Algorithms and Data Structures for the Implementation of a Relational Database System

by

© Jack A. Orenstein

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree Doctor of Philosophy

School of Computer Science
McGill University
Montreal, Quebec, Canada

November, 1982

Abstract

The problems of implementing a relational database are considered. In part 1, a new class of data structures for processing range queries is described. A member of this class is derived from a data structure which supports random and sequential accessing. We also describe two new data structures with this property that seem to have better performance than the Btree. In part 2, a new design for the physical database is proposed. This design is based on the separation of a relation into two parts: a static "master file" and a dynamic "differential file" which stores updates. Our design includes a new system for recovering from system failures and allows greater concurrency than is possible with existing systems.

Résumé

On considere des problemes d'implantation d'une base relationnelle de données. En premiere lieu, on décrit une nouvelle classe de structures de données pour le traitement des requêtes sur des rangees des valeurs ("range queries"). Un membre de cette classe est dérivé d'une structure de données qui soutient l'accès aléatoire ou séquential. On décrit aussi deux nouvelles structures de données avec cette propriété, et qui paraissent avoir une meilleure performance que l'arbre "B" ("B-tree"). En deuxieme lieu, on propose un nouveau plan pour la base physique de données. Ce plan est basé sur la séparation d'une relation en deux parties: un fichier principal statique et un fichier differential, ("differential file") dynamique qui garde les changements en mémoire. La plan comprend un nouveau système pour la reprise des échecs de système et permet un parallelisme plus grande que les systèmes existants.

Acknowledgement

I am indebted to my thesis adviser, Prof. Tim Merrett, for many things: for his constant encouragement and interest in this work, for our many hours of discussions about these and related topics, for helping me to organize this thesis and for getting me interested in the whole thing to start with.

I am also grateful to Prof. Luc Devroye for several enlightening discussions about data structures and to Dave Perlin for helping me with UNIX.

I thank McGill University and the Friends of McGill for financial support.

To my parents

...zzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzz...

Rip van Winkle

Go back, Jack, do it again.

Steely Dan

I thought how unpleasant it is to be locked out; and I thought it worse perhaps to be locked in.

Virginia Woolf

Contents

Chapter 1

Introduction

The relational model of data [Codd70] provides a simple view of data and powerful operators for manipulating it. In spite of this, relational databases are not widely used. More commonly, database systems are based on other models of data which are more complicated and provide lower level operators, or, systems are custom-built for specific applications.

One of the reasons for this situation is that the relational model of data is difficult to implement efficiently. The abstract view of the data does not correspond to any hardware facility provided by current computers. The operators are therefore "high level": they must be implemented using complex software. The network model [CODA71], on the other hand, provides a "low level" view of data which strongly indicates a particular storage scheme. The operators of this data model are relatively easy to implement.

Several relational database systems, mostly experimental, were implemented between 1970 and 1976. Since then, the problems of implementing a distributed relational database system have received a lot of attention. The sites of a distributed database are loosely coupled: each site can function, to some extent, as an independent database system. In this thesis, the problem of implementing the lowest level of a database system, the "physical database", is considered. This is an important practical problem because the centralized systems developed through 1976 are unnecessarily complicated and slow, and because, as suggested above, solutions to this problem are also applicable to distributed database systems.

We assume that the database will run on a typical computer system: a CPU with volatile primary memory and a much larger

amount of non-volatile, random access secondary memory, e.g. disk storage.

1. Architecture of a relational database system

A relational database system can be seen as a hierarchy of machines as shown in figure 1. Each user communicates with the logical database in a language meaningful to the user. Several users may use the database simultaneously.

The logical database translates the operations specified by a user into operations on relations: operations of the relational algebra and operations for updating the relations.

The physical database (PDB) provides the mapping from operations on relations to operations on data structures stored in disk files.

A major benefit of this organization is that the various levels, (the logical and physical databases etc.) are independent of one another. The implementation of any level can be changed without affecting other levels. This thesis proposes a new design for the physical database which is quite different from previous designs. Due to the independence of the various levels of the database there is no impact on levels above and below. (However a useful extension of the relational model can be easily supported using our design.)

