered to as "tables".

within the base relation.

The searching support of the RSS is analogous in capability to the OVQP of the INGRES system. In fact, OVQP and RSS are of particular interest to us in just this respect since we shall be adopting this level of search capability as part of the design specification of our own access path level. A legal search predicate to the RSS is a boolean expression of primitives in disjunctive normal form (see [Selinger et al. 79]). Each primitive has the form of a comparison operator on an attribute (or field number) and a value. All primitives in a given search predicate must of course apply to the same relation. A more formal illustration of this format is given in Table II-1 below in terms of a BNF syntax chart. The symbols "OR" and "AND" in the chart denote the logical disjunction and conjunction operators respectively.

All information in System R (including tuples, indices, etc) are stored in a set of segments (each of which has its own logical address space). Unlike OMEGA all tuples of a given base relation are constrained to occur in a single segment. Unlike all other systems, however, a single segment might contain more than one relation. Such a "vertical" aggregation[1] capability when combined with the link access path can be used to greatly enhance the performance of relational join operations.

---

[1]Recall in our review of OMEGA that "vertical" partitioning may also be advantagous.

Recall that the RDS provides tentative disk addresses for new tuples to the RSS. This facility exists for the purpose of

BNF Syntax of Legal Search Predicates for the RSS of System R

```
<legal predicate> ::= <conjunct> |
                      (<legal predicate> OR <conjunct>)

<conjunct>        ::= <primitive> |
                      (<conjunct> AND <primitive>)

<primitive>       ::= (<field number> <operator> <value>)

<operator>        ::= = | ¬= | < | > | <= | >=
```

Table II-1

clustering tuples of base relations in disk pages. This clustering capability, however, is limited to at most one image or link per base relation (clustering on a link makes sense in System R because of the allowance for vertical aggregation). In addition, all such clustering images or links must be explicitly declared through the RDI by the System R user. Nevertheless, the clustering capability is a fundamental and important aspect of System R.

The physical reorganization requirements of System R are the least of all RDMSs covered in this review. The access path level (RSS):

> "...supports dynamic addition and deletion of relations, indexes, and links, with full space reclamation, and the addition of new fields to existing relations - all without special

utilities or database reorganization."[1]

As we have seen, System R also provides for a clustering capability. There is, however, much room for improvement. Users of System R must still allocate disk segments and suitably distribute base relations among them and they must still specify which access paths are to exist. Furthermore, they must understand well enough the use of the base relations to determine an optimal set of clustering images and links.

An interesting approach to a batch oriented DMS is the SODMS (for Self-Organizing Data Management System) by [Stocker and Dearnley 74] (see also [Stocker and Dearnley 72] and [Kollias et al. 77]). As the name implies, the design emphasis in SODMS is to automate the selection of data structures and access methods. Such selections are based on an analysis of a history of system usage (a query history).

The operational environment of SODMS assumes the following:

(a) user to user interaction is minimal,

(b) the database functions primarily in responding to queries and

(c) query and update response or turnaround is uncritical (thus permitting use of batching techniques).

The breadth of application is, therefore, somewhat more limited than a general purpose RDMS such as System R. Nevertheless, such restrictions have allowed the experimenters

---

[1]This passage is taken from page 129 of [Astrahan et al. 76].

to concentrate on automating control over key physical organization details.

The data model presented to users by SODMS is that of a set of tables or relations. As yet, no subschema or protection facilities are provided on these (i.e. all users see all tables). In anticipation of confusion over terminology the designers have labeled these tables as "folios". A folio in SODMS may be realized by a variety of different files representing records, sub-records, amendments, directories, etc. All decisions with regard to the design and creation of these files are made by the system itself and are based on an analysis of past use.

Consider, for example, a folio (a relation) comprised of the five attributes A1 to A5. If the last evaluation of past use found that the following typified use of the folio:

(a) given values for A1 and A2 get A4,

(b) given a value for A1 get A5 and

(c) get (order not important) all attributes of all records.

then we could expect the folio as depicted in Table II-2 (a). (Bracketed attributes in the table denote an ordering on a file thus permitting associative access to the sub-records based on those attributes.[1]) In SODMS evaluations are periodic. If the next evaluation of this folio were to

---

[1]The present implementation of the system, however, allows orderings in files to be based on at most one attribute. File 2, therefore, cannot currently be realized.

determine that queries of the form of (a) above no longer occurred and that different queries of the form:

(d) given a value for A2 get A5.

took their place then we would see the folio altered as in Table II-2 (b). Notice that file 2 has not simply been destroyed but instead has been archived to tape. In general this is true of all information in SODMS. It is the opinion of the designers, in fact, that no information, once added in a DMS, should ever be destroyed.[1] This is not to say, however, that amendments continue to be applied to file 2.

In [Kollias et al. 77] we see the application of linear programming techniques to batched query evaluation and a type of file archiving. An analogy is made between developing an execution strategy for small batches of queries and the "set covering problem"; also for the allocation of files to on-line and off-line disc packs and the "knapsack problem". Consideration is also given to archiving parts of files to offline disc packs.

With respect to System R, SODMS has automated all allocation of disk storage and the selection of clustering images (linking data structures as access methods are not supported in SODMS). This, in itself, is an important accomplishment of the system. SODMS is, however, much more

---

[1]There are actually many good reasons for this opinion. Consider, in the case of accounting applications for example, the usually unpredictable arrival patterns of government auditors, law suits, etc.

limited in its range of applications. Excluded from consideration, for example, are highly dynamic on-line environments requiring immediate update and fast query

Example Effect of Change in Use of a Folio in SODMS

File Number	Attribute(s)	Location
1	A1,A2,A3,A4,A5	disk
2	(A1,A2),A4	disk
3	(A1),A5	disk

(a) before reorganization

File Number	Attribute(s)	Location
1	A1,A2,A3,A4,A5	disk
2	(A1,A2),A4	tape
3	(A1),A5	disk
4	(A2),A5	disk

(b) after reorganization

Table II-2

response. Secondary indices (on files) are also not supported. Furthermore, experimental results in [Stocker and Dearnley 74] suggest that System R's capabability of allowing the dynamic augmentation to a file of additional columns or attributes is highly desirable.

Unlike System R, SODMS batches all amendments to its folios. These amendments are then applied during periodic file (as opposed to folio) reorganization. While some indexing methods such as B-trees are unaffected in their performance with on-going alteration others such as ISAM techniques

degrade when such alteration is continually applied. [Yao et al. 76] have developed a dynamic database reorganization algorithm that may be used for the latter case. Input to their algorithm requires the following parameters for each physical organization occuring in the database:

(a) an initial search cost,

(b) an initial reorganization cost,

(c) a rate of deterioration of the search cost without reorganization,

(d) a rate of deterioration of the search cost assuming continuous reorganization and

(e) a rate of increase of reorganization cost.

They further consider the more realistic situation entailing nonlinear search costs and describe a system (called a File Design Analyser) that outputs search cost estimates for "real world" physical organizations (such as sequential and ISAM files). Their system accepts three basic varieties of descriptive parameters:

(a) logical record organization (such as number of attributes, record length, number of keywords, etc),

(b) utilization information (such as number of insertions, deletions and modifications, number of queries, etc) and

(c) physical storage parameters (such as block size, transfer rate, etc).

The result of their efforts is an approach capable of automating decisions regarding the scheduling of physical reorganization implied by record amendment (in our case by the addition and deletion of tuples from an RDMS). This is clearly

relevent to any access path level permitting the existence of non-dynamic data structures.

C. Detailed Problem Definition

Two aspects of physical reorganization implied by changing patterns of usage of existing relations are considered in detail in this thesis (see Introduction). Informally, what are considered is a variety of tuple (or record) clustering which can be characterized as "horizontal clustering" and the problem of dynamic secondary index selection. In terms of System R this concerns the automation of the selection of images (both clustering and non-clustering). This section presents a detailed introduction and definition of these problems.

A "base relation" in a RDMS corresponds to what might intuitively be called a stored relation. The tuples comprising a base relation physically exist on some form of mass storage media. In constrast a "derived" relation is defined in terms of base relations and requires some form of query evaluation in order to "materialize" its tuples.

Throughout previous sections reference has been made to the notion of clustering of base relations. In fact a variety of types of clustering have been mentioned. In the RDMS OMEGA [Schmid et al. 76] and in SODMS [Stocker and Dearnley 74] a base relation may have its attributes or columns partitioned

into (not necessarily distinct) subsets of attributes where each subset is assigned its own file. Such "vertical partitioning" requires each requested tuple of the base relation rebuilt from its component sub-tuples from each subset. In System R a "vertical aggregation" capability permits grouping sets of attributes from more than one base relation in the same physical segment. In addition, grouping tuples according to their probability of access is the subject of additional work on SODMS in [Kollias et al. 76]. Clearly then, clustering support at the access path level of a RDMS is of fundamental importance (and up to now we have relied on reader intuition in understanding this issue). A more formal outline is now presented.

One essential characteristic of all mass storage media is that they are block oriented. Communication with such media is in terms of usually large (relative to tuple size) physical blocks. Retrieval of any particular tuple or partial tuple always implies retrieval of a much larger block. When one considers that block retrieval time from a mass storage media is enormously slow in comparison to the time to access main memory (on the order of five to six orders of magnitude in the case of moving head disks) it is easy to see that the organization of tuples within blocks (or the clustering of tuples) has great impact on the performance of the RDMS. We define the problem of tuple clustering, therefore, as the following:

D1. clustering - Clustering concerns the placement of

> tuples of a base relation on mass storage media in such a way as to minimize block reads/writes relative to a "characterization of usage" of the base relation.

From the above definition it is clear that the clustering organization of tuples is relative at the least to each application of the RDMS. The problem is in fact fundamentally related to how each application is used (eg. the variety of queries which arise). It is important, therefore, that a notion of "characterization of usage" be more fully understood.

The number and variety of queries on a base relation is clearly an important component of such a characterization of use that is possible to be automatically maintained. Some possibilities of how such query use may be characterized are:

(a) an enumeration of all permitted queries,

(b) a sample of (a) above and

(c) query statistics.

Both (a) and (b) above seem at the least to be somewhat unwieldy. With respect to controlling our own clustering we will, in fact, be using query statistics. Furthermore, requiring a priori knowledge of the future use of a base relation is not possible to automatically predict. A characterization of query history is more appropriate for our purposes.

A characterization of use of a base relation also determines what variety of indexing should exist on the base relation. An index, in our case, is a performance

consideration. Furthermore, all such indices at the access path level may be characterized as either a "primary index" or a "secondary index". This distinction is important enough to warrant a more complete explanation of the purpose and role of these two basic varieties of indexing.

In a certain sense their "logical" purpose has been equivalent. In a procedural "tuple-at-a-time" data management language (DML) operating on a network model, for example, no distinction is made between a primary or secondary index either on initiation or during execution of a navigation through the network. Knowledge of a distinction arises in terms of performance only. Concerning the relational model scanning a base relation in System R via a clustering image is much more efficient than via an unclustered image.

The difference between a primary and secondary index, therefore, is that the former has the additional property of influencing the physical layout (more usually location) of a data entity. In terms of a RDMS the primary index of a base relation defines the physical location of its tuples. Since any other purpose of a primary index may be subsumed by further secondary indexing this role of a primary index is all important. With respect to the proposed design of an access path level of a RDMS the following definitions therefore

apply:[1]

D2. secondary index - As an access path to data a secondary index is a data structure permitting a variety of more efficient search capability on a base relation but with no relationship or bearing whatever on the clustering nature of the tuples within that base relation.

D3. primary index - A primary index is also an access path to data. It has the additional purpose, however, of facilitating the desired clustering of tuples within a base relation.

Consider what is implied by the above definition of a primary index. By virtue of facilitating clustering, the primary index provides for the mapping between the tuples of a base relation and their location on mass storage media. Clustering, in turn, is determined by a characterization of use (in our case a query history statistic). In an indirect sense, then, a primary index maps tuples of a base relation to physical store in a way dependent on how the base relation is used.

There are a number of ways in which the information in a base relation can be clustered. The vertical partitioning (OMEGA, SODMS) and vertical aggregation (System R) is one such approach. In vertical partitioning a base relation is divided vertically into a set of smaller tables each storing a subset (possibly non-disjoint) of the attribute values for each tuple (see Figure II-2 (a)). This approach presupposes that each such subset of the attribute values for the set of tuples is

---

[1]Keep in mind, however, that the DML/DDL presented at the relational language level of a RDMS (or at least the DML component) has no notion of an index (whether primary or secondary).

all that is required for a significant fraction of all queries (a "characterization of use"). Clearly smaller subsets of attribute values imply smaller partial-tuples. This in turn favourably impacts mass storage I/O because of the larger blocking factor of the partial-tuples within the disk pages comprising the tables.

The above suggests the following definition:

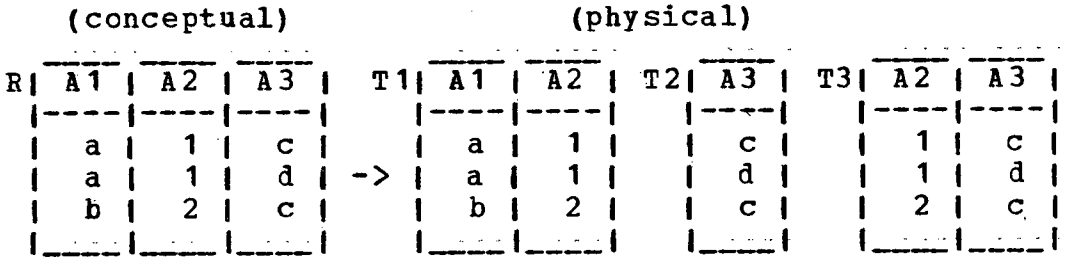
D4. vertical clustering - Vertical clustering of a set of base relations (conceptual) is a partitioning of their combined set of columns or attribute values into a set of tables (physical) in such a way as to reduce disk reads/writes relative to a characterization of use.

In [Schkolnich 76] the problem of vertical clustering is applied to a set of base relations logically related in terms of a hierarchy. An efficient algorithm is presented which generates an optimal vertical aggregation of the relations into a set of tables called linear address spaces or LASs in the paper. The algorithm is driven by a formulation of characterization of use of the base relations. The author mentions that the model for the characterization of use can easily be expanded to account for other forms of use of the base relations.

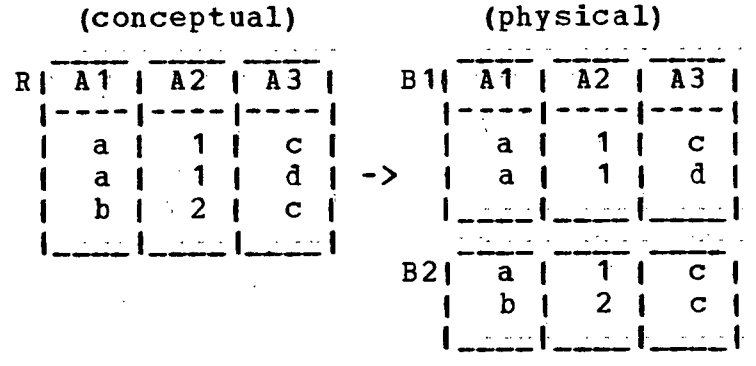
As we have seen, System R supports an ability to store (or aggregate) more than one base relation in the same file (thus resulting in locating tuples from different base relations in the same block). Such aggregation is controlled by the clustering option of links declared by the system's users. With the possible exception of applicability of work

(by Schkolnich) introduced above to an RDMS no consideration in the literature, as far as the author is aware, has been given to automating the choice of such clustering images. The problem remains open. No further consideration by this thesis is given, in fact, to the subject of vertical clustering.

Horizontal vs Vertical Clustering



(a) Vertical Partitioning Into Tables T1, T2 & T3.



(b) Horizontal Clustering Into Blocks B1 & B2.

Figure II-2

Horizontal clustering is another way in which information in a base relation can be clustered. Concern here is with possible groupings of the tuples (or partial-tuples if vertical partitioning clustering has been applied) themselves

within the physical pages comprising the table(s) (see Figure II-2 (b)). In general the sets of elements defined by these groupings need not be disjoint. In Figure II-2 (b), for example, two copies of one of the tuples is actually stored. It is this form of clustering that will be dealt with in detail by this thesis. Allowing replication of a tuple, however, will not be considered any further.

This second dimension of clustering has the following definition:

D5. horizontal clustering - Horizontal clustering of a base relation is a partitioning of its tuples into fixed sized blocks or disk pages in such a way as to reduce disk reads/writes relative to a characterization of use.

Figure II-3 and Table II-3 illustrate how a characterization of query use impacts the performance of sample horizontal clusterings of a relation in block reads. As expected, a random allocation of the tuples to blocks yields the poorest performance. More interesting is the observation that sorting does not yield the best. In fact, our approach to horizontal clustering in this thesis would result in the third horizontal partitioning of the tuples (i.e. cluster3) which, as shall be seen, is optimal. This leads us to our last definition concerning clustering:

D6. optimal horizontal clustering - The optimal horizontal clustering of a base relation is the horizontal clustering resulting in the minimal retrieval cost relative to a characterization of query history where retrieval cost is defined in terms of disk reads/writes.

Some interesting work regarding automation of horizontal

Example Horizontal Clustering

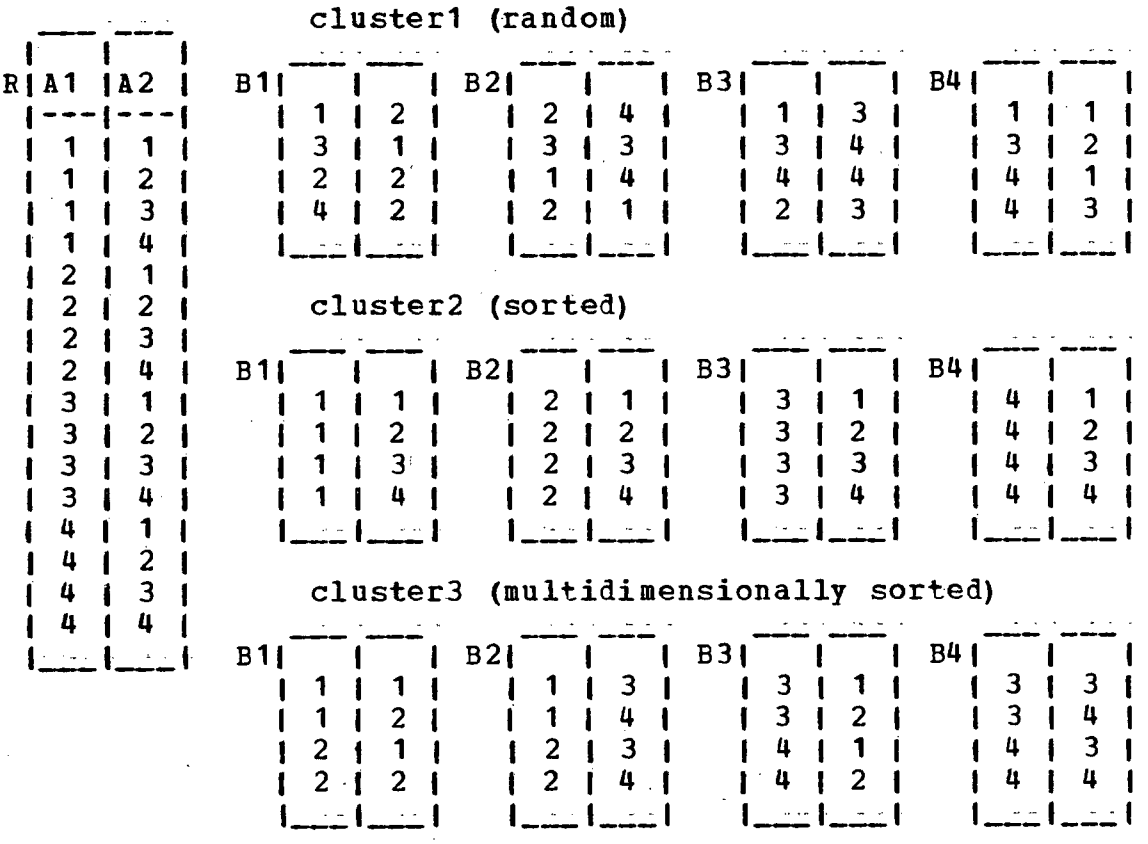


Figure II-3

clustering concerns document retrieval systems. How such systems are used differs from "normal" applications of a DMS. Users of document retrieval systems are usually never certain of what they wish retrieved. That is, records that may satisfy the query according to the user (and should therefore be retrieved) may not entirely fulfil the user's attribute/value constraints. Furthermore, users of such a system place greater weight on the system satisfying a "get about n of the most relevant records" constraint than on the more usual "get all and only relevant records".

In [Salton et al. 77] (see also [Salton 79a] and [Salton 79b]) a system is described that facilitates this environment. Although the horizontal clustering[1] accommodated by the system

Cluster Performance Relative to a Query Group

Query	cluster1 cost	cluster2 cost	cluster3 cost
A1=1	4	1	2
A1=2	3	1	2
A1=3	4	1	2
A1=4	3	1	2
A2=1	3	4	2
A2=2	2	4	2
A2=3	3	4	2
A2=4	2	4	2
totals	24	20	16

Table II-3

is more of a by-product than a design goal its realization is directly implemented as the set of lowest level cluster groups. Two aspects of search in their system are retrieval "precision" and "recall". In their words:

> "Precision is defined as the proportion of retrieval items actually found relevant, whereas recall is the proportion of relevant materials

---

[1]Their use of the word "cluster" has a different meaning from our own.

[2]This passage is taken from page 13 of [Salton et al. 77].

actually retrieved."[2]

Such characteristics of search will not be considered, however, in the proposed access path level. The assumption is made that "those asking" know exactly what they want. Furthermore, the passive environment of document retrieval is considered overly constaining. In contrast to Salton's static data structures dynamic support for optimal horizontal clustering is provided by the access path level. The dynamic nature relates to accommodating changing characterizations of usage.

Another important consideration in determining how to group the tuples of a base relation within disk blocks concerns each tuple's probability of access. Consider, for example, a relation R with the attributes SEX and NAME. Assume further that there are 5 tuples in R with SEX=MALE and 95 with SEX=FEMALE. If each disk block can hold 10 tuples from R and if half the queries specify SEX=MALE and the other half SEX=FEMALE then it is obviously desirable to locate the 5 tuples in R with SEX=MALE in the same disk block.

Such a clustering consideration is just another aspect of our notions of horizontal and vertical clustering since a characterization of use of R can of course convey information (such as retrieval probability) on individual tuples and attributes of R. It is nevertheless useful to distinguish tuple retrieval probability or any other information regarding individual tuples from horizontal clustering considerations. There are several reasons:

(a) Dynamic maintenance of a characterization of use for the purposes of horizontal clustering is greatly simplified,

(b) the problem of determining optimal horizontal clustering becomes much more managable when information regarding individual tuples does not have to be taken into account and

(c) in excising individual tuple retrieval probability from horizontal clustering considerations we divorce the important subject of information archival.

One possible accommodation for tuple retrieval probability might involve considering it a more general clustering problem than horizontal or vertical clustering. A base relation may then be distributed among several levels (i.e. different areas/types of storage); each of which is then dedicated to storing information with the same likelyhood of retrieval; each of which may then have horizontal and vertical clustering occurring within. As a result, relation R, above, would have tuples occurring in at least two levels since those with SEX=MALE have greater probability of retrieval. Such considerations, however, remain in the area of future research possibilities.[1] The horizontal clustering support discussed in the thesis assumes a uniform probability of retrieval of the tuples of a base relation.

Our definition of a secondary index has so far been very general. Automating the selection of indices requires a much

---

[1] One applicable reference is work in [Kollias et al. 77] regarding the automatic allocation of disk files to on-line and off-line disk packs.

more explicit understanding of their format and of how they are used in query evaluation. Details concerning both these issues with regard to secondary indexing considered by this thesis are given in Chapter III. There are, however, some general design characteristics that are common to almost all secondary indexing considered in the literature.

Secondary indices are usually implemented by inversions. This is in contrast to a multilist [Prywes and Gray 63] or multiring [Wiederhold 77] approach. Consider that each value of an indexed attribute has a set of tuples associated with it. Each tuple can be represented by a tuple pointer that is the physical address of the tuple in <page,displacement> form. An inversion approach physically stores each tuple pointer set with its respective value in the index.[1] In such a scheme it can be beneficial to compute tuple pointer set intersections and unions when evaluating queries that provide more than one indexed (inverted) value. Performing such operations is much easier and quicker if each tuple pointer set in the indices is maintained in sorted order.

The problem of secondary index selection for inversion approaches has been the subject of some research. In [Schkolnick 74] and [Anderson and Berra 77] functions are derived that produce a cost value for a given permutation of

---

[1] See [Salton et al. 77] for an excellent brief outline of "inverted file" and "multilist" approaches to secondary indexing.

indexing on the attributes of a base relation. Schkolnick demonstrates a "regularity condition" on his cost function permitting a more efficient algorithm to find the optimal indexing set whereas Anderson and Berra rely on a brute force approach. The latter include much more implementation detail into their cost function however.

Schkclnick assumes that computing intersections and unions of tuple pointer sets retrieved from secondary indices can always be accomplished in main memory and therefore contributes negligibly to overall cost. His function incorporates retrieval and maintenance (i.e. update, insertion and deletion) costs.

In addition to retrieval and maintenance Anderson and Berra consider the cost of storage. They also deal in much more detail with various forms of inversions. In particular they consider the merits of maintaining sorted tuple pointer sets over unsorted sets.

A much more pragmatic approach to secondary index selection is described in [Farley and Schuster 75]. The emphasis in their paper is on details concerning query evaluation on single relations with secondary indices on single attributes implemented as inversions with sorted tuple pointer sets. The strategy adopted in query evaluation incorporates the use of a cost function.[1] This function can be

---

[1]This is true of all query evaluation schemes of all RDMSs discussed in this thesis.

used to derive an overall cost of evaluation of any query given a particular indexing set. Using this cost function along with a representative sample of past queries allows them to measure the performance of any proposed indexing set. As in [Anderson and Berra 77] they then follow a brute force approach in deriving this measurement for each possible indexing set.

Their approach, however, fails to take into consideration both maintenance and storage costs (these considerations along with the ultimate indexing choice are left to the database administrator). Another simplifying assumption adopted by Farley and Schuster as well as Anderson, Berra and Schkolnick is that secondary indexing on single attributes only is permitted. Recall that an image in System R can be based on more than one attribute. The author is not aware of any work regarding secondary index selection dealing with this generalization.

The subject of dynamic secondary index selection is clearly non-trivial (see for example [Comer 78]). As in the case of clustering, such selection is determined primarily by how base relations are used. A suitable characterization of use, therefore, is as important to secondary index selection as it is to clustering considerations.

One last issue concerning problem definitions ends this section. This concerns terminology used in describing various query types. All definitions given are relevant to an access path level of an RDMS. Furthermore, any instance of any of the

following query types (the semantics of these is obvious) is assumed to apply to a relation R with k columns:

D7. partial match - A partial match query provides a value for a proper subset of the attributes of R. It has the form:
{COLn1=VAL1} & {COLn2=VAL2} & ... & {COLnj=VALj}
where j<k and the COLni are distinct (i.e. i¬=m implies ni¬=nm).

D8. exact match - An exact match query provides a value for each column of R. Its form is as above save that j must now equal k.

D9. partial range query - A partial range query provides a range of values for a proper subset of the attributes of R. It has the form:
{COLn1 between VAL11 and VAL12} &
{COLn2 between VAL21 and VAL22} & ...
... & {COLnj between VALj1 and VALj2} where j<k and the COLni are distinct.

D10. range query - A range query provides a range of values for each attribute of R. Its form is as above save that j must now equal k.

D11. intersection query - Informally, an intersection query is disjunctions of the above. It has the form:
{Q1 or Q2 or ... or Qm} where m>0 and each Qi is a query of any variety of those above.

D. Simplifying Assumptions

There are a number of simplifying assumptions regarding the hardware environment for a RDMS. The most important concerns the mass storage media itself. In particular, tape storage (along with the sequential mode of use it implies) is no longer permitted for applications other than information archival and backup, statistics acquisition and transaction logging. Currently active base relations are assumed to reside

on storage media of at least the performance of moving head hard disks.

Another important assumption concerns the amount of main memory and centrol processing unit (CPU) overhead required to support a RDMS. With phenominal technological advances in logic densities it is now feasible to assume an ability to allocate large amounts of main memory to buffering applications and even to dedicate CPUs to the RDMS software. Traditional balancing constraints for information control systems (and even operating systems) such as main memory cost and CPU overhead are no longer considered to apply. Taking their place is a requirement for generality, homogeneity and ease of use. Response time is considered a more important measurement of system performance than either throughput or minimization of hardware costs.

Several assumptions concerning the applications environment are made. Dedicated real time control applications, for example, are not supported by the access path level. A requirement to ensure a constant upper bound for any operation provided at the access path level would add considerably to the complexity of the issues considered. Furthermore, the very important topic of distributed support for a RDMS relational language level is not considered. Because of the emphasis on response time no attempt is made to apply batching techniques to query evaluation or base relation

maintenance.[1]

The access time for a block on a moving head hard disk is comprised of three major components. The first of these, called seek time, is the time required to physically move a read/write head to the desired track. Rotational delay is the time required for the read/write head to locate the start of the desired block on the track. The transfer time taken to read the desired block is the last of these access time components.

The problem of locating the blocks containing the tuples of base relations on the disk in a way that reduces the overall seek and rotational delay times in accessing these blocks is also not considered by this thesis. Both this and the scheduling of disk I/O operations are considered the responsibility of lower level software or hardware.

---

[1]Recall that the designers of SODMS assumed just the opposite.

Chapter III: The System

A. Overall Design

The first part of this section concerns the specification for a proposed access path level of a RDMS. The overall design for an implementation called SORAAM (for Self-Organizing Relational Associative Access Method) that attempts to satisfy this specification completes the section.

Generally, the access path level provides the initial relational abstraction. To accomplish this, control over all physical organization details concerning all base relations resides completely within it. This includes the following clustering and indexing considerations:

(a) dynamic support for the horizontal clustering of all base relations and

(b) automatic selection of single-attribute inversions.

The decisions at the access path level with regard to horizontal clustering and secondary index selection for a base relation are governed by a variety of statistics that determine a characterization of use and secondary index usefulness for the base relation. All statistics used by the access path level are maintained by it.

There is good reason, however, to endow the access path level with an ability to take "advice" in regard to physical organization details. Such situations requiring this are an initial base relation creation (when the system has had no

opportunity to acquire a characterization of use itself) and the acquisition of a priori knowledge by a user of a dramatic change in the characterization of use of an existing base relation (not predictable by the system). The latter case may imply an immediate alteration on the currently existing secondary indexing set. The advice taking ability is accomplished by permitting the user to directly override the control information used by the access path level to govern physical organization.

Support for associative access to information is provided for single base relations only. This is consistent with the capabilities of access path levels of most existing RDMSs. Furthermore, finding all tuples of a base relation satisfying any given search predicate never requires consideration of a tuple within the base relation more than once. That is, a tuple may be accepted or rejected for retrieval on first examination. In the worst case, therefore, responding to a query may require at most one complete scan of all the tuples within a base relation. Nevertheless, all the following variety of queries are directly supported (see section C of Chapter II for the definitions):

(a) exact match

(b) partial match

(c) range query

(d) partial range query

(e) intersection query

In section D of Chapter II a number of simplifying

assumptions concerning the proposed environment were outlined. Assuming all of these, the access path level still retains a large measure of suitability for its proposed environment (i.e. the access path level of a RDMS which replaces the file system of an operating system). Such an environment, however, necessitates support for a large measure of concurrency of access and maintenance on all base relations. This includes concurrency at both the logical level (more than one ongoing access/modification on the same base relation) and physical level (more than one ongoing access to the same physical block or disk page). In addition, a type of concurrency permitting normal use of all base relations during any physical reorganization is essential (see section A of Chapter II). We distinguish this latter as "reorganization concurrency" from the preceeding reference to "user concurrency". In facilitating reorganization concurrency every effort is made to minimize the degeneration of retrieval performance of those base relations undergoing physical reorganization.

User concurrency may be understood as a type of multiprogramming. In this case the individual "jobs" are transactions. A transaction in this sense is a sequence of (usually more than one) retrieval and maintenance operations on a set of base relations. The "shared resources", therefore, are the base relations themselves. As a result, locking requirements arise in order to ensure the logical integrity of all base relations to each transaction.

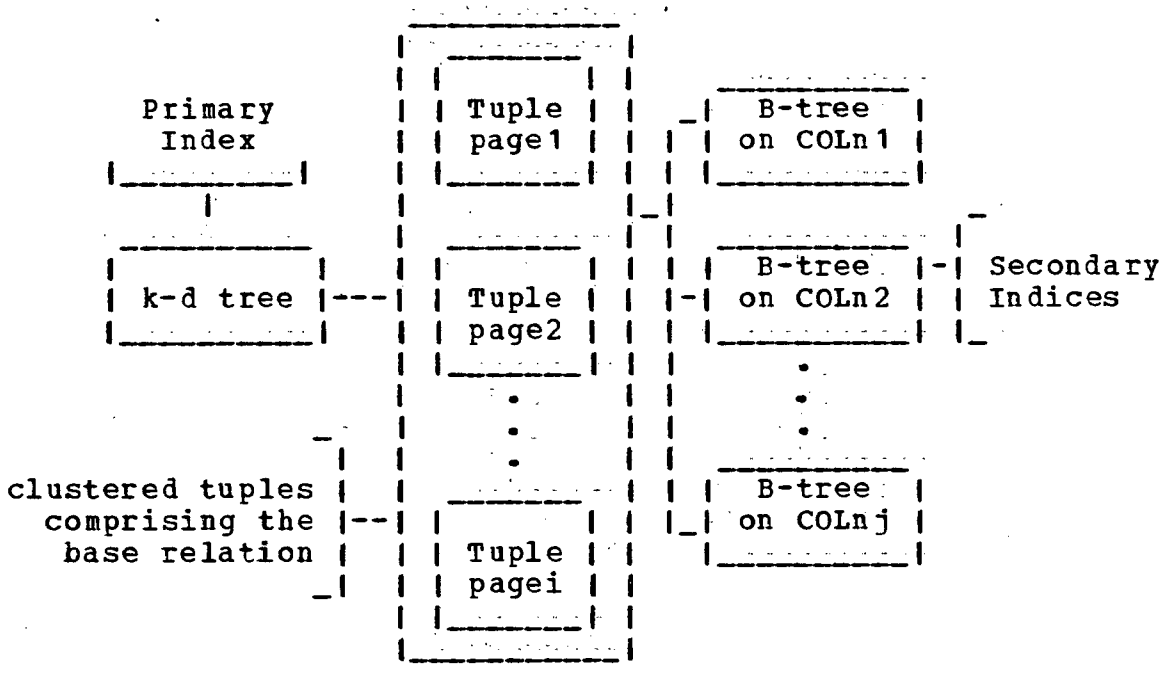
All such locking requirements (both the logical and

physical levels) are the sole responsibility of the access path level. Because of this the level of user concurrency that may arise can result in transaction deadlocking. The facilities necessary to support the locking requirements must therefore include capabilities for transaction deadlock detection. Furthermore, recovery facilities necessary for deadlock recovery (and, of course, system failure) are called for. All such recovery facilities are also the sole responsibility of the access path level.

We begin our description of the proposed system by an introduction of the layout and data structures resident on mass storage that are used by SORAAM to access and maintain each base relation. Figure III-1 below illustrates the storage structures involved (the base relation is assumed to have at least j attributes).

The primary index of a base relation used to control the horizontal clustering of the base relation is the k-d tree data structure (see [Bentley 75], [Bentley 78a] and [Bentley 78b]). This data structure is extremely robust in the variety of queries that it supports. In fact, any instance of an intersection query can always be evaluated with the exclusive use of the k-d tree primary index. Note therefore that, given the definition above of the variety of valid queries to our access path level (see the user interface), the use of the primary index is <u>always</u> possible. This fact is incorporated into the search strategies used (see Section E of this chapter).

Storage Structure of a Base Relation in SORAAM



The indexing set is: {n1, n2, ..., nj}

Figure III-1

Another very important property of k-d trees with respect to horizontal clustering is that, with a suitable characterization of use of the base relations and control of the k-d trees, optimal horizontal clustering of their tuples results. Both this property and the control issues are discussed in detail in Sections B and C of this chapter.

The secondary indexing illustrated by Figure III-1

involves the use of the B-tree data structure.[1] A B-tree index may exist for each possible "relation/attribute" pair of the base relations. In the example illustrated j attributes of the base relation are each indexed by a B-tree. In SORAAM a base relation need not be fully inverted (i.e. a B-tree for each possible attribute). We denote the set of attributes for a base relation that are indexed as the "indexing set".

There are three user supplied parameters for each base relation that control its indexing set. These parameters (Plow, Phigh and S) are introduced below in Table III-1. The parameters along with various other statistics (including the important "global performance measure" or g.p.m.) that are maintained for each base relation are used by our system to control the selection of the single attribute inversions. Section D of this chapter presents more fully the details of these aspects of SORAAM.

There are three base relations that must exist in all implementations of SORAAM (i.e. that are required and used by SORAAM itself) called RELATIONS, ATTRIBUTES and PROCESSES. The existence of all three is necessary for the proper operation of the system. The tuples in RELATIONS and ATTRIBUTES define

---

[1] For an excellent introduction to B-trees see [Comer 79] or [Knuth 73]. [Bayer and McCreight 72] first introduced these structures. There has been much work on B-trees since. [Bayer and Scholnick 76], for example, discuss an interesting variant called the prefix B-tree and [Bayer and Unterauer 76] consider concurrency of operations on them. [Lomet 79] presents an interesting method of organizing internal nodes of a prefix B-tree.

the existence, location and format of all base relations and their attributes respectively. Furthermore, all statistics used by the access path level are also contained in these relations. PROCESSES exists for the purpose of those aspects of SORAAM concerning transaction maintenance. Tuples in PROCESSES convey the existence of processes with ongoing transactions and their respective recovery points (we elaborate on this in the outline of the user interface).