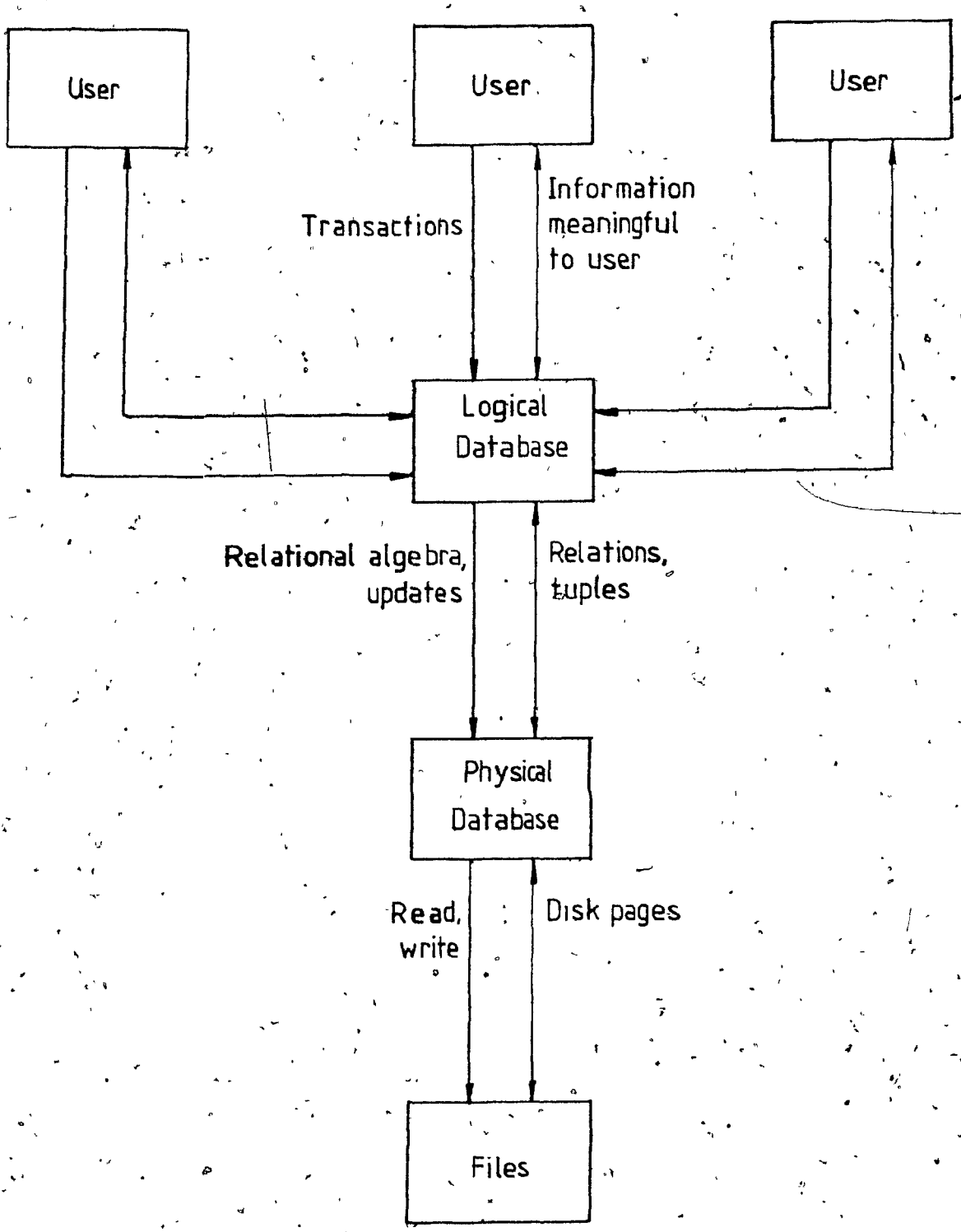


Figure 1. A relational database system.

2. Operations on relations

All actions on the database, initiated by users, are translated into operations on relations. These operations are described in this section.

An attribute is an identifier. Associated with each attribute, A, is a domain dom(A): a set of values. A k-ary relation, R, on the set of attributes $\{A_0, \ldots, A_{k-1}\}$, $R(A_0, \ldots, A_{k-1})$ is a subset of the cartesian product $dom(A_0) \times \ldots \times dom(A_{k-1})$. An element of the relation, $[a_0, \ldots, a_{k-1}]$, is a tuple. $a_i \in dom(A_i)$, $i = 0, \ldots, k-1$.

The values in a domain are all of the same type and these types are usually atomic, e.g. strings or numbers. In practice the cardinality of each $dom(A_i)$ is finite and is known.

2.1. The relational algebra

Codd defined some operators for manipulating relations. These comprise the relational algebra [Codd70]. Merrett has defined additional operators which generalize those of Codd [Merr77, Chiu82]. We now describe these operators using relations $R(X,Y_R)$ and $S(Y_S,Z)$; $dom(Y_R) = dom(Y_S)$. Upper case letters: R, S, X, Y, Z denote relations and attributes. Lower case letters: r, s, x, y, z denote tuples and domain values.

2.1.1. Selection

$$R(Q) = \{r: r \in R \ \& \ Q(r)\}$$

Q is a query (or "predicate"). R(Q) consists of those tuples of R which satisfy the query. Note that Q has exactly one tuple as its argument.

Various kinds of queries are discussed in section 2.1.7.

2.1.2. Projection

$$R[X] = \{r[X] : r \in R\}$$

r[X] is a tuple of R with the Y_R value removed. Note that duplicates are "removed" since a relation is a set.

$R[x,Y_R]$ is an abbreviation for $R(X = x)[Y_R]$, (selection followed by projection).

2.1.3. Θ-join

$$R[Y_R \Theta Y_S]S = \{[x,y_R,y_S,z]: [x,y_R] \in R \ \& \ [y_S,z] \in S \ \& \ \Theta(y_R,y_S)\}$$

where $\Theta(i,j)$ is one of the comparison operators: $i = j$, $i \neq j$, $i < j$, $i \leq j$, $i > j$ or $i \geq j$.

The natural join is

$$R[Y_R * Y_S]S = (R[Y_R = Y_S])[X,Y_R,Z]$$

$R[Y_R * Y_S]$ may be abbreviated to R*S when there is no ambiguity.

2.1.4. μ-join

The natural join $R[Y_R * Y_S]S$ can be defined as follows:

$$R[Y_R * Y_S]S = \{[x,y,z] : y \in R[Y_R] \cap S[Y_S] \ \& \ [x,y] \in R[X,y] \ \& \ [y,z] \in S[y,Z]\}$$

Thus the natural join is the "intersection" join: $R[Y_R \cap Y_S]S$. By replacing "$\cap$" by other set operations, other kinds of joins are created. These are the μ-joins. Some of these joins (e.g. the union join) require the introduction of null values. For example, suppose that $R[X,y] = \{\}$ and $S[y,Z] = \{[y,z]\}$. The intersection join does not have a tuple containing y, but the union join has the tuple $[-,y,z]$.

2.1.5. σ-join

$$R[Y_R \ \sigma \ Y_S]S = \{[x] : x \in R[X] \ \& \ \sigma(R[x,Y_R],S[Y_S])\}$$

where $\sigma(P,Q)$ is one of the set comparison operators: $P \subseteq Q$, $P \supseteq Q$, $P \subset Q$, $P \supset Q$, $P \cap Q = \emptyset$ or $P \cap Q \neq \emptyset$. Codd's division operator is obtained with $\sigma(P,Q) = P \supseteq Q$.