Table III-1 illustrates the format of these base relations. The attribute(s) enclosed in parenthesis for a

Special Relations Used By SORAAM

Relation Name	Attribute List
RELATIONS	(Rname), Bsize, N, D, PIB, PIBloc, IX, STPB, ff1, ff2, Psize, g.p.m., Plow, Phigh, S, qno, qnoold, radix1, RDQreads, DQreads, radix2, Pcost, radix3, buf
ATTRIBUTES	(Rname, Aname), COL#, format, n, d, Iflag, Isize, SIB, SIBloc, f, fold, min, max, b
PROCESSES	(process#), quiet#

Table III-1

relation comprise the key to that relation. All users (or processes) are permitted access and maintenance operations on them (no security considerations are assumed by SORAAM). All operations on other base relations require their definition by appropriate entries in RELATIONS and ATTRIBUTES. A description

of each of the attributes in Table III-1 is given in Appendix A.

One of the decisions regarding the user interface to SORAAM was to require no expression parsing. Because of this the user interface are varieties of tree-like structures with the nodes themselves denoting the operations required and the conditions for search as well as serving simply as buffers for tuples and log records. In a sense, "operation trees" submitted to SORAAM are analogous to a parse tree created by the parse phase of a compiler. This approach to interfacing is not, however, unique to SORAAM (see for example the LSL interface of OMEGA in [Schmid et al. 76]).

Communication, therefore, involves a variety of sixteen node types the first ten of which serve as choices for the root node of an "operation tree". Each of the six non-root node types is associated also with a "syntactic identifier" enclosed in brackets of the form "<...>" which, in turn, denote possible pointer values in the pointer fields of node types having them. The first field of all node types identifies the node type to SORAAM. All but one of the node types (type 14; or <values>) have a fixed format with a fixed number of fields.

All operations on SORAAM may be categorized as one of three types: base relation retrieval, base relation maintenance and transaction maintenance/recovery/audit operations. In the following, node types have the form "{n,...}" (the first field being a numeric value identifying

the node type). Other fields have one of three other forms: a value of an attribute as described in Appendix A, a syntactic identifier denoting a pointer field to other non-root nodes or a fixed selection of values. Brackets of the form "[...]" enclose selections in the latter case. Also, fields tagged with "*" must be supplied by the user. We first introduce the variety of root node types possible then follow this by a description of the syntactic identifiers along with their associated non-root node types.

The reader is encouraged to bear in mind when reading the following that the design of the interface to SORAAM is not intended as an end user interface. Knowledge and use of this interface is properly the responsibility of a relational language level that would comprise the top end of a full RDMS.

Retrieval Operations

There are two root node types available for retrieval operations (for expressing queries). These have the form:

(a) {0*,process#*,Path#,Rname*,<predicate>*,<values>}

(b) {1*,Path#*,<values>}

We note first that "Process#" identifies the process issuing the retrieval operation to SORAAM. Both of these conform to a notion of a "RETRIEVE" and "NEXT" operation respectively that essentially "linearizes" the return from a base relation of the tuples satisfying a predicate of a query. For a RETRIEVE operation SORAAM first assigns a unique "Path#" that is used

by SORAAM to identify the sequence of tuples to be returned. The first tuple in "Rname" satisfying "<predicate>"[1] is then returned in "<values>". Subsequent tuples are returned via subsequent user supplied NEXT operations using the provided "Path#" value.

Such a linearization of retrieval response is not, of course, clearly desirable. As a consequence greater responsibility for retrieval control rests on higher level users (i.e. the relational language level). We feel, however, that the benefits resulting from greatly simplified buffer management far outweigh any such disadvantages.

The format of these operations facilitate more than one ongoing retrieval by the same process. Furthermore, no opening or closing of any form is called for. Clearly there is, however, an implementation issue concerning values of "Path#". There is a question of under what conditions may a path number and its associated resources be freed for use by other retrieval operations. Exhausting the set of tuples satisfying "<predicate>" is one such condition but the question deserves greater thought. We leave, for now however, this implementation issue.

Maintenance Operations

---

[1]If no <predicate> is supplied in the RETRIEVE node then all tuples in "Rname" are assumed to qualify for retrieval. A scan of all tuples in a base relation is therefore facilitated.

There are three root node types available for maintenance operations on base relations. Their form is:

(a) {2*,process#*,Rname*,<values>*}

(b) {3*,process#*,Rname*,<predicate>*}

(c) {4*,process#*,Rname*,<predicate>*,<values>*}

These correspond to "INSERTION", "DELETION" and "UPDATE" operations respectively. All fields in the above are user supplied. Besides identifying the requesting process to SORAAM the "process#" field is also necessary for purposes of transaction recovery. Destructive operations on base relations are logged along with the value of "process#" to facilitate this. The role of the other fields in the performance by SORAAM of these maintenance operations should be obvious.

Transaction Maintenance/Recovery/Audit Operations

The first four of the root node types described below provide for transaction maintenance and recovery. Audit trail requirements are facilitated by a logging root node type. These formats are:

(a) {5*,process#*,quiet#}

(b) {6*,process#*}

(c) {7*,process#*,quiet#}

(d) {8*,process#*,quiet#*}

(e) {9*,process#*,<buffer>*}

These correspond to "begin transaction" (BEGINTRANS), "end transaction" (ENDTRANS), "checkpoint" (QUIET), "checkpoint

recovery" (RECOVER) and "LOG" operations respectively. The first four are entirely analogous to the System R RDI (Relational Data Interface) operators "BEGIN_TRANS", "END_TRANS", "SAVE" and "RESTORE" (see [Astrahan et al. 76]). A "LOG" operation involves writing the contents of "<buffer>" to a system log. Recall that maintenance operations use this same system log (see above).

All maintenance operations on base relations must occur inside "BEGINTRANS"/"ENDTRANS" pairs. In this way all processes not currently involved in a transaction can be assured (for their retrieval operations) of both the logical and physical integrity of the base relations. Furthermore, all processes with ongoing transactions are "insulated" from feeling the effects of other ongoing (i.e. uncompleted) transactions. The "QUIET" and "RECOVER" operations are useful for large transactions involving a great deal of maintenance on the base relations. Recovery from a deadlock may then involve backup to a more recent "quiet#" of the transaction.

Syntactic Identifiers and Non-root Nodes Types

We now introduce the variety of non-root node types that can occur in an operation tree communicated to SORAAM. Each is associated with a syntactic identifier some of which were used to denote pointer fields in the above. A syntactic identifier, however, may sometimes involve more than one form. We therefore deal with each such identifier in turn.

Retrieval and maintenance operations described above allow the use of a <predicate>. This has one of three forms (the first two of which are other syntactic identifiers):

(a) <primitive>

(b) <conjunct>

(c) {10*,<conjunct>*,<predicate>*}

Note that the third form of <predicate> is a non-root node type that permits a pointer field to another <predicate>. This third form is interpreted by SORAAM as a disjunction (OR) of the form's second and third argument.

A <conjunct> syntactic identifier has the following two possible forms:

(a) <primitive>

(b) {11*,<primitive>*,<conjunct>*}

Note again that the second form of <conjunct> is a non-root node type that permits a pointer field to another <conjunct>. Analogously, this second form is interpreted as a conjunction (AND) of the form's last two arguments.

A <primitive> syntactic identifier also has two possible forms. Both of these, however, are non-root node types:

(a) {12*,Aname*,[< | <= | = | ¬= | >= | >]*,value*}

where "value" is a value of attribute "Aname".

(b) {13*,low*,|< |*,Aname*,|< |*,high*}
|<=| |<=|

where "low" and "high" are values of the attribute "Aname".

The first of these non-root node types (note that both are leaves in an operation tree) is interpeted by SORAAM as specifying an explicit value or an upper or lower bound for an attribute. The second is interpreted as specifying a range of values for an attribute.

Consider now that the definition of the <predicate> syntactic identifier resulting from the above is nothing more than a tree structured disjunctive normal form expression. The <primitive> forms allowed, therefore, facilitate the expression of any instance of an intersection query as a <predicate> to SORAAM. Furthermore, the tree structure corresponds to an easily used parse of the query. We shall use the term "<predicate> instance", therefore, to denote a non-null <predicate> field occurring in RETRIEVE, DELETION or UPDATE root node types.

The two remaining syntactic identifiers used by root node types are <values> and <buffer>. Both have only one form that are non-root node types. The forms for <values> and <buffer> respectively are:

(a) {14*,Aname1*,v1*,Aname2*,v2*,...,Anamei*,vi*}

where v1*,v2*,...,vi* are values of attributes Aname1,Aname2,...,Anamei respectively.

(b) {15*,log}

where "log" is the text to be written to the log file.

The interpretation of the above non-root node types by SORAAM should be obvious.

An Example Retrieval

We conclude this first section of Chapter III with a much needed example use of SORAAM. The operation tree for a query on an example base relation is given.

Consider that the base relation "PARTS" on which our query applies has the following attributes (given in brackets):

PARTS[P#,PNAME,PCOLOR,PWEIGHT,QTY]

The key of the relation is the "P#" numeric attribute. Other numeric attributes are "PWEIGHT" and "QTY". Each tuple in "PARTS" describes a specific part by its name, color, weight and quantity. Note that tuples describing "PARTS" must already reside in "RELATIONS" and "ATTRIBUTES" before any operation is possible on it. There are of course six such tuples necessary (one for the relation entry; five for the attribute entries). All six would have been added by prior maintenance operations on SORAAM.

The query we consider may be expressed in English as:

"Get all bolts more than 10 ounces and nuts of which we have less than 40."

This immediately decomposes into two different operation trees corresponding to:

(a) Get the first nut or bolt then

(b) get the rest.

Figure III-2 illustrates the operation trees for both parts.

Example Operation Trees On SORAAM

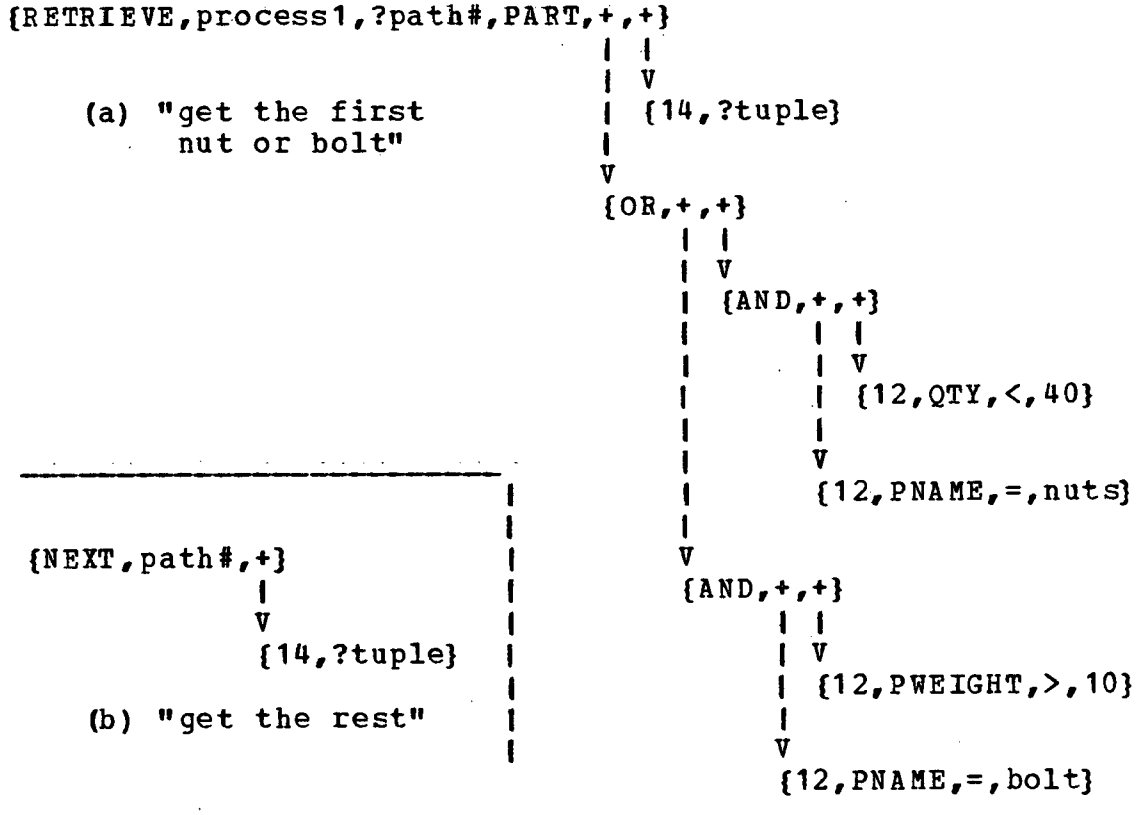


Figure III-2

For readability, node type fields have been replaced by more meaningful mnemonics. The first defines the relation and predicate to SORAAM. The first tuple is returned in "?tuple"

and SORAAM assigns a "path#" value. This value is then used in the second operation tree by the root "NEXT" node to acquire, via "?tuple" again, the next nut or bolt. Successive use of the second operation tree accomplishes "getting the rest".

B. The Primary Index

In preceeding sections we have discussed the role of the primary index of a base relation. In general, the primary index determines the clustering of the tuples in fixed sized blocks. With SORAAM, however, we have limited ourselves to a consideration of horizontal clustering considerations only. Recall that such considerations themselves are defined in terms of a characterization of use of the base relation.

In this section we introduce our chosen characterization of use. Consideration is then given of a number of access methods used as primary indices to realize desirable horizontal clustering determined by the characterization. The first section of this thesis introduced the particular data structure that we have adopted for this role as the k-d tree. An introduction to k-d trees and an initial demonstration of their suitability are presented followed by a detailed examination of the problems that arise. The next section considers in detail the specific problem of "discriminator selection".

The characterization of use of a base relation used by

SORAAM is a set of statistics based on the history of queries on the base relation. All such statistics occur in the special relations "RELATIONS" and "ATTRIBUTES" as values of the attributes "qno" and "f". Each base relation has a "qno" value and a value of "f" associated with each of its attributes. The "qno" value represents a count of the total number of queries that have been applied to a base relation and the "f" for each attribute represent a count of the number of times a value for each was supplied in past queries.

The description of the user interface in the preceeding section introduced and defined the syntactic identifier "<predicate>". Each occurrence of a <predicate> in a "RETRIEVE", "DELETION" or "UPDATE" root node type of an operation tree (each <predicate> instance) received by SORAAM is interpreted as an occurrence of a query. The arrival of one of these root node types together with its <predicate>, therefore, signals to SORAAM to increment the "qno" value of the associated "Rname" (given also in the root node) in RELATIONS. For each "<primitive>" occurring in the predicate there is an associated attribute name "Aname". On <predicate> arrival these attribute names together with "Rname" of the root node are used by SORAAM to locate and increment the associated "f" values. In this way the following information regarding a characterization of use of each base relation is available to SORAAM:

(a) a relative frequency of specification count "f" for each "Rname/Aname" pair and

(b) with respect to the above a relative count "qno" of the total number of queries on a base relation.

If the maximum value a numeric attribute such as "qno" may have is a limiting factor then "recovery" from reaching this maximum is necessary. Clearly there is a danger that the statistics described above may become useless if the "qno" and "f" reach their maximum values and stay that way. A particularly simple recovery approach, then, is simply to divide the "qno" and "f" values of a base relation in half whenever "qno" exceeds its maximum. Note that the "f" values, as a result, are ensured never to exceed the value of "qno" and, more importantly, stay in proper proportion to each other and to "qno" (thus preserving the quality of the characterization of use).

There is an interesting repercussion of this recovery approach that results from a loss of information whenever a "divide by 2" is called for. The maximum value then determines how much of the query history will actually be remembered. Because these statistics control horizontal clustering (and because the support for horizontal clustering in SORAAM is dynamic in nature) control over the maximum value yields control over the rate of adaptation of the horizontal clustering to changing patterns of use. The attribute "radix1" in RELATIONS is provided for just this purpose. Reducing the value of "radix1" increases the rate of adaptation of the horizontal clustering (and also the system overhead necessarily involved) while increasing "radix1" results in

greater immunity to short and temporary bursts of unusual use but less willingness to change.[1]

There are a number of existing access methods that might be used as the choice for the primary index of base relations in SORAAM. The access methods we consider take progressively more advantage of the above frequency of access statistics comprising the characterization of use to accomplish progressively better degrees of horizontal clustering.

Possibly the least likely candidates for the primary index are methods involving hashing techniques. Their most condemning characteristic is clearly the resulting "randomization" of the relative physical locations of tuples in a base relation.[2] Regardless of the choice of columns that comprise the domain of a hash function no significant relationship can result for tuples located physically near to each other. No characterization of use, therefore, can have any constructive bearing on a primary index based on a hashing method.

Another major problem with a traditional hash file is the requirement of knowing in advance the size of the file expected (i.e. the number of tuples comprising a base relation). This is necessary in order to determine an appropriate range for the hash function itself. This problem,

---

[1] A new insight in cognitive science perhaps.

[2] Note that general purpose order preserving hashing functions do not exist.

however, has been recently overcome with the advent of an "extendable hashing" access method first introduced in [Fagin 78]. The expected case performance of retrieval on the hash key using this method is two block reads for any file size.

Other candidates for the primary index are methods involving tree structures. Of these the most predominant currently used by existing RDMSs for both primary and secondary indexing is the B-tree. There are several reasons for this. For one, a B-tree always remains balanced regardless of the variety and number of insertions and deletions performed on it. Since no degeneration of search performance occurs no physical reorganization is ever necessary. Furthermore, by its nature, a B-tree can be used to maintain a physical sequence of tuples in a sort order determined by an ordered subset of the attributes of a base relation.

This second property of B-trees endows them with an ability to make constructive use of the characterization of use statistics introduced above. The sort order of the tuples of a base relation would be chosen as the order implied by the sequence resulting from arranging the attributes of the base relation in descending order of their "f" values. The primary sort column would therefore correspond to the attribute most frequently specified in queries. Tuples would then be physically stored nearer those with similar values for this primary sort attribute. Thus we can see that a beneficial horizontal clustering results.

Recall from Figure II-3 and Table II-3, however, that the

horizontal clustering resulting from the maintenance in sort order of the tuples of a base relation can be improved upon. In fact a third clustering, "cluster3", yielded an absolute improvement in block reads over a sorted clustering (that would result from the use of a B-tree as the primary index) equaling the improvement gained by the sorted clustering over a random clustering (that would result from the use of extendable hashing for example). Figure III-3 illustrates the realization of "cluster3" in Figure II-3 with a "k-d tree" index (another tree structure). Clearly, then, a k-d tree holds the potential for improved horizontal clustering.

For this reason a k-d tree data structure was chosen as the main contender for the role of primary index of all base relations in SORAAM. We shall see in the next section that its suitability for this role is enhanced by the fact that, with various simple interpretations of our characterization of use, an optimal horizontal clustering results. A simple introduction to k-d trees now follows.