2.1.6. Set operations

Relations $X(A_0, \ldots, A_{k-1})$ and $Y(B_0, \ldots, B_{j-1})$ are union-compatible if $k = j$ and $dom(A_i) = dom(B_i)$, $i = 0, \ldots, k-1$. The set operations can be applied to union-compatible relations to yield other relations. (N.B. It does not make sense to apply set operations to relations which are not union-compatible. For example, $X(A_0,A_1) \cup Y(B_0,B_1,B_2)$ is not even a relation since it contains both 2-ary and 3-ary tuples.)

2.1.7. Important classes of queries

We will be mostly concerned with the selection operation. There are several reasons for this:

- It is a fundamental operation. Finding efficient ways to search for information in a file, (essentially the same problem), has been the motivation for a huge amount of research.

- As far as performance is concerned, selection is the most important operator, (see chapter 5 section 4.2).
- Data structures and algorithms for searching multidimensional data (i.e. relations) efficiently is currently an active research topic.

It will be convenient to consider each domain as a set of integers $\{0, \ldots, D-1\}$ where D is the cardinality of the domain. This point is discussed further in section 4.

The query used in selection is Q(t), a logical function of one tuple. A large and important class of queries is the class of range queries. A range query is

$$Q([a_0, \ldots, a_{k-1}]) = L_0 \leq a_0 \leq U_0 \ \& \ \ldots \ \& \ L_{k-1} \leq a_{k-1} \leq U_{k-1}$$

That is, a lower and upper bound is specified for each attribute. "$\leq$" is a numerical comparison since we are considering each domain to be composed of integers.

If no restriction is placed on attribute i then $L_i = 0$ and $U_i = D_i - 1$ where $D_i = |dom(A_i)|$.

A partial match query is a range query in which $L_i = U_i$ or no restriction is placed on A_i, ($L_i = 0$ and $U_i = D_i - 1$). Even the class of partial match queries is an important one. For example, locating a record given its key is a partial match query.

2.2. Updating relations

It is usually assumed that a relation is updated by inserting a tuple, deleting a tuple or modifying one or more attribute values of a tuple. Other ways of thinking about updates will be presented in section 4. In that section we will discuss the advantages of using just two update operations: insert and delete.

2.3. Forming transactions

A sequence of operations on relations can be grouped into a transaction using the commands Start and End as delimiters. A transaction is a sequence of operations that should be treated as an atomic operation (from the user's point of view). That is, a transaction takes effect at some instant. Before that instant, none of its updates are in effect; afterwards, all of them are.)

3. The physical database, (PDB)

The problem considered in this thesis is the implementation of the physical database; i.e. the software that creates the machine used by the logical database. Relations must be represented using data structures stored in files in secondary memory. Operations on relations must be translated to operations on these data structures. Finally, the atomicity of transactions must be guaranteed. This last requirement would not be a problem except for two facts of life:

1) Several users can work on the database simultaneously.

Atomicity requires that a user not see the database in a state corresponding to the partial execution of another user's transaction.

2) The system will occasionally "crash", sometimes damaging the contents of secondary memory. Following recovery, transactions interrupted by the crash must not remain "partially executed".

For these reasons, a concurrency control system and a recovery system must be included in the physical database.

Both of these issues are more complicated in a distributed system but the additional complexity does not show up in the physical database; it has to do with communication among the sites. The actions to be performed at each site are the same as for a centralized database. Thus our techniques are applicable in both environments."

4. Models of relations

A relation is commonly thought of as a table where a column represents an attribute and a row represents a tuple. Other models of relations can be imagined. Our solutions to some of the problems of physical database design were obtained only after an appropriate model was selected. In this section, various models of relations are explained.

The "table" model seems to have had a very strong influence on builders of databases. In all relational database systems that we know of, relations are stored in indexed-sequential or hash files, augmented by inversions. This is a very obvious organization if a relation is thought of as a table: the model

allows the implementer to think in terms of pointers to rows of a table. Furthermore, any ordering of the rows is reflected in the ordering provided by an indexed-sequential file. Several problems with this organization are discussed in chapter 2.