Each internal or non-leaf node in a k-d tree has the fields defined by the layout diagrammed in Figure III-3. The latter three fields (i.e. "key value", "left subtree" and "right subtree") are entirely analogous in purpose and function to those of a regular binary tree. The additional "discriminator" field simply determines the column or attribute to which the "key value" applies. "level 1" of the k-d tree diagrammed (the root node) discriminates on the first column while "level 2" discriminates on the second. Therefore,

A k-d Tree Index to "cluster3"
(from Figure II-3)

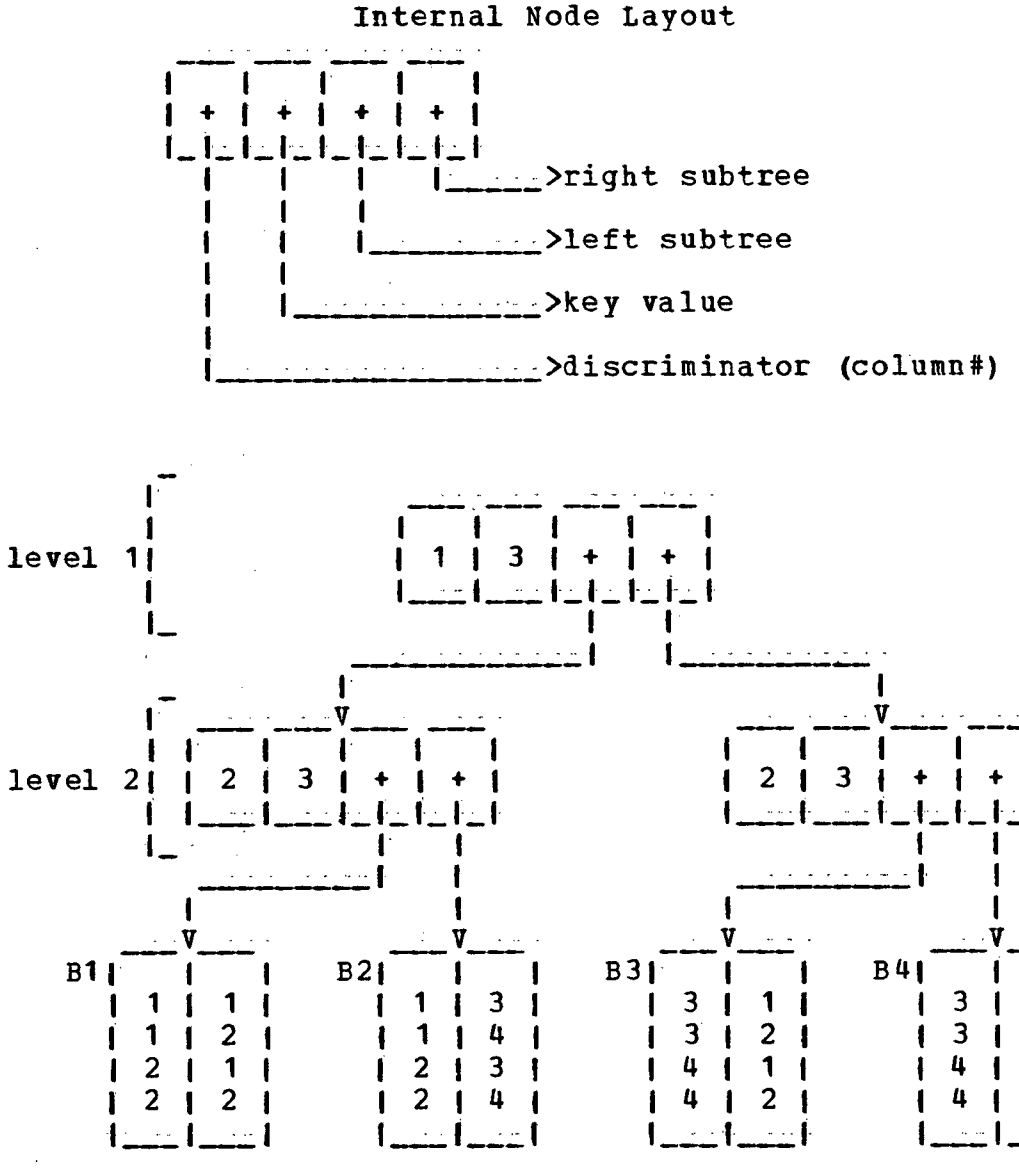


Figure III-3

only tuples with values for their first (second) attribute greater than or equal to 3, the "key value"s, occur in the right subtree of the root node (level 2 nodes).

Searching on a k-d tree is simple. Consider an exact match query that would arise, for example, from tuple

insertion. Starting at the root node we follow the left or right pointers as implied by the comparison of the appropriate attribute value determined by a node's discriminator with the node's key value. In this way we arrive at a single leaf block in which we must then search.

In a partial match query, however, a node may discriminate on an attribute for which no value is provided. In this case the search must recurse down both the left and right subtrees. As an example consider the partial match query {column#1=2} applied to the base relation given in Figure III-3. We would follow the left subtree of the root node since the first column value provided by the query compares less than the root node key value of 3. However, since no value is provided for the second column in our query both the left and right subtree of the level 2 node must be searched. As a result the first two leaf blocks must be searched for tuples satisfying our query {column#1=2}. Note that both range queries and partial range queries can similarly be facilitated.

There are, however, a number of problems with k-d trees:

(a) With respect to our characterization of use how should discriminator values be selected in order to arrive at a best possible horizontal clustering of the tuples of a base relation?

(b) How do we ensure balancing of the internal nodes of a k-d tree?

(c) How do we cluster the internal nodes of a k-d tree themselves within fixed size blocks in order to ensure minimal disk accesses?

(d) How do we facilitate dynamic physical

reorganization of the horizontal clustering necessitated by a changing characterization of use?

We defer discussion of the first of these problems until the next section. The latter three of the problems, however, are considered in the remainder of this section.

Our first problem, then, concerns balancing of the internal nodes. Imbalance in a k-d tree can occur when the tree itself must be dynamically generated according to the tuple insertion sequence (i.e. the sequence of INSERTION operations on a base relation submitted to SORAAM). To better understand how this problem arises it is necessary to have a clearer idea of the insertion process through the k-d tree primary index.

Figure III-4 below illustrates both this insertion process and a resulting imbalance of the internal nodes (more specifically the root node) arising when the last tuples are added. The leaf blocks in the figure contain the tuples and are labelled according to their order of creation. Each leaf block is assumed to have a capacity of at most four tuples. A label of the form "*n" denotes the order of tuple arrival.

The phrase "leaf page splitting" best describes the event resulting in the creation of a new leaf block and internal node. Such page splits occur when a new tuple must be added to a leaf tuple block already full. The arrival, for example, of the fifth tuple forces both tuple block "B1" in Figure III-4(a) to be split into two tuple blocks "B1" and "B2" and an internal node created to distinguish between their tuples in

Sorted Order Insertion in a k-d Tree

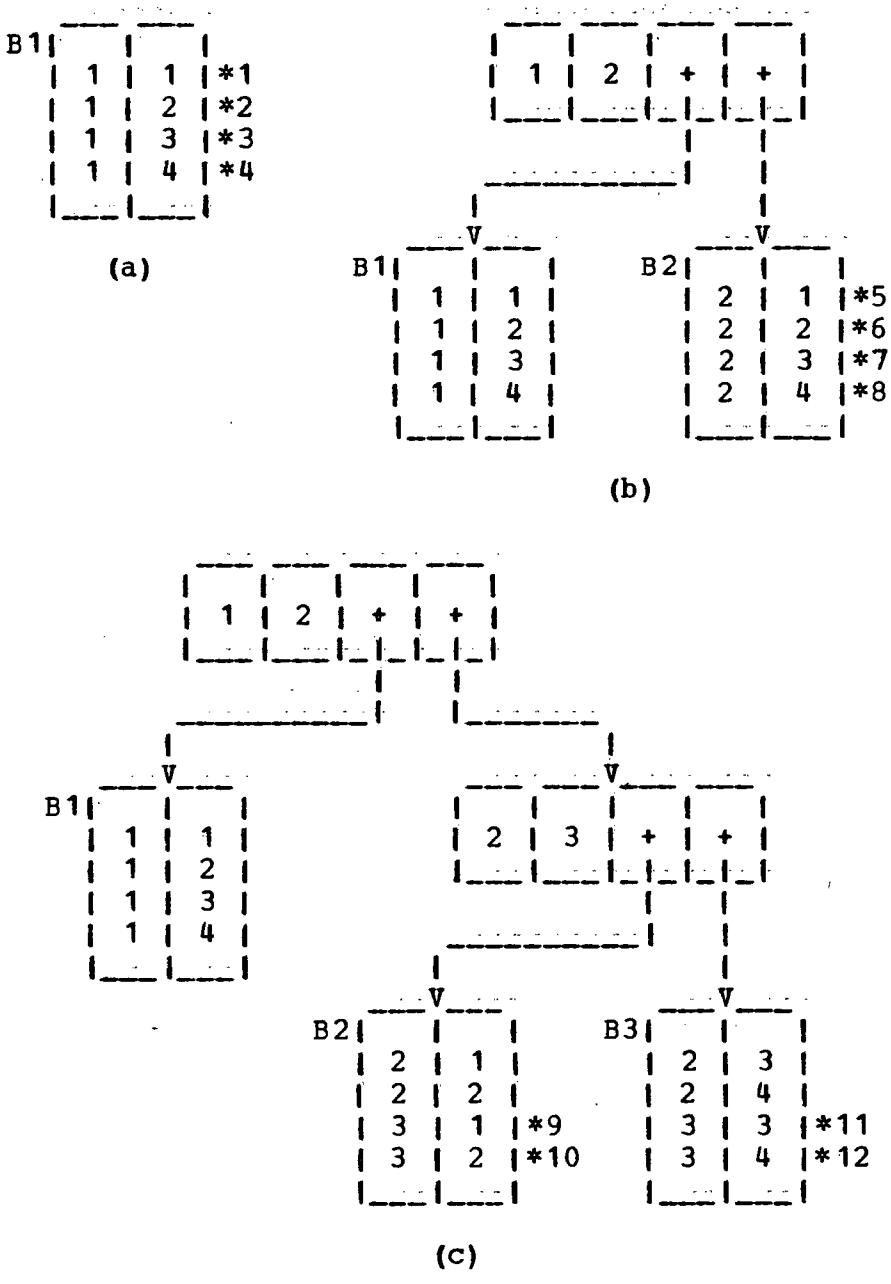


Figure III-4

Figure III-4(b). In a similar fashion tuple block "B2" is split twice more on arrival of the ninth and thirteenth tuples.

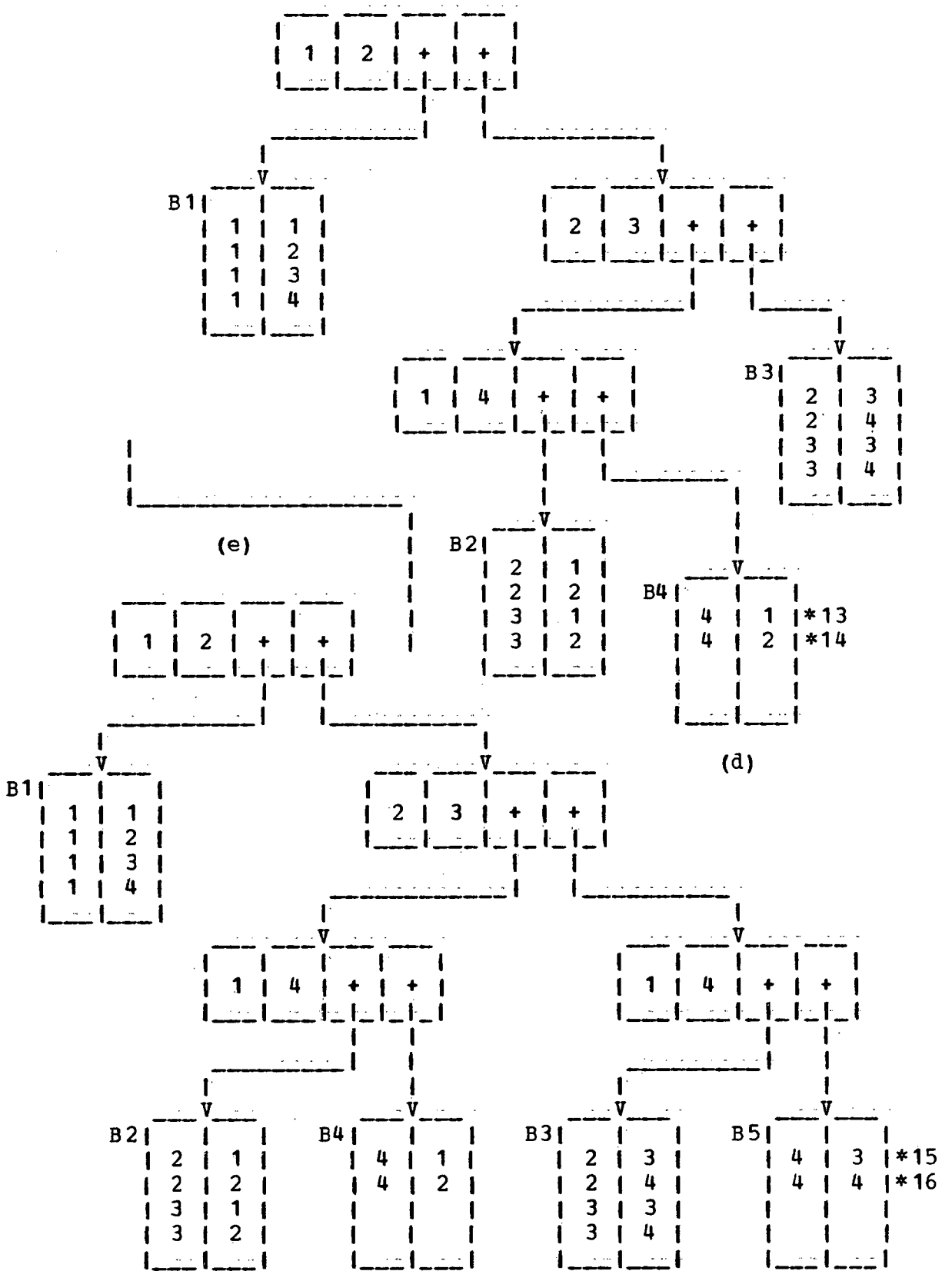


Figure III-4 (cont'd)

The first thing to remember about k-d trees is that they behave exactly as 1-d (or binary) trees in expected case performance for random tuple insertions:

> "Thus we know that typical insertions and record lookups in a k-d tree will examine approximately $1.386 \log_2 n$ nodes."[1]

Indeed, should the tuples of Figure III-4 have been inserted in a random order illustrated by "cluster1" of Figure II-3 instead of the sorted order illustrated by "cluster2" of Figure II-3 the perfectly balanced k-d tree illustrated in Figure III-3 would have resulted (the reader is encouraged to verify this for himself).

One must realize, however, that k-d trees are basically a static data structure.[2] Problems arise when insertions are not random. In such cases a k-d tree can become unacceptably unbalanced (possibly when insertions are made in some sorted order as in Figure III-4). Balancing "tricks" used by 1-d trees such "rotations" in the case of AVL trees and "internal node splitting" in the case of B-trees cannot generally be applied to k-d trees. The next best approach, then, is to design a rebuilding phase that accomplishes balancing but that

---

[1] This passage is taken from page 512 of [Bentley 75]. A derivation of this result for binary trees may be found on pages 426-427 of [Knuth 73].

[2] A "dynamic multidimensional searching data structure" appropriate for exact or partial match and range or partial range searching and efficient in both space and time (no worse than O[n log n] for either) does not yet exist.

permits concurrent normal use made of the k-d tree primary index and base relation. The phase would be initiated upon detection of a suitable imbalance condition.

There are many ways that imbalance can be detected. A particularly simple condition suggested by Knuth (see page 451 of [Knuth 73]) that we adopt has the form:

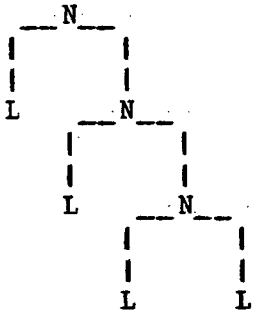
$$D \geq c \cdot \log_2 (IX+1) \qquad (1)$$

In the above expression "D" represents the maximum depth of a k-d tree index and "IX" the number of internal nodes. Both values are automatically maintained by SORAAM and are located in the "RELATIONS" special base relation (see Appendix A). Note that the number of leaf tuple blocks equals IX+1. A minimum value of 1 for the constant "c" implies intolerance for any but perfectly balanced k-d trees. Higher values of c allow greater degrees of imbalance to exist in the primary index. For example, c=1 would imply that the k-d tree in Figure III-4(e) was imbalanced whereas c=1.5 would not. The condition (1) would be checked for a base relation whenever a page-split results from an insertion operation (thus incrementing the value of IX for that base relation).

The rebuilding phase adopted by SORAAM involves the random deletion and re-insertion of the tuples of the imbalanced k-d tree. In this case the value of c should be chosen to be no less than approximately 1.386 (see above quotation from [Bentley 75]).

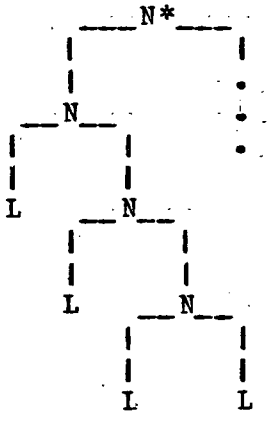
The first mechanism used to facilitate this involves the

use of a special internal node. Such a node, which we denote as "N*", differs from a regular internal node in the number of fields present and its effect on searching and insertion. Consider, for example, that we have a k-d tree that has become unbalanced in the sense that the above condition (1) has become true:



where "N" are internal nodes and "L" are leaf tuple blocks.

In this circumstance the special internal node N* is added to the front of the root node of the root page of the k-d tree as follows:



where "N" and "L" are defined as above.

No descriminator or key value fields are defined for the N* node. Furthermore, searching and insertion are affected by N* as follows:

(a) All searching recurses unconditionally down both the left and right subtree of N* and

(b) all insertions are made to the right subtree of

N* only.

Note that the operation of adding N* to a k-d tree primary index can be accomplished quickly and, once done, allows continuing use made of the left and/or right subtrees. Note also that N* could be inserted prior to any unbalanced subtree (a property that might possibly be used to advantage should we have balancing information pertaining to each individual internal node of a k-d tree).

The point of this operation is that it now allows an "idle process" (i.e. a certain utility program executing when the CPU is idle) to delete and re-insert the tuples of the left subtree of N* (the old unbalanced k-d tree). In this way the left subtree eventually disappears (including N*) and a new balanced k-d tree is the result.

Such a balancing process must ensure that randomness is introduced in the selection of tuples to delete and re-insert. Ideally the randomness should apply to the set of tuples comprising the base relation. Unfortunately, random deletion of tuples from a k-d tree results in an unacceptable space utilization of the leaf tuple blocks. There arises the possibility of an arbitrarily large percentage of the leaf blocks containing only a few tuples each.[1] We therefore introduce randomness in the selection of leaf tuple blocks

---

[1]Note that "rotations" used to ensure good space utilization in B-trees when deleting records cannot be used in the case of k-d trees.

from which tuples are deleted and re-inserted.