Another model, the "space" model, has generated several data structures for range searching and has motivated our own work in this area. Recall that each domain is considered to be a finite set of integers, $\{0, \ldots, D-1\}$ where D is the cardinality of the domain. The transformation of non-integer data, e.g. strings and reals, to integers is trivial. The "$\leq$" ordering of the integers can be used to reflect an ordering of the elements of the original domain.

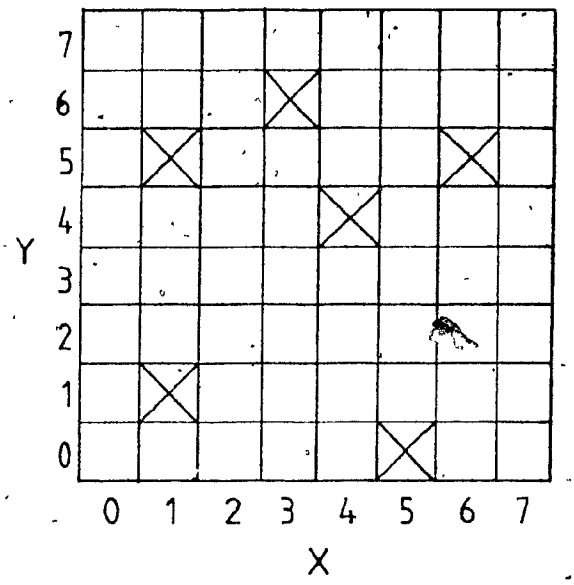
Now the cartesian product, $dom(A_0) \times \ldots \times dom(A_{k-1})$ can be seen as a k-dimensional space of bits. The bit at coordinates $(a_0, \ldots, a_{k-1})$ is on iff $[a_0, \ldots, a_{k-1}]$ is a tuple of the relation, (see figure 2). Our work in chapter 3 is based on this model.

The "table" and "space" models are suitable for thinking about static relations. But the problems of concurrency control and recovery involve dynamic relations. Furthermore, if the recent history of the relation is available, it is possible to use concurrency control and recovery systems which are simpler and have better performance than would otherwise be possible. In dealing with these issues it was helpful to use another model of (dynamic) relations and to use a reduced set of update operations.

We view every element of $dom(A_0) \times \ldots \times dom(A_{k-1})$ as a tuple whether it is present or not in the relation at a given time. At any time, each tuple has a status of "present" or "absent". In

X	Y
1	1
1	5
3	6
5	0
4	4
6	5

Table representation



Space representation

Figure 2. A relation represented by the table and space models.

this model, the idea of "modifying" a tuple is meaningless. If any attribute value of a tuple is "changed" it becomes another tuple. Thus the only updates possible are insertion (change status from absent to present) and deletion (change status from present to absent). What is normally thought of as a modification can be achieved by an insertion and a deletion. This model is demonstrated in figure 3 which shows the history of a 1-ary relation.

A model which allows a "modify" update could be used but it is more complicated and the algorithms of chapters 5, 6 and 7 would be more complicated as a result. Also, if a tuple can be modified then its name must not be affected by the modification. Then primary key attributes (for example) cannot be modified.

To summarize, the modify operation is unnecessary, it complicates the algorithms of the physical database and it isn't always permissible so we do not include it.

5. Thesis outline

The thesis is divided into three parts. Part 1 is concerned with data structures: chapter 2 is a survey of data structures for searching files. The emphasis is on associative searching; finding all tuples that satisfy some predicate. Chapter 3 contains new results in this field. A class of data structures for range searching is described. A data structure in this class can be derived from a data structure which supports random and sequential accessing. We also describe two new data structures with this important property.