There are many ways in which this may be accomplished. One possibility is to introduce a hash file (possibly using Fagin's extendable hash file scheme) in which to store leaf tuple block addresses. That is, the address of each new leaf block arising from a leaf page split would be added to the hash file. A scan of the hash file in the physical storage sequence would then introduce the desired randomness in the selection of leaf tuple blocks. Now when N* is added (due to imbalance) this hash file is fixed (blocks under the right subtree of N* are added to another "new" hash file) and the sequential scan of the fixed hash file is made.

This scan proceeds as follows:

(a) A fixed number m first blocks (independent of the size of the base relation) are considered.

(b) Deletion and re-insertion of tuples from each of these m blocks is made in turn.

(c) Whenever one of the m blocks is exhausted the next block is retrieved from the hash file.

Some observations apparent with this approach are:

(a) The expected distribution of the range of values for <u>all</u> attributes of the tuples in the m blocks is randomly distributed (m parcels) throughout the value range of all attributes.

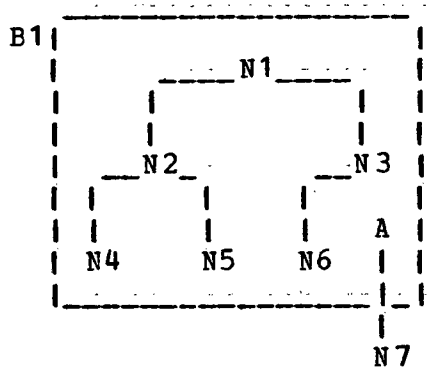
(b) At most m (a fixed small number) of blocks in the left subtree of N* can be poorly utilized.

The occurrence of imbalance in a k-d tree primary index does introduce a significant rebuilding phase. This rebalancing operation does, however, permit concurrent normal use of the k-d tree data structure. Furthmore, such a

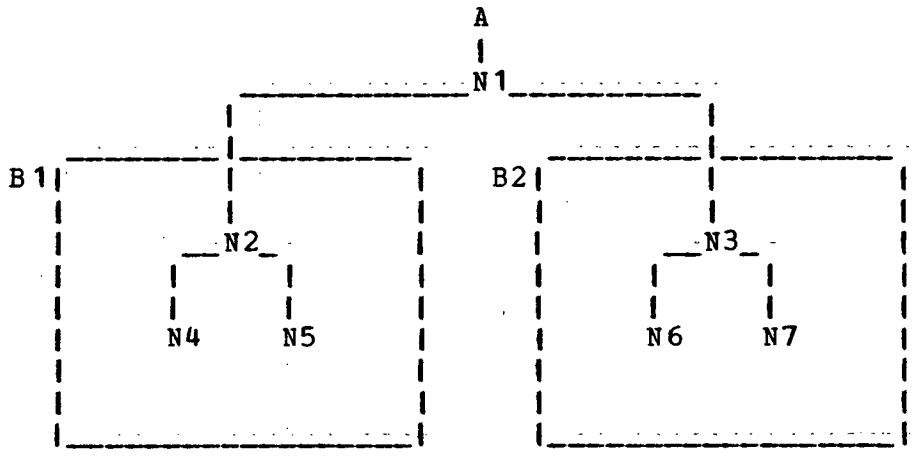
rebalancing operation is never absolutely necessary.

Consider now the third problem concerning the clustering of the internal nodes of a k-d tree themselves within fixed sized disk blocks. An approach is desired that ensures both good performance in terms of page faults that occur while chasing internal nodes and good disk block utilization. Furthermore, the approach should facilitate concurrent use of the primary index during its invocation and should be dynamic in the sense that no reformatting of an entire k-d tree is ever necessary.

The approach taken in SORAAM is very simple. Consider the following situation in which a primary index block with a maximum capacity of six internal nodes of a k-d tree is filled prior to the arrival of another internal node from below:



In this case a page split occurs creating another primary index block. As a result the internal node "N1" is sent one level above:



Note, therefore, that an internal node can arrive from below as the result of a leaf tuple block split or an internal node block split. In the above, the internal node "N1" may cause further internal node block splits above "B1" and "B2".

Because of the prior concern with balancing we are assured of a reasonable internal node disk block utilization (approximately 50 percent worst case). The obvious analogy with the B*-tree data structure also ensures good performance in terms of page fault rates. An exact match search on the k-d tree yields an expected number of disk reads given by:

$$\left\lceil 1 + \log_{\lceil \frac{IX}{PIB} + 1 \rceil} (IX+1) \right\rceil \qquad (2)$$

where $[IX/PIB]+1$ is the expected branching factor of each internal node block and (IX+1) is the number of leaf tuple blocks.[1] Note that allowing only one internal node per

---

[1]See Appendix A for a definition of "IX" and "PIB".

internal block (i.e. IX=PIB) yields an expected number of disk reads given by:

$$\left\lceil 1 + \log_{2}(IX+1) \right\rceil$$

Since we can expect at least 50 percent utilization of internal node blocks the following holds:

$$(2) \geq \left\lceil 1 + \log_{\frac{B}{2}}(IX+1) \right\rceil$$

where "B" is the internal node blocking factor.

The last problem concerning a k-d tree primary index that we deal with in this section concerns how a dynamic physical reorganization of the horizontal clustering necessitated by a changing characterization of use is facilitated. The problem is ccmprised of two parts:

(a) determining when a characterization of use has changed sufficiently to warrant physical reorganization and

(b) accomplishing the desired reorganization.

In SORAAM the characterization of use that determines the selection of discriminators during leaf tuple block splits are the "gnocld" and "fold" values found in the special base relations RELATIONS and ATTRIBUTES respectively (see Appendix A). These values therefore determine the currently existing horizontal clustering. Now recall from the start of this section that a characterization of use is dynamically maintained via the "qno" and "f" values found also in RELATIONS and ATTRIBUTES.

Determining a sufficient change between the "f" and "fold" (and similarly the "qno" and "qnoold") values to warrant physical reorganization (i.e. re-horizontally cluster; (a) above) is accomplished in SORAAM in a simple fashion. In the next section we derive a cost function based on weights derived from the above and on the "d" values (representing the number of levels in the k-d tree discriminating on each attribute[1]). Letting this function be denoted as "C" a sufficient change is deemed to have occurred by SORAAM when the following condition holds:

$$\frac{C(fold,qnoold)}{C(f,qno)} \geq c \qquad (3)$$

where "c" is an experimentally determined constant greater than 1. The test, then, is simple and overcomes the first part of our problem ((a) above). We note in passing that a more elaborate test (weighing the benefits of reorganization verses the reorganization cost itself; as in [Yao et al. 76]) necessarily involves the introduction of a "characterization of change of use". Such information is not currently maintained or used by SORAAM.

Accomplishing the desired reorganization ((b) above) is now simple. SORAAM simply reuses the mechanism used to maintain balancing in a k-d tree. That is, when condition (3) becomes true for a particular base relation:

---

[1]Note that the "d" values, themselves, are a function of the "qnoold" and "fold" values since discriminator selection determines the former and is determined by the latter.

(a) the special internal node "N*" is added,

(b) the "qnoold" and "fold" values are reset from the "qno" and "f" values respectively (for that base relation) and

(c) the rebalancing process is initiated.

Note from the above that discriminator selection for the right subtree of N* is now based on the current characterization of use. Furthermore, as with re-balancing, normal concurrent use of a base relation and the primary index is also facilitated.

We have seen in this section how SORAAM has automated one aspect of a RDMS's physical reorganizational requirements. Specifically, all management of horizontal clustering considerations including a characterization of use have been described. In the next section we deal with the specific problem of discriminator selection arising from leaf tuple block splitting.

C. Discriminator Selection

Recall that the primary indices in SORAAM grow dynamically in the number of internal nodes according to the size of the base relations. In particular, a new internal node is generated whenever an insertion operation forces a leaf tuple block split. One of the problems outlined in the previous section concerned the problem of selecting the attribute or column number on which to discriminate between the two leaf tuple blocks that arise when this event occurs.

Such discriminator selection should clearly be a function of our chosen characterization of use statistics (in particular: "fold" and "qnoold" values).

In SORAAM all discriminators occuring in internal nodes that are at the same level in the k-d tree are equal. We can therefore characterize the problem of discriminator selection as the problem of determining a sequence of discriminator values h such that cost functions describing:

P1. average external tuple page visits per query (most importantly) and

P2. average internal node visits per query (less importantly)

are, in some sense, minimized. In Figure III-3 in the preceeding section, for example, h = {h1,h2} = {column#1,column#2} = {1,2}. In the remainder of this section we outline two cost functions that characterize the first of these problems based on our chosen characterization of use. An identical closed form solution to problem P1 above is derived for both these cost functions and a method for choosing between the two is given. Following this we demonstrate that the two problems above are intimately related and derive a dynamic algorithm solving both problems that directly generates h. SORAAM's implementation of this algorithm completes the section.

Consider that the characterization of use statistics "fold" and "qnoold" of a particular base relation R <u>directly</u> reflect its query history. That is, all queries that have arrived against R have provided one and only one attribute

value. The probability of a particular attribute i being specified in a query is then given by:

$$p_i = \frac{fold_i}{qnoold}$$

Note that the directness assumption above implies:

$$\sum_{i=1}^{k} \frac{fold_i}{qnoold} = \sum_{i=1}^{k} p_i = 1 \quad \text{where k is the number of attributes in R.}$$

Letting $D \equiv |\{h\}|$ and $di \equiv |\{hj \mid i=j \text{ and } 1 \leq j \leq D\}|$ where $\{...\}$ denotes a set and $|\{...\}|$ denotes its size we may characterize P1 as:

$$\sum_{i=1}^{k} p_i 2^{D-di} \quad \text{where } \sum di=D \text{ and } 2^{D} \text{ is the number of leaf tuple blocks in R.} \qquad (4)$$

Taking out constants (assuming a fixed sized R) and letting wi=pi we derive a cost function characterizing external node visits:

$$F(d_i) = \sum_{i=1}^{k} w_i^{-di} \qquad (5)$$

subject to the constraint $\sum di=D$

In [Aho and Ullman 79] we are given an account of a derivation of a closed form expression yielding optimum distribution of discriminating bits (determining "bucket" addresses) among a base relation's attributes for partial

match retrieval. Their characterization of use, however, is based on the assumption that attributes are <u>independently</u> specified. The probability of a query specifying the first m of k attributes of R, therefore, would be:

$$p_1 \cdot p_2 \cdot \ldots \cdot p_m \cdot (1-p_{m+1}) \cdot \ldots \cdot (1-p_k)$$

or

$$\prod_{i=1}^{m} p_i \cdot \prod_{i=m+1}^{k} (1-p_i)$$

Similarly, their expression characterizing average bucket accesses (in our case external node visits) corresponding to (4) above is given by:

$$2^D P \prod_{i=1}^{k} \left(1 + \left[\frac{pi}{1-pi}\right] 2^{-di}\right) \qquad (6)$$

where $P = \prod_{i=1}^{k} (1-pi)$ and D, di and pi are defined as above.

Again taking out constants but now letting wi=pi/(1-pi) we have a cost function characterizing external node visits assuming independently specified attributes:

$$G(d_i) = \prod_{i=1}^{k} (1 + w_i 2^{-di}) \qquad (7)$$

subject to the constraint $\sum di=D$.

Using the method of Lagrangian multipliers to solve for the di in (5) above we have:

$$\frac{d}{dd_i}\left[\sum_{i=1}^{k} w_i 2^{-di} + \lambda\left[\sum_{i=1}^{k} d_i - D\right]\right] = 0 \quad \text{for } i=1,2,\ldots,k$$

or

$$-w_i (\ln 2)\, 2^{-di} + \lambda = 0$$

or

$$d_i = \log_2 w_i - c \quad \text{for some constant } c. \qquad (8)$$

The reader may check that (8) is also derived when we apply the same method above to solve for the d_i in (7). From [Aho and Ullman 79], then, (8) leads to the closed form:

$$d_i = \frac{D}{k} + \log_2 w_i + \frac{1}{k}\sum_{i=1}^{k} \log_2 w_i \qquad (9)$$

An optimal integer valued solution can now be derived from the above by a simple rounding process. The same reference demonstrates that the following proceedure yields such an optimal integer solution:

> "Add the fractional parts (which must sum to an integer) to determine how many d_i's must be rounded up. Select this number of d_i's having the largest fractional parts and round them up; round the others down."[1]

---

[1] From page 174 of [Aho and Ullman 79].

Constraining the di in the values they may assume by a lower and/or upper bound is also easily accommodated. In (4) and (6) above, for example, the di are constrained to be non-negative. [Aho and Ullman 79] give us the following procedure to deal with such constraints:

(a) From the application of (9) determine all i for which di is outside a bound.

(b) Set each such di to its nearest bound and then resolve the system with these di deleted.

(c) Should other di, as a result, "overstep a limit" then repeat from (b) above.

Note that accomplishing step (b) is simple since (9) implies that unconstrained di vary with a constant amount in a change of D.

We have now derived a solution to P1 in the static case of fixed D yielding a distribution of the discriminator values di among the attributes of a base relation. The discriminator value sequence h can now be generated from these di values. Furthermore, the average internal node visit cost (P2 above) can now be expressed as a function of the permutation of the di within h. In the following we derive a dynamic algorithm for k-d tree generation to optimally horizontally cluster tuples of a base relation according to either of the two interpretations of our characterization of use. This algorithm will be seen to minimize both P1 and P2 above.

To reiterate, the two interpretations of our characterization of use can be described as follows:

(a) if we map foldi/qnoold -> pi then pi -> wi we are assuming a <u>direct</u> representation of query history

and

(b) if we map -> pi as in (a) and pi/(1-pi) -> wi we are assuming an <u>independent</u> representation of query history.

Clearly the choice of mapping depends on which more closely relates the characterization of use with query history. The approach used in SORAAM to decide among these mappings is the following:

$$\underline{\text{if}} \sum_{i=1}^{k} \text{fold}_i > c \cdot \text{qnoold} \quad \text{(for some experimentally determined constant c)}$$

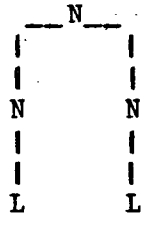
$$\underline{\text{then}}\ \text{choose}\ \frac{pi}{(1-pi)} \rightarrow w_i \quad \text{(independent model)}$$

$$\underline{\text{otherwise}}\ \text{choose}\ p_i \rightarrow w_i \quad \text{(direct model)}$$

Regardless of the choice of mappings the functions given in (4), (5), (6) and (7) above are all minimized at like distributions di of the discriminator values given by h. Consider now an alternative formulation of (4). Given a k-d tree of depth D with the discriminator value sequence (from root to leaves) h=h1,h2,...,hD we define an i-tree of the k-d tree ($1 \leq i \leq k$) as the subtree with the characteristic that all nodes with discriminator value i have one son only. Given the k-d tree in Figure III-3, for example, with D=2 and h={1,2} the 1-tree would be:

where "N" are internal nodes and "L" are leaf tuple blocks.

and the 2-tree would be:



where "N" and "L" are defined as above.

Now we can reformulate (4) as:

$$H_D(h) = \sum_{i=1}^{k} w_i L_{i\,D} \qquad (10)$$

where h, D, w_i are defined as above and $L_{i\,D}$ are the number of leaves in the i-trees (a function of h).

Likewise problem P2 can now be characterized as minimizing the cost function:

$$J_D(h) = \sum_{i=1}^{k} w_i M_{i\,D} \qquad (11)$$

where $M_{i\,D} = \sum_{j=1}^{D-1} L_j^i$ (i.e. the number of internal nodes of the i-tree)

and $M_1^i = 0$

We can relate (10) and (11) as follows:

$$J_D(h) = \sum_{i=1}^{k} w_i \left[\sum_{j=1}^{D-1} L_j^i \right] = \sum_{i=1}^{k} \sum_{j=1}^{D-1} w_i L_j^i$$

$$= \sum_{j=1}^{D-2} \sum_{i=1}^{k} w_i L_j^i + \sum_{i=1}^{k} w_i L_{D-1}^i$$

$$= J_{D-1}(h) + H_{D-1}(h) = \sum_{i=1}^{D-1} H_i(h) \qquad (12)$$

From (9) and its integer solution it is clear that a minimal $H_m(h)$ and $H'_{m-1}(h)$ (for $m \leq D$) have the property that there exists a unigue i such that:

(a) $d_j^m = d_j^{m-1}$ for $i \neq j$ and

(b) $d_j^m = d_j^{m-1} + 1$ for $i = j$

where $d_i^n = |\{h_j \mid i=j \text{ and } 1 \leq j \leq n\}|$

This now suggests how we can generate h to minimize both H and J above. Recalling that L_D^i is the number of leaves in the i-tree given by h (with depth D). This may be rewritten as:

$$L_D^i(h) = 2^{D-N_D^i(h)} \qquad \text{where } N_m^n(h) \text{ is defined as the number of } h_i \ (i<m) \text{ such that } h_i = n.$$

The approach involves a judicious choice of a summand i in (10) whenever the depth of the k-d tree increases by one. We simply choose the largest summand i. This approach ensures that for all $1 \le m \le D$ the following holds:

$$w_j 2^{m-N_m^j} \le w_{hm} 2^{m-N_m^{hm}} \qquad \text{for } 1 \le j \le k$$

or

$$\log_2 w_j - N_m^j \le \log_2 w_{hm} - N_m^{hm} \qquad (13)$$

Note that an optimal h minimizing H and J may not be unique since, for example, there may be more than one choice for hD (i.e. at the lowest level) satisfying (13). By convention SORAAM always chooses the minimal column number for hm. Specifically, when a leaf tuple block at depth m of base relation R splits SORAAM behaves as follows:

(a) The set S of constrained discriminator values j ($1 \leq j \leq k$) is determined as those j such that:

$$2^{N^{j}_{m}+1} < c \cdot n$$ (for an experimentally determined constant c)

where n, found in ATTRIBUTES, is the number of distinct values of the attribute with {Aname=R & COL#=j}.

(b) The discriminator value i is chosen as the minimal i such that:

$$\log_2 w_j - N^{j}_{m} \leq \log_2 w_i - N^{i}_{m}$$

for all i,j not in S.

(c) N^{j}_{m+1} is assigned N^{j}_{m} for ($1 \leq j \leq k$) and ($j \neq i$) and

N^{i}_{m+1} is assigned $N^{i}_{m} + 1$.

(d) If m=D then both D found in RELATIONS with {Rname=R} and d found in ATTRIBUTES with {Rname=R & COL#=i} are incremented.

The attributes n, d and D used above are located in the special base relations RELATIONS and ATTRIBUTES (see Appendix A). The N values are represented as a single k-vector K in each leaf tuple block as follows:

$$K_i = N^{i}_{m} \quad \text{where} \quad \sum_{i=1}^{k} K_i = m$$

Step (c) above, therefore, is accomplished by simply incrementing Ki. Lastly, the weights w used in step (b) above are determined by the appropriate mapping from the "fold" and

"qnoold" values found also in the special base relations.

According to two fundamentally different interpretations of our characterization of use, therefore, we have derived a dynamic algorithm for discriminator selection in primary indices that realizes an optimal horizontal clustering of the tuples within base relations.