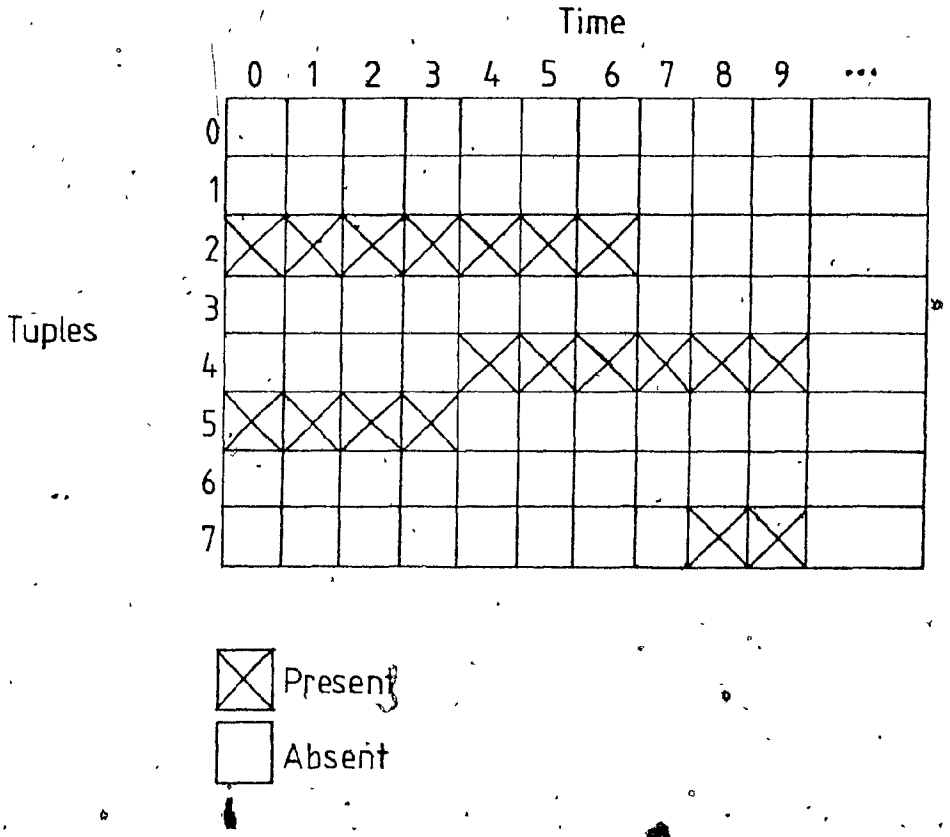


Figure 3. History of a 1-ary relation. The domain of the single attribute is {0, 1, 2, 3, 4, 5, 6, 7}.

In part 2, transaction processing is considered: chapter 4 is a survey of recovery and concurrency control techniques, (proposed and implemented). In chapter 5, a design for the physical database is given. The design is based on the "differential file" which simplifies both recovery and concurrency control.

In part 3, the results from parts 1 and 2 are integrated: chapter 6 discusses data structures for the representation of the components of the differential file system. In chapter 7, the design is extended to provide archives, (so that previous states of the database can be examined), and to provide improved protection from crashes. Chapter 8 contains a summary and conclusion.

Part One

Data Structures

Chapter 2

A Survey of Data Structures for Relational Databases

This chapter is a survey of the literature on data structures for use in a relational database. Most of the research on data structures for searching has considered the retrieval of a record given its key. This is essentially a query on one dimensional data. In a relational database system, the data is multidimensional and other techniques are needed. Some of these techniques are based on those used for one dimensional data.

In section 1, data structures for one dimensional and multidimensional data are described. In section 2, the data structures used in some implementations are described.

1. Data structures for associative searching

The problem of associative searching is to find all the tuples in a relation satisfying some predicate. We will restrict our attention to the class of range queries, (see chapter 1 section 2.1.7).

In order to support the processing of range queries on relations, a data structure should have at least the following three properties:

1) The cost of processing the query should decrease as more attributes are specified in the query.

2) Sequential accessing should be possible. That is, the successor according to some ordering can be located quickly. This is important since many algorithms, (e.g. for set operations), are based on merging which requires that the data be ordered.

3) It should be dynamic unless the data is known to be static.