D. Secondary Indexing

This section deals primarily with the automatic selection of single attribute inversions on base relations in SORAAM. Such inversions are called secondary indices and are desirable for their potential of facilitating more efficient search within base relations.[1]

We shall see that automating selection of secondary indices, however, requires that SORAAM have an intimate knowledge of both the physical and logical aspects of such indices. For this reason we begin the section with a detailed presentation of the variety of secondary indexing currently considered by SORAAM. Consideration is then given of a variety of balancing constraints used to control the indexing. The control parameters adopted by SORAAM for this purpose are then presented followed by the index selection details themselves.

---

[1] See in Chapter II, Section C our definition of secondary indexing.

The section concludes with a more general consideration of the notion of secondary indexing.

Figure III-5 below presents the example base relation "PJ" represented in terms of its primary index, leaf tuple blocks and single secondary index on the second column attribute "J#".[1] The two internal nodes of the primary index have the form "discriminator#:key value". The tuples in the leaf tuple blocks of Figure III-5 (b) are addressed by "tuple identifiers" or "tids" of the form tij where i and j index tuple blocks and tuples within blocks respectively.

In SORAAM a secondary index may exist on any of the three attributes P#, J# and QTY of PJ but not, for example, on a concatenation of P#+QTY. Furthermore, any such index is an inversion (not a multilink or multiring).

Inversions are actually comprised of two components. The first is a B-tree index with entries of the form {v,p} where v is a value of the indexed attribute occurring in the base relation and p is a pointer to a physically sequential list of tids of tuples having that value. The second component of an inversion is the set of secondary tuple pointer blocks (STPBs) in which the tid lists themselves are stored. In Figure III-5 (c) four such STPBs are used to contain the four tid lists. Furthermore, all tids in each tid list are maintained in

---

[1] We may consider that PJ is interpreted by a contracting firm as representing the quantity ("QTY") of each part ("P#") supplied by the firm to each job ("J#").

Example Base Relation With
Primary and Secondary Indexing

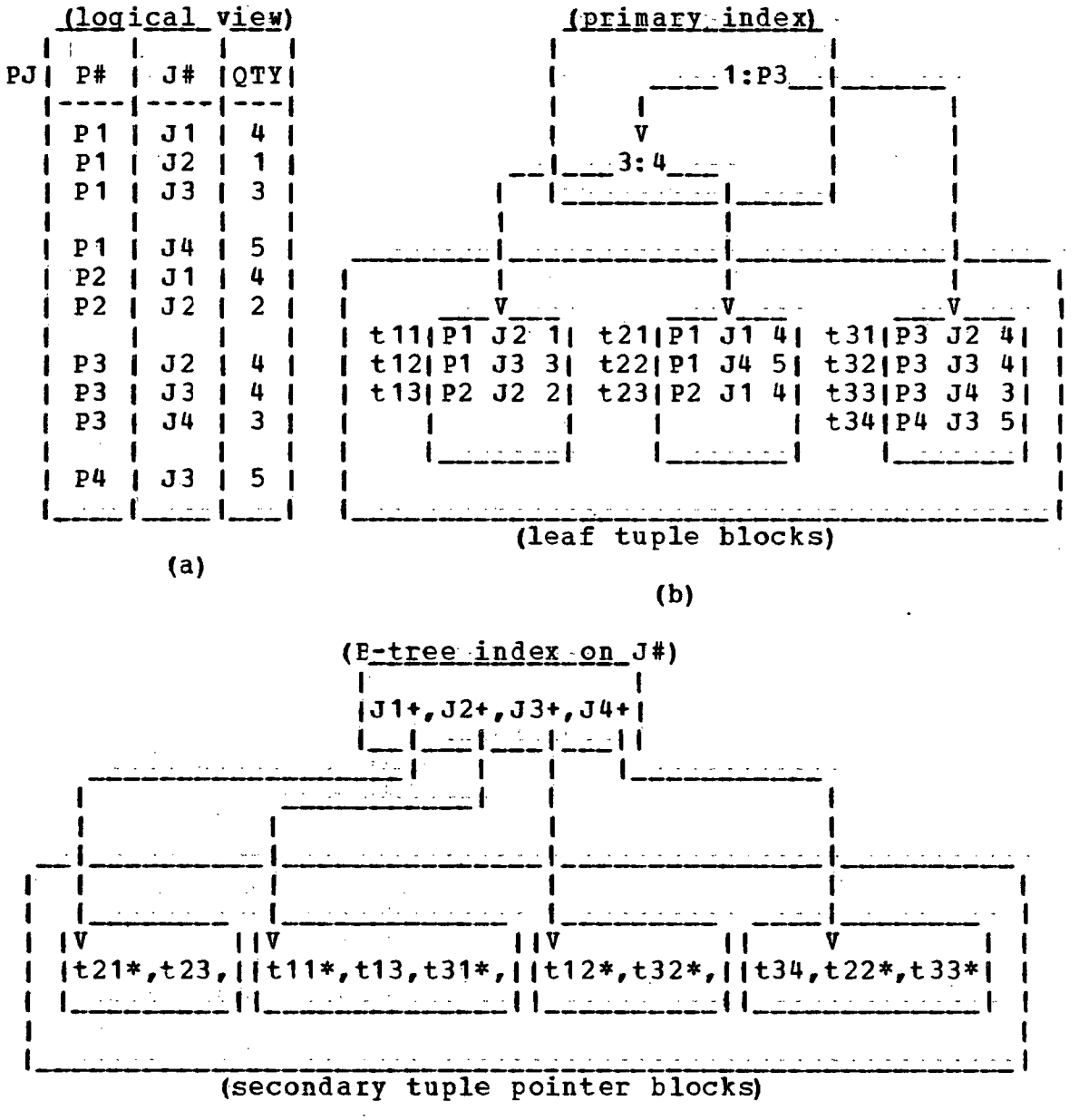


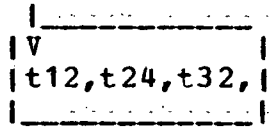
Figure III-5

sorted order.

A scan on PJ via the J# inversion would produce the sequence of tids given by their left to right sequence in

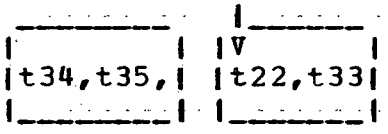
Figure III-5 (c). Notice that such a scan visits each STPB only once. When retrieving the tuple associated with a tid it may be necessary to retrieve the tuple block, in which it resides, from mass storage. Such an event is a function of the relevence to the scanned attribute of the horizontal clustering, of the size of the tid lists themselves and of the number of main memory buffers available for the operation. Assuming that one buffer only is available to contain a leaf tuple block then scanning PJ via the J# inversion would produce seven tuple block reads. Those tids in Figure III-5 (c) causing such reads are postfixed with "*". In the next section on search strategy we elaborate on estimating such costs.

If an insertion or deletion occurs in a base relation then additional effort is required to suitably alter any secondary indices on the base relation's attributes. Consider, for example, an insertion into PJ of the tuple T={P2,J3,6}. According to the primary index T would be inserted into the second leaf tuple block and assigned the tid t24. Since the value "J3" already exists in the B-tree then updating the secondary index simply involves adding t24 to J3's tid list. Another entry in the third STPB would then ensue. The resulting STPB would appear as:



New STPBs are created whenever tid insertions cause overflows.

Should the maximum capacity of an STPB be three tids and should T's assigned tid have been t35 then the fourth STPB would split creating a new STPB in order to accommodate t35 as follows:



Note from the above that J4's tid list pointer must also be updated.

Further details concerning our variety of secondary indexing can now be simply deduced. It should be clear, for example, that the STPBs for a particular index be doubly linked to one another in the obvious manner. Furthermore, the splitting approach to STPB creation implies a worst case of 50 percent space utilization. However, the expected case space utilization is much better and is very likely uniform for all secondary indices of any base relation. For each base relation this expected case utilization is represented as a fractional value of the attribute "ff2" in RELATIONS (see Appendix A).

The reason for our concern with making so explicit the architecture of permitted secondary indexing is that deciding which secondary indexing to create and delete is intimately related to search strategy (next section) which in turn is intimately concerned with cost estimation of query evaluation. This cycle is illustrated by Figure III-6 below with arrows denoting the dependency.

The next thing to consider in automating the selection of

A Cyclic Dependency In Secondary Index Selection

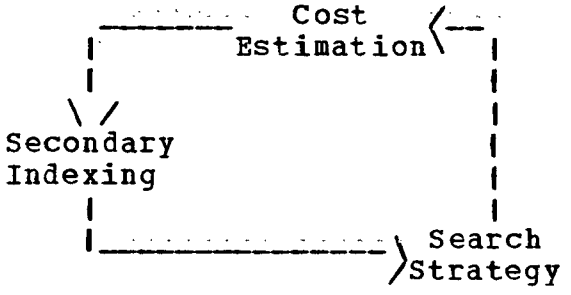


Figure III-6

the variety of indexing discussed above is the specific control mechanism with which decisions about the choice of indexing are made. Suppose, for example, that we adopt a design constraint that requires complete isolation of all index selection considerations from SORAAM's users. This might be accommodated by a method based on any of the following:

(a) All emphasis is placed on retrieval performance only. That is, no constraint exists in selecting indices. In this case all base relations would exist fully inverted (i.e. with an index on each attribute).

(b) All emphasis is placed on minimizing total I/O traffic to mass storage. In this case the balancing constraint to index selection is the increased I/O implied by maintenance operations. A secondary index reduces total block reads for retrieval evaluation and increases block reads and writes for insertions, updates and deletions.

(c) A global space/time tradeoff is assumed by SORAAM. In this case a selection of the most advantagous subset of the possible inversions fitting within the permitted space overhead of a base relation is made.

Should all base relations be fully inverted then the problem of index selection is trivial. Both the space and

potentially large maintenance overhead of this approach, however, are clearly unacceptable in the environment assumed by SORAAM. Consider, for example, a base relation implementing a spooling or auditing function. The operations on such a relation may almost entirely involve insertions and deletions. Alternatively, in using maintenance I/O as a balancing constraint a tradeoff is assumed between query and maintenance performance. SORAAM would unconditionally weight a block read or write caused by an insertion equal to a block read caused by a retrieval. There are clearly applications for which such a tradeoff would be entirely inappropriate. Consider, for example, an on-line critical patient information system for a hospital. One can understand that retrieval performance would rank far above maintenance performance for base relations involved in such an application. This same criticism also applies in a more obvious manner to an approach involving a global space/time tradeoff assumption. The retrieval performance requirement for one base relation can entirely differ frcm another.

For these reasons a design constraint of complete isolation of all index selection considerations within SORAAM has not been adopted. Instead an attempt is made to render the necessary user communication concerning index selection in as simple a form as possible.

Towards this end SORAAM maintains for each base relation a measure of the indexing effectiveness. The global performance measure (g.p.m.) is simply a ratio of the number

of relevant block reads (i.e. a disk read of a leaf tuple block containing a desired tuple) to total block reads (including primary index blocks, B-tree blocks and STPBs) for past retrieval operations. The attributes "g.p.m.", "RDQreads" and "DQreads" in RELATIONS exist for this purpose (see Appendix A). In the same way as "radix1" can be used to control the rate of adaptation of the characterization of use statistics[1] the attribute "radix2" can be used to similarly control the g.p.m.

The control over secondary indexing for each base relation is now accommodated by the use of the following three user supplied parameters:

(a) Plow, Phigh - These are a lower and upper bound respectively on g.p.m. (the retrieval performance). Increasing Plow would therefore tend to create further indexing while decreasing Phigh would tend to delete current indexing. For each base relation SORAAM attempts to ensure that the following condition holds:

$$\text{Plow} \leq \text{g.p.m.} = \frac{\text{RDQreads}}{\text{DQreads}} \leq \text{Phigh}$$

(b) S - This parameter conveys the maximum percentage of indexing storage overhead permitted for the base relation. All blocks used by the internal nodes of the primary index and by secondary indices (both the B-tree nodes and STPBs) are regarded as storage overhead.

All three of the Plow, Phigh and S are also located in RELATIONS.

For each base relation the latter parameter S is assumed

---

[1] See beginning of Chapter III Section B.

to have precedence over Plow. That is, should the user implied minimum retrieval performance imply an indexing set (i.e. set of single attribute inversions for a base relation) exceeding the permitted storage overhead implied by S then SORAAM derives an alternative indexing set best utilizing the permitted storage. Also, should Plow exceed a certain threshold value then a full inversion of the base relation may ensue. Setting Plow to a high value then allows a direct control of a space/time tradeoff via S by a user.

In addition to g.p.m. SORAAM automatically maintains a vector b on each base relation with elements bi (i>0) of the relative number of times each possible secondary index would have been used should it have always existed. Note that i denotes a secondary index on the attribute with {COL#=i}. With respect to this vector another statistic "Pcost" is also maintained. Pcost, however, denotes a value comparable to that of DQreads with the assumption that no indexing has ever existed.

The value of Pcost is increased by an estimate in block reads of the cost for each <predicate> instance evaluation via the primary index. Furthermore, should the maximum depth of the primary index increase by one then Pcost is also incremented. In this case the discriminator COL# for the new level determines the increment. In particular, the increase in primary index and leaf tuple block visits doubles for the fraction of queries not supplying values for the new discriminating COL#. That is:

$$Pcost \leftarrow Pcost + \frac{qnoold - fold_i}{qnoold} \cdot Pcost$$

These statistic are primarily maintained by a component of the search strategies (see Section E of this chapter) which is invoked for each such <predicate> instance submitted to SORAAM. Vector b is stored as values of the attribute "b" in ATTRIBUTES; Pcost is found in RELATIONS.

The specific search strategy for evaluating any individual operation required of SORAAM always involves the use of a single index only (be it primary or secondary). As a result of this the bi are independent in their implication of index applicability to query evaluation; that is, the sum bi+bj (i¬=j) makes sense. SORAAM's approach to index selection, therefore, is to apply approximate algorithms for an appropriately formulated knapsack problem.[1] In light of this we define the following benefit vector:

$$b'_i \equiv b_i \cdot \max\{0, \lceil \text{block reads if index i doesn't exist} \rceil - \lceil \text{block reads if index i does exist} \rceil\}$$

(A) (B)

$$\text{where (A)} = \frac{2^{D-di}}{2^D} \cdot \{(IX+1) + PIB\} = \frac{(IX+PIB+1)}{2^{di}}$$

(i.e. a fraction of tuple blocks + index pages of k-d tree)

---

[1] A family of such algorithms may be found in [Sahni 75].

$$\text{and } (B) = \left\lceil \log_{\left\lfloor \frac{n_i}{SIB_i} \right\rfloor + 1} n_i \right\rceil + \frac{STPB}{n_i} + \min\left\{ \frac{N}{n_i}, 2^{D-di} \right\} \qquad (14)$$

(i.e. index block reads + STPB reads + tuple block reads)

In the above the values of D, di and IX have already been introduced in previous sections. The other values PIB, ni, SIBi, STPB and N are maintained in RELATIONS and ATTRIBUTES. The reader is refered to Appendix A for a description of each.

The benefit vector given by (14) illustrates the requirement for intimate knowledge of the architecture of the indexing by index selection components. Cost estimation in block reads of using the various possible access paths clearly requires such knowledge. The third component of the cost estimation for (B) above, for example, incorporates the possible effect of horizontal clustering by the primary index.

A cost vector c is defined in terms of the number of blocks necessary for each index. More specifically:

$$c_i \equiv \left\lceil \text{\# of index blocks and STPBs necessary for index } i \right\rceil$$

$$\equiv SIB_i + STPB$$

Note that c is defined in terms of storage cost only. Should one assume that maintenance I/O overhead is uniform over any indexing set then such a cost function indirectly incorporates maintenance cost.

We now wish to calculate a threshold performance requirement P for a new indexing set. Denoting the current indexing set as a vector x' where:

$$x'_i = \begin{cases} 1 \text{ if COL\#=i currently indexed}^1 \\ 0 \text{ otherwise} \end{cases}$$

for i varying over all possible COL# in the base relation of concern then the following represents another estimate of a value for DQreads (i.e. block reads for past retrieval operations) relative to an indexing set:

$$Pcost - (b' \cdot x')$$

Allowing Plow = RDQreads/y we have:

$$\frac{Pcost - (b' \cdot x')}{DQreads} = \frac{Pcost - P}{y}$$

Now solving for P we have:

$$P = Pcost - \frac{RDQreads}{Plow} \cdot \frac{Pcost - (b' \cdot x^0)}{DQreads}$$

$$= Pcost - \frac{g.p.m.}{Plow} \cdot \{Pcost - (b' \cdot x')\}$$

We can now formally specify the approach used by SORAAM to determine an indexing set x for a base relation. Whenever either of the following two conditions occur:

(a) g.p.m. < Plow (i.e. not enough indexing)

---

[1] Iflag is true in this case (see Appendix A).

(b) g.p.m. > Phigh (i.e. too much indexing)

then SORAAM solves the following knapsack problem:

$$\text{minimize } c \cdot x \qquad (15)$$

$$\text{subject to } b' \cdot x \geq P \quad \text{where } x_i \text{ in } \{0,1\}$$

and, similarly, whenever either of the following occur:[1]

(c) (c . x) + PIB > (S . [IX+1])

(i.e. above solution (15) requires too much storage overhead)

(d) (c . x') + PIB > (S . [IX+1])

(i.e. current indexing set requires too much storage overhead)

then the problem becomes:

$$\text{maximize } b' \cdot x \qquad (16)$$

$$\text{subject to } c \cdot x \leq (S \cdot [IX+1]) - PIB$$

The chosen indexing set corresponds to all xi=1 in x.

Thus we see how SORAAM automates the selection of single-attribute inversions thereby accomplishing automation of another major physical organization detail. There are a number of ways in which these results may be extended. The most obvious is to add to the choice of indexing data structures available to SORAAM. Perhaps minimally one should consider hashing and TRIE techniques and even various other static searching structures. Linking data structures such as the link access path in System R are also clearly desirable.

---

[1]Note that in both conditions the cost of the primary index is included.

Consider a situation in which the constraints of an aspect of a world model dictate a very regular distribution of values on a specific attribute of a relation. Consider also that the model dictates that the attribute is a key of the relation. The obvious representation of an index for this "nice" attribute would simply be a fixed size FORTRAN-like array. Furthermore, such a situation is not so unlikely. The "entity domains" first presented in [Hall et al. 76] and adopted in [Codd 79] are a possible example. Values in entity domains are system generated and unseen and unknown to end users.

One may even consider a design decision to not commit SORAAM to a fixed small number of searching data structures and search algorithms. SORAAM would facilitate the easy extension of both its secondary indexing capabilities and its user interface (i.e. augmenting the variety of non-root node types). Justification for such a decision seems all the more clear when one considers the existence of searching data structure transformations as in [Bentley 78c].

Accommodating additional indexing data structures is relatively simple as long as we maintain a constraint that no more than one be used to evaluate any given instance of a <predicate>; as long as problems (15) and (16) are well defined. Should n different varieties of indexing data structures be available then for a base relation of k columns the benefit vector would be of size nk.

The formulation of the problem of index selection given

by (15) and (16), however, seems to fail when arbitrary combinations of indices are permitted in <predicate> evaluation. The results also fail to extend nicely when inversions based on a concatenation of more than one attribute are permitted to exist. In such situations the values c . x in (15) and b'. x in (16) no longer make sense.