1.1. Scans and inversions

The simplest method for processing any kind of query is to store a list of tuples and scan the entire list, checking each tuple against the query. This method is used by MRDS [Merr76] and PRTV [Todd76]. This organization is very easy to maintain but it is very inefficient when selective queries are being processed. It is not really a feasible approach.

The most popular method is to use inversions, (see [Knut73]). An inversion is a list of pairs (value, pointer-set), (possibly ordered on value) which permits the efficient retrieval of (v, P) given v. The value v may be the concatenation of values from several attributes. The access set is the set of attributes used to form v. To find the tuples associated with a value v, find the (v, P) entry in the inversion and then retrieve the tuples pointed to by each pointer in P. If the inversion is ordered on the value field then sequential processing is possible. The file of records may also be sorted on some access set, (it is then "clustered" [Astr76]), improving the performance of sequential processing on that access set. The relation may be clustered on no more than one set of attributes at a time.

An inversion may be stored in a variety of data structures such as ISAM [IBM66] or the Btree [Baye72] or one of its variants [Come79]. A single inversion, by itself, is suitable for processing range queries only on the access set of the inversion.

1.1.1. Hashing

Inversions can be stored in hash tables, but, unless the hash function is monotonic (i.e. order preserving) the processing of range queries will require a scan of the relation. Furthermore, relations are often dynamic. Therefore, traditional hashing methods, (see [Knut73] or [Stan80] for a survey), are not suitable for use in a relational database. Recently, some new hashing methods have been proposed which are order preserving and/or dynamic.

Extendible hashing [Fagi79] applies a hash function yielding a uniform distribution in [0, 1). The prefixes of the binary representations of the hash values are used to cluster groups of records together. When used with the hash function $h(k) = k$, extendible hashing is order preserving and the processing of one dimensional range queries is possible, but the distribution may no longer be uniform. (This is a special case of EXCELL [Tamm80]. See section 1.2.) Records can be inserted and deleted without causing degradation of performance.

Trie hashing [Litw81] is a related method. The records are stored in a trie. It is considered a form of hashing because only the prefix of the key of the record is used in classification, (i.e. not all the information is used). If a trie complete to its height were used, the leaf nodes would be the structure formed in order preserving extendible hashing. Both extendible hashing and trie hashing are dynamic.

Linear hashing [Litw80] is another dynamic method. It grows and shrinks as does extendible hashing but overflow records appear in the structure occasionally. The contents of the overflow records are eventually moved to the primary storage

area. Linear hashing is discussed in more detail in chapter 3 section 3.1.1.

Two order-preserving variations of linear hashing are discussed in chapter 3 section 3. A variation of linear hashing by Larson [Lars80] allows for a more even distribution of records to buckets but the order preserving variations do not apply.

1.1.2. Inversions used for multidimensional data

Two ad hoc methods have been proposed to use inversions for multidimensional data. Modifications of this type are necessary if general range queries are to be supported efficiently using inversions. One method is to create a key by concatenating the values of several attributes into a single value [Lum70]. This organization supports queries involving a prefix of the synthesized key and is therefore not completely general. C(k, $\lfloor k/2 \rfloor$) such inversions are necessary to obtain full generality.

Another method is to store several inversions each providing access on a different access set. A complex query on several attributes might access several inversions. Set operations on the retrieved pointer sets yield pointers to the tuples satisfying the query. There are two drawbacks to this method:

1) Each additional inversion has a cost in space and in time for maintenance.

2) A query involving several inversions requires the merging of pointer-sets. For conjunctive queries, the cost of this work increases as the size of the result decreases, (violating requirement (1)).

There are a number of ways to counter these problems. The problem of deciding which inversions to maintain can be

approached analytically [Schk75, Lum70], or, the decision can be based on observations of usage [Hamm76].