We note in concluding this section that a specific multi-attribute inversion may indeed be implied by statistics already maintained by SORAAM. We conjecture the implication of such an inversion in situations where the "qno" value of a specific column i greatly exceeds its b value. This signals that queries supplying values for column i almost always do so in combination with at least one other attribute j whose index is prefered by the search strategies over an index on i in evaluating such queries.

E. Search Strategies

This final section of Chapter III concerns the decision process for the selection of access paths to be used in the searching components of operations on SORAAM. Fundamental to this selection process is a cost estimate of the number of block reads that would be incurred for each index available. The first part of the section presents the method used by SORAAM to determine such costs and the implied search strategy. The section ends with a discussion of related work

concerning the RDMSs System R and INGRES.

One of the simplifying assumptions adopted by SORAAM is that the use of one index only is permitted in the evaluation of any <predicate> instance.[1] Such an assumption greatly simplifies the selection of access path and, as we have seen in the previous section, the problem of automating index selection.

Given a query, therefore, (defined as a <predicate> instance) the problem is to determine which of the currently existing secondary indices or the k-d tree primary index to use for the search. Ietting k be the number of columns in the base relation and I subset {0,1,...,k} represent the indexing set (where i>0 in I implies the attribute with COL#=i has a secondary index and where i=0 denotes the primary index) then the problem is to determine the j in I yielding the least block reads in the search implied by the query. We now outline the procedure used by SORAAM to handle this problem.

Recalling that all queries are in disjunctive normal form we consider this procedure in two parts. The first, P1, concerns each <conjunct> (a partial match or partial range query; see Table II-1 for illustration) and the second, P2, the query as a whole.

For each <conjunct> P1 returns to P2 a list of the form:

---

[1]Throughout this section we will be assuming familiarity with the design of the user interface outlined in Section A of this chapter.

$$L \equiv \{cost0, cost1, \ldots, costk\} \qquad (17)$$

denoting estimated cost in block reads to evaluate the query using the primary index (cost0), a secondary index on the first column (cost1), on the second (cost2), etc. If i is not in I then costi is assigned a large number which we denote as LARGENUMBER. Otherwise for each i in I P1 does the following:

1. cost$_i$ <- LARGENUMBER

2. for each <primitive> in the <conjunct> do

   2.1 if COL# ¬= i {where COL# is implied by the "Aname" field of <primitive>}
   then repeat from 2 for next <primitive>.

   2.2 determine the selectivity factor SF

   2.3 temp <- SF . (*)

   2.4 if temp < cost$_i$ then cost$_i$ <- temp

   2.5 repeat from 2 for next <primitive>

3. return.

An introduction to and definition of SF (a selectivity factor) closely qualifying for our own purposes is given in [Selinger et al. 79]. Basically, SF is an attempt at estimating the fraction of tuples or blocks that would be accessed should an index i be used with an appropriate <primitive>. This estimate, a function of the <primitive> and of information regarding the indexed attribute and maintained by SORAAM, is calculated as follows:

(a) For a non-root node type 13 <primitive>:

$$SF \leftarrow \frac{high - low}{max_i - min_i}$$

where max_i and min_i are boundary values of the attibute with COL#=i and are located in ATTRIBUTES.

(b) For a non-root node type 12 <primitive> if the comparison operator is one of {<,<=} then:

$$SF \leftarrow \frac{value - min_i}{max_i - min_i}$$

or if it is one of {>,>=} then:

$$SF \leftarrow \frac{max_i - value}{max_i - min_i}$$

else if the comparison operator is "¬=" or "=" then:

$$SF \leftarrow 1 \quad \text{or} \quad SF \leftarrow \frac{1}{n_i}$$

respectively.

Notice from the above that we are assuming the validity of the minus operator for range <primitive>s. There are, of course, some attributes for which only the "=" and "¬=" comparison operators would be valid. Notice also that the estimates assume a uniform distribution of both index values and tuples on values.

Step 2.3 sets the estimate "temp" of block reads in using index i with <primitive> as the product of SF and (*). (*), therefore, is an estimate of the cost in block reads of a scan

of the entire base relation via index i. In SORAAM this estimate is a function of:

(a) the number of buffers available to hold index and tuple blocks read from mass storage (this is given by the value of the attribute "buf" in ATTRIBUTES),

(b) the nature of the horizontal clustering on the base relation and

(c) the availability of various scanning algorithms on the STPBs of the secondary index.

There are four STPB scanning algorithms considered by SORAAM. All vary in their implied CPU overhead and minimum number of required buffers. These algorithms can be described as follows:

SCAN1. A sequential scan of the STPBs.

SCAN2. An aquisition and sort of all relevant tids.

SCAN3. An aquisition and sort of n STPB blocks of tids at a time.

SCAN4. Merge of m tid lists.

Assuming we have j buffers available then devoting all but two of them to buffering leaf tuple blocks suggests the first scan algorithm SCAN1 above. This assumes that the number of levels is the primary index discriminating on the relevant index (di) is large enough. That is, di must partition the set of leaf tuple blocks into units small enough such that each may be entirely contained within the available main memory buffers.

Should this not be the case then we may still consider intelligent use of the available buffers for buffering index and STPB blocks themselves. Since tids are almost always much

smaller in size than tuples a much smaller number of buffers may only be required. SCAN2, therefore, is a simple first step in this direction. In this case we require enough buffer storage to contain all STPB blocks containing all relevant tids (i.e. tids pointing to tuples satisfying the query <predicate>).

SCAN1 took advantage of the partitioning of the set of leaf tuple blocks by di. Consider now that this same partitioning must necessarily apply also to the set of index and STPB blocks. Each subset of index or STPB blocks resulting in this partitioning is associated with its own disjoint subset of leaf tuple blocks. Both SCAN3 and SCAN4 may take advantage of this phenominon. We may now concern ourselves with acquiring and sorting the tids for each element of the STPB partition in the case of SCAN3 or merging the tid lists for each element of the index block partition in the case of SCAN4. Because of the lack of knowledge of the exact location of these partitions in the case of secondary indices we can expect that boundary overlap will roughly double the scan costs of these approaches over that implied by the use of SCAN1.[1]

Finally we note that should the range of index values be sufficiently small then SCAN4 may also apply. In this case an

---

[1]If we had adopted the overhead of marking these index and STPB partitions in the design of our secondary indices (i.e. marked index values occurring as values in the primary index) then this doubling of scan cost over best possible would not occur.

m-way merge of the tid lists is performed where m is the number of index values occurring in the range.

Table III-2 below summarizes this variety of scan algorithm in terms of the minimum number of buffers required and the resulting cost estimate (*) in block reads for a scan of the tuples in a base relation. The use of this table by SORAAM in calculating (*) proceeds as follows:

(a) For each approach for which enough buffers are available the estimated scan cost is computed.

(b) From these the approach involving the least disk I/O and then the least CPU overhead is selected and (*) is assigned its calculated scan cost.

The approach selection implied by step (b) above is consistent with the design emphasis in SORAAM on retrieval response. With respect to CPU overhead, if we assume sequential scans are cheaper that merges which in turn are cheaper than sorts, then the ordering of approach selection is implied by the top to bottom ordering of Table III-2.

We have now seen how costi (i>0) is calculated by P1 (the first part of our search strategies procedure). In estimating a cost for the primary index (cost0) SORAAM abbreviates the representation of a <conjunct> as a partial match query Q subset {1,..,k} where i in Q implies that an "Aname" in a <primitive> in the <conjunct> exists such that the COL# associated with Aname is i. Using this representation for the <conjunct> cost0 is calculated by P1 as follows:

Scan Cost of a Base Relation Via a Secondary Index

Approach	min. necessary # of "buf"fers	Scan Algorithm	Cost in Block Reads
(A)	$\frac{IX + 1}{2^{di}} + 2$	SCAN1	Best[1]
(B)	$\lceil SF.n_i \rceil + 2$	SCAN4 ($m=\lceil SF.n_i \rceil$)	Best[1]
(C)	$\lceil SF.STPB \rceil + 2$	SCAN2	Best[1]
(D)	$\left\lceil \frac{n_i}{2^{di-1}} \right\rceil + 2$	SCAN4 ($m=\left\lceil \frac{n_i}{2^{di-1}} \right\rceil$)	2.Best[1]
(E)	$\left\lceil \frac{STPB}{2^{di-1}} \right\rceil + 2$	SCAN3 ($n=\left\lceil \frac{STPB}{2^{di-1}} \right\rceil$)	2.Best[1]
(F)	3	SCAN1	Default[2]

[1] "Best" is defined as $(SIB_i + STPB + IX + 1)$; i.e. no index, STPB or tuple block accessed more than once.

[2] "Default" is defined as $SIB_i + STPB + n_i \cdot \min\left(\left\lceil \frac{N}{n_i}, 2^{D-di} \right\rceil\right)$. The worst case here occurs when each tuple reference by its tid requires a block read. The third term takes into account the effect of sorted tuple lists. This effect is felt as 2^{di} approaches n_i.

Table III-2

$$\text{cost0} \leftarrow \frac{1}{\left(\sum_{i \text{ in } Q} di\right)^{2}} \cdot \{PIB + IX + 1\} \qquad (18)$$

This abbreviation of <conjunct> format in estimating cost0 is justified when one considers that range <primitive>s can only improve the performance of the primary index. That is, the ratio of relevant leaf tuple block reads to total block reads (a true measure of index performance) can only improve. Note that cost0 need only be evaluated once for any predicate instance.

Now consider that the second part of our index selection procedure, P2, has for each <conjunct> of the query <predicate> a list L given by (17) denoting estimated costs to evaluate the <conjunct> for each possible index. The final index selection by P2 is determined as follows:

(a) Any index i for which costi for any <conjunct>=LARGENUMBER is disqualified from consideration. Note that, by this criterion, the primary index is never disqualified.

(b) Of those indices remaining the index i chosen has the minimum of the values:

$$\text{cost}_0 \quad \text{or} \quad \sum_{\text{<conjunct>}} \text{cost}_i \quad \text{for } i>0.$$

This completes the outline of cost estimation and index selection used by SORAAM. Notice that the query evaluation facilitated has (admittedly not gracefully) finessed any consideration or necessity of performing intersections and

unions of tid lists. This immediately follows from the restriction stated at the beginning of this section of permitting one index only in the evaluation of any <predicate> instance.

Following the determination of the actual index to use for the given predicate instance SORAAM reuses the preceeding procedures to determine the most desirable index (whether or not it currently exists) for this purpose. Should a secondary index be chosen in this case then the associated bi value in ATTRIBUTES is incremented. Furthermore, the "Pcost" value associated with the implied base relation is incremented by the estimated value of cost0; by the value given in (18) above. In this way the search strategies maintains the necessary information for use by the secondary index selection proceedures outlined in the preceeding section.

To illustrate the effect of SORAAM's horizontal clustering together with its various base relation scan algorithms consider the case of a particular base relation schema R(a1,a2,...,ak) where R and ai represent the relation and attribute names respectively. Assume that R contains 120,000 tuples in 4,000 leaf tuple blocks at 30 tuples per block. Assume also that secondary indices exist on the first two attributes only and that:

(a) D=12; d1=6; d2=6 - implying that access to R is exclusively via a1 and a2 and that <predicate>s supply values for a1 as often as for a2.

(b) Psize=1000; ff2=2/3 - implying that the number of STPBs per index is 180.

Approach (E) using SCAN3 in Table III-2 now tells us that with just 8 buffers a scan of R via the secondary index on a1 or a2 yields an expected 8,000 leaf tuple block reads! A mean cost in leaf tuple block reads for the complete scan of R via any secondary index is therefore 8,000 (again assuming just 8 buffers available). This compares very favourably with scan costs for secondary indices of all RDMSs reviewed in this thesis that support them. Consider that [Selinger et al. 79] predict a comparable cost for a base relation scan via unclustered images in System R as the cardinality of the base relation. A mean cost of 120,000 tuple block reads therefore arises even with the availability of hundreds of buffers. In this case SORAAM exhibits an order of magnitude decrease in mass storage I/O.

The search strategies component of SORAAM accomplishes only a simple part of the more general task of compiling best possible deterministic execution strategies from non-procedural DMLs. That is, the kind of searching implied by a non-procedural DML of an RDMS is much more complex than that supported by SORAAM. Aggregation operators such as MAX or AVG and data relatability operators such the the JOIN operator of the relational algebra contribute enormously to the difficulty of the problem.

Literature dealing with the subject of this section, therefore, usually concerns the more general setting. In System R the "optimizer", a component of the RDS (the relational language level of System R), has full

responsibility for all searching strategy considerations. This includes searching within and between base relations. [Blasgen and Eswaran 77] and more recently [Selinger et al. 79] present the details of System R's optimizer.

In SORAAM it is necessary to split the responsibility of search strategy selection between the access path and relational language levels. SORAAM's commitment to controlling all clustering and index creation details implies that it must also be responsible for search strategy within base relations. That is, since SORAAM can only know of the existence of a secondary index at a given point in time then SORAAM must decide when such an index is to be used.

SORAAM, however, does not represent a precedent in this regard. The DML of INGRES called QUEL is based on the relational calculus. In QUEL the module responsible for optimizing one-variable queries is distinct from the module responsible for optimizing multi-variable queries. The searching capability of the former (called OVQP[1]) is analogous to that provided by SORAAM. For an interesting outline of the approach used by INGRES in handling multi-variable queries see [Wong and Youssefi 76].

Query processing in SORAAM is distinct from OVQP of INGRES and the optimizer of the RDS of System R with respect to the more complex degree of clustering supported. We have

---

[1]See the review of INGRES in Section B of Chapter II.

seen in a preceeding example that enormous reductions in I/O traffic can ensue from support for multidimensional horizontal clustering in performing search within base relations.

Chapter IV: Future Directions and Conclusions

A. The Next Step

In this first section of the final chapter we discuss what may be considered the next logical step in the study of a self-organizing access path level: a demonstration of feasibility of the concept of SORAAM. This necessarily requires further empirical evidence. The nature of this evidence and the testing environment necessary to acquire it are discussed.

A number of "experimentally determined constants" (or tuning parameters) were introduced throughout Chapter III. A test system in which to empirically determine suitable values for these constants is clearly necessary. To reitterate, the problems for which solutions involved the use of these constants can be outlined as follows:

(a) the detection of a suitable imbalance condition in the primary index of a base relation. In Section B of Chapter III imbalance was determined by the value of the condition:

$$D \geq c1 \cdot \log_2 (IX+1)$$

Furthermore, we saw that the real valued constant c1 was constrained to be greater than 1.386.

(b) the introduction of randomness in the selection of leaf tuple blocks from which tuples are deleted and re-inserted according to the balancing or re-clustering phase of a primary index. Also in Section B of Chapter III an integer constant number c2 of leaf tuple blocks are selected at a time.

(c) The determination of a sufficient change in the

characterization of use to warrant re-clustering of a base relation. Such re-clustering (Section B; Chapter III) was determined by the value of the condition:

$$\frac{C(fold, qnoold)}{C(f, qno)} \geq c3$$

where C is defined as the value of the functions F or G (see Section C; Chapter III) and c3 is a real valued constant greater than 1.

(d) the interpretation of the characterization of use statistics for the purpose of discriminator selection in the primary index. In Section C of Chapter III the choice for the direct verses indepentent models was based on the following condition:

$$\sum_{i=1}^{k} fold_i > c4 \cdot qnoold$$

where c4 is again a real valued constant greater than 1.

(e) the acceptance of a minimal number of unique existing attribute values for qualification of discrimination in the primary index. In Section C of Chapter III we permitted discrimination on an attribute in a leaf tuple block split only if the following condition was satisfied:

$$2^{kj+1} < c5 \cdot n$$

where c5 is a real valued constant in the range (0,1).

In addition to determining suitable values for the constants c1,..,c5 above a number of other issues requiring the use of a test system must be resolved. These issues are outstanding in regard to an ultimate demonstration of feasibility of practical self-organizing or self-adapting sytems.

In order to make decisions about its physical

organization SORAAM requires the dynamic maintenance of various information. This information can be categorized as follows:

(a) information comprising a characterization of use and

(b) information regarding anticipated index effectiveness and ineffectiveness.

It is certainly not yet clear that dynamic unconditional maintenance of such information is feasible. With our high regard for transaction response such "procedural" overhead must be experimentally demonstrated acceptable.

There are a number of aspects of the kind of horizontal clustering supported in SORAAM (of the choice of data structure for the primary index) that are, as yet, unclear. The most obvious questions one may ask concern comparisons of our multi-dimensional clustering with traditional 1-dimensional clustering in regard to retrieval response. More succinctly, is it all worth the extra trouble; is mass storage I/O per query significantly reduced?[1]

SORAAM has also incurred a rebuilding cost arising from its selection of a static data structure for the primary index. It is therefore desirable to acquire a more concrete measure of this overhead in relation to various tuple insertion sequences, characterizations of use and, indeed, values of c1 above. Recall, for example, that sorted input may

---

[1]The simple example given in Section E of Chapter III, however, must certainly contribute to one's enthusiasm for k-d trees.

imply frequently occurring imbalance in the growth of a primary index.

In SORAAM the control of secondary index selection is not entirely transparent. In particular there were three user supplied parameters "Plow", "Phigh" and "S" that had great influence in this regard. Index creation is contingent not only on these parameters but on the nature of use of the base relation. Consider, for example, that the more complex a given <predicate> instance becomes the more probable the unconditional selection of the primary index for the query evaluation by the search strategies.[1] A much clearer picture of the nature of interaction of these user parameters and of query complexity is still necessary as well as the experimental determination of suitable default values for the parameters themselves.

Although harder to evaluate, some estimation of the overall effectiveness of the search strategies component of SORAAM is clearly called for. In particular, the accuracy of the cost estimation routines in predicting "best possible" choice of access path is, as yet, unknown.

We have seen, therefore, a number of outstanding issues requiring the design and implementation of a test environment for their resolution. Such a test system would differ markedly

---

[1]Surprisingly, should all <predicate> instances be complex enough for a base relation then, no matter how heavily used, no benefit may accrue from the creation of <u>any</u> secondary index.

from a "full-scale" production system in a number of important ways:

(a) Support for actual storage on mass storage media is not necessary. Realistic turnaround for simulations in the test system imply complete main memory residence. Large numbers of leaf tuple blocks would then imply small "toy" buffer sizes.

(b) The test system would involve the use of an entirely different user interface. It must be possible for the user to specify whole sets of transactions and timing of transaction arrival, determine the accumulation and reporting of much more extensive statistical information and have more effective control over all aspects of the system (e.g. the values of c1 to c5 above).

(c) A production system would almost certainly imply extensive modification to the file system of the anticipated operating system environment; the test system would not.

The test system would be comprised of four major parts:

(a) a transaction stream generator (including maintenance transactions)[1],

(b) a test data generator[1],

(c) an access methods simulator and

(d) a statistical acquisition and report generator.

Fundamental to the design of the test system would be the nature of the user interface for the above. Note that the simple design outline of the components given here of a test system would have little correspondence to those of a production system. Again we stress that the motivations for a testing environment discussed here are to facilitate

---

[1]See [Farley and Schuster 75] for a discussion of these components for an analogous testing environment.

acquisition of empirical evidence of feasibility of access path level design (in particular that of SORAAM).

B. Conclusions

In this thesis we have begun an investigation of automating physical reorganizational requirements of an RDMS. A review of "state-of-the-art" experimental systems appropriate for on-line applications revealed a lack of regard for this problem. The first section of Chapter II, therefore, presented arguments strongly motivating additional research. We observed the growing tendancy of operating systems to assume more direct responsibility for support of data management needs. In particular, an operating system environment in which an RDMS has supplanted the role of a file system firmly implied the need for such automation.

A design specification and outline for the access path level of an RDMS appropriate for such an environment (SORAAM) was given. Implicit in this design was the automation of two major physical organization details concerning a type of record or tuple clustering and index selection. Analysis of these details has yielded an access path level with significant advantages gained over other systems. We feel the results of this analysis have contributed significantly to our overall optimism for both the desirability and feasibility of such automation.

A complete demonstration of feasibility for our proposed access path level and for its design principles rests heavily, however, on future experimental research. We have outlined what we believe to be the next such step in the preceeding section. With regard to the legitimacy of the motivating notion of an "RDMS instead of a file system" in a general purpose computing environment perhaps the only acceptable evidence rests with the user feedback from a full-scale production system.

In formulating the problems of clustering and index selection careful attention to the meaning of existing terminology was necessary. In particular a strong advantage was realized in distinguishing between a primary index, in which we dealt with horizontal clustering, and secondary indices, in which we improved indexing performance (reduced, that is, the length of access path).

We saw, however, that these two notions are far from independent. A primary index, by virtue of its effect on tuple clustering, clearly influenced the selection of a secondary index for the purpose of query evaluation which in turn influenced the selection of the secondary indexing set itself. A more subtle relationship (not previously discussed) exists in the opposite direction. The choice of secondary indexing set can affect the action of the primary index towards tuple clustering. Should, for example, an attribute A of a base relation R satisfy the following:

(a) a secondary index always exists on A,

(b) no queries specify value ranges for A,

(c) the query sequence implies a random distribution of A-values and

(d) the number of unique values in R for A is close to the number of tuples in R

then little justification exists for ever discriminating on A in the primary index. The characterization of A given above is an example of the sort of attribute for which tuple blocking can have no beneficial effect. Accounting for such cases in SORAAM, however, is an open problem.

The architecture of the specific variety of secondary indexing considered by SORAAM and the problem of automating their selection led, in an almost obvious manner, to a formulation of this problem in terms of the well known (and well studied) "knapsack problem". The latter is also well known to be a computationally difficult problem.[1] In light of the very basic secondary indexing considered such a result lends credence to the use of appropriate heuristic approaches to the problem of secondary index selection in general.

Recall that such a heuristic approach was hinted at the end of Section D of Chapter III in the extension of SORAAM to support for multi-attribute inversions as secondary indices. Such an extension is one of many possible avenues for future research. We conclude with some suggestions in this regard:

(a) the extension of SORAAM to support for other

---

[1]It is a member of the class of such difficult problems denoted as "NP-complete".

forms of secondary indexing and index selection methodology.

(b) consideration for the problem of information archiving.

(c) the extension of SORAAM to support for large numbers of small sized base relations. Vertical clustering of base relations becomes very significant if base relation size is small ccmpared to the size of a disk block. We have clearly concentrated in this thesis on support for base relations where the opposite is true.

(d) the design of the relational language level (SORAAM's user).

(e) consideration of the remaining physical reorganizational requirements (see the Introduction).

(f) the implementation of a complete operating system environment including the "integrated" RDMS that is completely self-organizing with regard to the physical organization of itself.

References

[Aho and Ullman 79]
Aho, Alfred V. and Ullman, Jeffrey D., "Optimal Partial-Match Retrieval When Fields Are Independently Specified", ACM Transactions on Database Systems, Volume 4 Number 2, pp. 168-179, June(1979).

[Anderson and Berra 77]
Anderson, Henry D. and Berra, Bruce P., "Minimum Cost Selection of Secondary Indexes for Formatted Files", ACM Transactions on Database Systems, Volume 2 Number 1, pp. 68-90, March(1977).

[Astrahan et al. 76]
Astrahan, M. M., Blasgen, D. D., Chamberlin, D. D., Eswaran, K. P., Gray, J. N., Griffiths, P. P., King, W. F., Lorie, R. A., McJones, P. R., Mehl, J. W., Putzolu, G. R., Traiger, I. L., Wade, B. W. and Watson, V., "System R: Relational Approach to Database Management", ACM Transactions on Database Systems, Volume 1 Number 2, pp. 97-137, June(1976).

[Bayer and McCreight 72]
Bayer, R. and McCreight, E., "Organization and Maintenance of Large Ordered Indices", Acta Informatica, Volume 1, Springer-Verlag publisher, pp. 173-189, (1972).

[Bayer and Schkolnick 76]
Bayer, R. and Schkolnick, M., "Concurrency of Operations On B-trees", Research Report RJ 1791, IBM Research Laboratory, San Jose, May(1976).

[Bayer and Unterauer 76]
Bayer, R. and Unterauer, K., "Prefix B-trees", Research Report RJ 1796, IBM Research Laboratory, San Jose, June(1976).

[Bentley 75]
Bentley, Jon Louis, "Multidimentional Binary Search Trees Used For Associative Searching", CACM Volume 18 Number 9, pp. 509-517, September(1975).

[Bentley 78a]

-----, "A Survey of Algorithms and Data Structures for Range Searching", Technical Report CMU-CS-78-136, Carnegie-Mellon University, August(1978).

[Bentley 78b]
-----, "Multidimensional Binary Search Trees In Database Applications", Technical Report CMU-CS-78-139, Carnegie-Mellon University, September(1978).

[Bentley 78c]
-----, "Decomposable Searching Problems", Technical Report CMU-CS-78-145, Carnegie-Mellon University, August(1978).

[Berryman and Fowler 79]
Berryman, J. and Fowler, A., "Building a Database Management System Into MTS", Internal Memo, The Computing Center, University of British Columbia, May(1979).

[Berstein 79]
Berstein, Philip A. (editor), "Proceedings of the ACM-SIGMOD International Conference on Management of Data", May(1979).

[Blasgen and Eswaran 77]
Blasgen, M. W. and Eswaran, K. P., "Storage and Access in Relational Data Bases", IBM Systems Journal, Volume 16 Number 4, pp. 363-377, November(1977).

[Brodie et al. 75]
Brodie, M. L. (editor), Chan, S., Czarnik, B., Leong, E. Schuster, S. and Tsichritzis, D., "ZETA: A Prototype Relational Data Base Management System", Technical Report CSRG-51, Computer Science Research Group, University of Toronto, February(1975).

[Cargill 79]
Cargill, T. A., "A View of Source Text for Diversely Configurable Software", proposed Ph.D. thesis, University of Waterloo, (1979).

[Chamberlin et al. 76]
Chamberlin, D. D., Astrahan, M. M., Eswaran, K. P., Griffiths, P. P., Lorie, R. A., Mehl, J. W., Reisner, P. and Wade, B. W., "SEQUEL 2: A Unified Approach to Data

Definition, Manipulation, and Control", IBM Journal of Research and Development, November(1976).

[Chamberlin 76]
Chamberlin, D. D., "Relational Data Base Management Systems", Research Report RJ 1729, IBM Research Laboratory, San Jose, February(1976).

[Codd 70]
Codd, E. F., "A Relational Model of Data For Large Shared Data Banks", CACM Volume 13 Number 6, pp. 377-387, June(1970).

[Codd 79]
-----, "Extending the Database Relational Model To Capture More Meaning", Supplement to Proceedings of ACM-SIGMOD International Conference on Management of Data, pp. 29-52, see [Berstein 79].

[CODASYL 71]
CODASYL, "Data Base Task Group Report", CODASYL DBTG, ACM, New York, April(1971).

[CODASYL 73]
-----, "Data Definition Language Committee Journal of Development", CODASYL DDLC, IFIP Administrative Data Processing Group, Amsterdam, June(1973).

[Comer 78]
Comer, Douglas, "The Difficulty of Optimum Index Selection", ACM Transactions on Database Systems, Volume 3 Number 4, pp. 440-445, December(1978).

[Comer 79]
-----, "The Ubiquitous B-Tree", ACM Computing Surveys, Volume 11 Number 2, pp. 121-137, June(1979).

[Datapro 79]
Datapro Research Corp., "TOTAL, Cincom Systems, Inc.", databook 70 - the EDP buyer's bible, Volume 3, software section, (1978).

[Date 77]
Date, C. J., "An Introduction to Database Systems",

Second Edition, Addison-Wesley Publishing Company, ISBN 0-201-14456-5, (1977).

[Fagin 78]
Fagin, Ronald, "Extendable Hashing - A Fast Access Method for Dynamic Files", Research Report RJ 2305, IBM Research Laboratory, San Jose, July(1978).

[Farley and Schuster 75]
Farley, Gilles J. H. and Schuster, Stewart A., "Query Execution and Index Selection for Relational Data Bases", Technical Report CSRG-53, Computer Systems Research Group, University of Toronto, March(1975).

[Gray 78]
Gray, J., "Notes On Data Base Operating Systems", Research Report RJ 2188, IBM Research Laboratory, San Jose, February(1978).

[Hall et al. 76]
Hall, P., Owlett, J. and Todd, S., "Relations and Entities", Proceedings IFIP TC-2 Working Conference on Modelling in Database Management Systems, pp. 201-220, see [Nijssen 76b].

[Henry 78]
Henry, G. G., "Introduction to IBM System/38 Architecture", IBM S/38 TECH. DEV., pp. 3-6, see [IBM 78].

[Hewitt 72]
Hewitt, Carl, "Descriptions and Theoretical Analysis (using Schemata) of PLANNER: A Language for Proving Theorems and Manipulating Models in a Robot", Ph.D. Thesis, AI-TR-258, The Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Massachusetts, (1972).

[IBM 78]
IBM, "IBM System/38 Technical Developments", IBM Product Design and Development, General Systems Division, ISBN 0-9333186-00-2, (1978).

[Klimbie and Koffeman 74]
Klimbie, J. W. and Koffeman, K. L. (editors), "Data Base

Management", North-Holland Publishing Company, North-Holland ISBN: 0 7204 2809 2, (1974).

[Knuth 73]
Knuth, Donald E., "The Art of Computer Programming - Sorting and Searching", Volume 3, Addison-Wesley Publishing Company, ISBN 0-201-03803-X, (1973).

[Kollias et al. 77]
Kollias, J. G., Stocker, P. M., and Dearnley, P. A., "Improving the Performance of an Intelligent Data Management System", The Computer Journal, The British Computer Society, Volume 20 Number 4, ISSN 0010-4620, November (1977).

[Lomet 79]
Lomet, David B., "Multi-Table Search for B-tree Files", Proceedings of ACM-SIGMOD International Conference on Management of Data, pp. 35-42, see [Berstein 79].

[Lorie 74]
Lorie, R. A., "XRM - An Extended (n-ary) Relational Memory", Technical Report No. 320-2096, IBM Cambridge Scientific Center, January (1974).

[McDonell 77]
McDonell, Ken J., "A Unified Approach To Secondary Storage Input-Output Operations", Ph.D. Thesis, Technical Report TR77-6, University of Alberta, September (1977).

[McGee 76]
McGee, William C., "On User Criteria for Data Model Evaluation", ACM Transactions on Database Systems, Volume 1 Number 4, pp. 370-387, December (1976).

[Nijssen 76a]
Nijssen, G. M., "A Gross Architecture for the Next Generation Database Management Systems", Proceedings IFIP TC-2 Working Conference on Modelling in Database Management Systems, pp. 1-24, see [Nijssen 76b].

[Nijssen 76b]
Nijssen, G. M. (editor), "Modelling in Database Management Systems", North-Holland Publishing Company, ISBN 0 7204 0459 2, (1976).

[Prywes and Gray 63]
Prywes, N. S. and Gray, H. J., "The Organization of a Multilist-Type Associative Memory", IEEE Transactions on Communication and Electronics, pp. 488-492, September(1963).

[Rodriguez and Echhouse 77]
Rodriquez, Rosell J. and Echhouse, R., "Management of Data by Future Operating Systems", New Directions for Operating Systems, A Workshop Report, Browne, J. C. (editor), ACM SIGOPS Operating Systems Review, Volume 11 Number 1, pp.23-25, (1977).

[Sahni 75]
Sahni, Sartaj, "Approximate Algorithms for the 0/1 Knapsack Problem", Journal of the ACM, Volume 22 Number 1, pp. 115-124, January(1975).

[Salton et al. 77]
Salton, Gerard, Bergmark, D. and Wong, A., "Generation and Search of Clustered Files", Technical Report TR 77-299, Dept. of Comp. Sci, Cornell University, (1977).

[Salton 79a]
Salton, Gerard, "Suggestions for a Uniform Representation of Query and Record Content in Data Base and Document Retrieval", Technical Report TR79-363, Cornell University, (1979).

[Salton 79b]
-----, "A Progress Report On Automatic Information Retrieval", Technical Report TR79-368, Cornell University, (1979).

[Schkolnick 74]
Schkolnick, M., "The Optimal Selection of Secondary Indices for Files", Technical Report, Carnegie-Mellon University, November(1974).

[Schkolnick 76]
-----, "A Clustering Algorithm for Hierarchical Structures", Research Report RJ 1806, IBM Research Laboratory, San Jose, July(1976).

[Schmid et al. 75]

Schmid, H. A. (editor), Bernstein, P. A. (editor), Arlow, B., Baker, R. and Pozgaj, S., "The Relational Data Base System OMEGA (Progress Report)", Technical Report CSRG-72, Computer Systems Research Group, University of Toronto, July(1976).

[Selinger et al. 79]
Selinger, P. G., Astrahan, M. M., Chamberlin, D. D., Lorie, R. A. and Price, T. G., "Access Path Selection in a Relational Database Management System", Proceedings of ACM-SIGMOD International Conference on Management of Data, pp. 23-34, see [Berstein 79].

[Stocker and Dearnley 72]
Stocher, P. M. and Dearnley, P. A., "Self-Organizing Data Management Systems", British Computer Journal, Volume 16 Number 2, pp. 100-105, (1972).

[Stocker and Dearnley 74]
-----, "A Self-Organizing Data Base Management System", Proceedings IFIP Working Conference On Data Base Management, pp. 337-349, see [Klimbie and Koffeman 74].

[Stonebraker et al. 76]
Stonebraker, M., Wong, E., Kreps, P. and Held, G., "The Design and Implementation of INGRES" ACM Tranactions on Database Systems, Volume 1 Number 3, pp. 189-222, September(1976).

[Todd 76]
Todd, S. J. P., "The Peterlee Relational Test Vehicle - a system overview", IBM Systems Journal, Volume 15 Number 4, pp. 285-308, (1976).

[Tsichritzis and Lochovsky 77]
Tsichritzis, Dionysios C. and Lochovsky, Frederick H., "Data Base Management Systems", Academic Press, Inc., ISBN 0-12-701740-2, (1977).

[Watson and Aberle 78]
Watson, C. T. and Aberle, G. F., "System/38 Machine Data Base Support", IBM S/38 TECH. DEV., pp 59-62, see [IBM 78].

[Wiederhold 77]

Wiederhold, Gio, "Database Design", McGraw-Hill publisher, ISBN 0-07-070130-X, (1977).

[Wong and Youssefi 76]
Wong, Eugene and Youssefi, Karel, "Decomposition - A Strategy for Query Processing", ACM Transactions on Database Systems, Volume 1 Number 3, pp. 223-241, September(1976).

[Yao et al. 76]
Yao, S. B., Das, K. S. and Teorey, T. J., "A Dynamic Database Reorganization Algorithm", ACM Transactions on Database Systems, Volume 1 Number 2, pp. 159-174, June(1976).

Appendix A. Base Relations Used by SORAAM

This appendix presents a more detailed introduction to the three base relations and their attributes used by SORAAM itself. Table III-1 in the first section of Chapter III outlined the schema for these relations.

A description of each of the attributes given in Table III-1 follows. The descriptions themselves are intentionally cursory since their motivation and use are a central topic in Chapter III of the thesis. The reader is encouraged to refer to this appendix during the course of reading Chapter III.

In the following definitions those attributes whose values are automatically maintained by SORAAM are postfixed with "*":

RELATIONS

Rname = the character string identifier of a base relation.

Bsize* = blocksize in terms of records per block of a base relation.

N* = the number of tuples in a base relation.

D* = the maximum depth of the primary index tree structure of a base relation (equals the sum of the associated attribute's "d" values).

PIB* = the number of disk blocks used for the primary index of a base relation.

PIBloc* = the location of the root block for the primary index of a base relation.

IX* = the number of internal nodes used in the primary index of a base relation.

STPB* = the number of secondary tuple pointer buffers per index on a base relation (equals N/[ff2*Psize]).

ff1* = the fill factor of a secondary index.

ff2* = the fill factor of a tuple pointer buffer.

Psize* = the tuple pointer blocksize in terms of pointers per block.

g.p.m.* = the global performance measure of a base relation (equals RDQreads/DQreads).

Plow = the user supplied lower bound on the g.p.m. of a base relation.

Phigh = the user supplied upper bound on the g.p.m. of a base relation.

S = the fraction of storage overhead permitted for secondary indices on the associated base relations.

qno* = the relative number to the "f" of the associated attributes of queries on the base relation.

qnoold* = as above but relative instead to the "fold" of the associated attributes.

radix1 = the maximum permitted "qno" value.

RDQreads* = the relative number to "DQreads" of relevent block reads for query response of a base relation. A relevent block was one that contained a tuple that was retrieved.

DQreads* = the relative number to "RDQreads" of total block reads for query response of a base relation. This includes primary and secondary index blocks.

radix2 = the maximum permitted "DQreads" value.

Pcost* = the relative number of the total block reads for query response, assuming no indexing, to the "b" of the associated attributes.

radix3 = the maximum permitted "Pcost" value.

buf = the number of main memory buffers available for operations on a base relation.

ATTRIBUTES

Aname = the character string identifier of an attribute name.

COL# = the column number in which values of an attribute occur in tuples of a base relation.

format = the storage format for values of an attribute.

n* = the number of distinct values of an attribute occurring in a base relation (assumed

not greater than N ... the number of tuples in the base relation).

d* = "d" conveys attribute dependent information about the primary index of a base relation. Specifically it represents the number of levels in the k-d tree discriminating on the associated attribute.

Iflag* = if a secondary index currently exists on the associated attribute of a base relation then Iflag is a true value; otherwise false.

Isize* = the index blocksize in terms of attribute values per block.

SIB* = the number of secondary index blocks used if a secondary index exists on the associated attribute (equals n/[ff1*Isize]).

SIBloc* = the location of the root block of a secondary index (should it exist) on the associated attribute.

f* = the relative access frequency to the "f" of the other attributes and "qno" of the associated base relation.

fold* = as above but relative instead to the "fold" and "qnoold" values.

min* = for numeric attributes the current minimum value occurring in the base relation.

max* = for numeric attributes the current maximum value occurring in the base relation.

b* = the "benefit" value of a secondary index on the associated attribute of a base relation. This value is relative to the "b" of the other attributes and to "Pcost" of the associated base relation.

PROCESSES

process# = the identification of a process using SORAAM (assigned usually by the operating system).

quiet# = a "quiet" log record identifier.