The time spent performing merges of pointer-sets from inversions in evaluating complex queries can be reduced by rearranging the parse tree which represents the query and using the transformed query for the search [Liu76]. Another approach to reducing merging costs is to simply ignore some inversions in certain situations. The merging cost is reduced but, in the case of conjunction, the number of tuples returned is greater. This method is used by System R [Astr76, Seli79].

It should be clear from this survey that inversions alone are inadequate for use in a relational database. There are data structures that can be used to implement inversions properly (e.g. Btree and trie hashing) but generality, the ability to process any range query efficiently, is achieved at a high price.

1.2. Multidimensional data structures

The methods for processing multidimensional range queries, discussed above, all use techniques designed for one dimensional data. Many data structures designed specifically for multidimensional data are known. Some of these have been surveyed in [Bent79b].

The multilist organization [Dodd69] organizes the records into several lists. Tuples containing the same value for some attribute are linked into a list representing the attribute value. This organization is expensive to maintain and the cost grows as more lists are maintained. Also, the method is not efficient for the evaluation of complex queries. It is not a

suitable data structure for processing range queries.

The doubly chained tree [Suss63], multiple attribute tree [Kash77], modified multiple attribute tree [Gopa80] and the multidimensional Btree (MDB tree) [Sche82] are based on the following idea: select the m attributes of the relation that will be queried. Order the attributes (so that some performance criterion such as access time or storage space will be optimized), yielding the permutation i_1 , ... , i_m. Store each tuple in an m level trie as dictated by its attribute values a_{i_1} , ... , a_{i_m} . Queries are evaluated by traversing the nodes at levels of the trie corresponding to attributes specified in the query. (Not all nodes of each such level would be visited.) These data structures differ primarily in the way the trie is represented. All of these data structures are biased against attributes low in the ordering: it is more expensive to query these attributes than those higher in the ordering. The MDB tree is the most recent data structure in this line of evolution. It can handle range queries and it is dynamic.

The multiple attribute tree and its relatives are not actually trees. They are tries of the type described by Rivest [Rive74].

Rivest considered the evaluation of partial match queries on relations stored in hash tables and tries. To store a relation in a hash table is to partition its records. Each partition contains all records in a "sub-cube" of the space representing the relation, (see chapter 1 section 4). One way of performing such a partitioning is to classify the tuples according to characters in the attribute values. This can be done in such a way that the average search time is minimized (assuming that all partial match queries are equally likely). The partitioning imposed is similar

to that of a trie; it was found that the trie has performance close to that of the hash table (for partial match queries).

The hashing scheme has also been proposed by Rothnie and Lozano: Multiple key hashing [Roth74] applies hash functions to attributes that will be queried. The concatenation of the hash values creates "characteristic tuples". All tuples yielding the same characteristic tuple are stored together. A partial match query is processed by applying the hash functions to the values specified in the query and retrieving the cells associated with the referenced characteristic tuples.

Since 1974, several data structures have appeared which are based on the view of a relation as a space of bits, (see chapter 1 section 4): A relation of degree k can be represented by a k dimensional space of bits. Attribute values are represented by integers 0, 1, ... , $|dom(A_i)|-1$, i = 0, ... , k-1. The tuple $[a_0, \ldots, a_{k-1}]$ is represented by an on bit at coordinates (a_0, ... , a_{k-1}). The assumption that each domain is finite is a reasonable one in practice. The translation of reals and strings to integers is trivial.

These data structures partition the space into cells containing no more than c on bits (or "points") each, where c is the capacity of a cell. Typically, a cell would be stored on one disk page. Some methods, instead of sub-dividing all cells containing more than c points, allow some cells to overflow.

The cells are stored in a data structure which is searched when a query is evaluated. Under this view of a relation, a query is a region of the space (usually, but not necessarily connected). In particular, a range query is represented by a hyper-rectangle. To process the query, the set of cells

overlapping the query region are retrieved. Each point in these cells is tested for inclusion in the query region.

The data structures based on these ideas differ primarily in the way they partition the space.

The first proposed data structure of this type was the quad tree [Fink74]. A quad tree node represents a point and has up to 2^k children, one for each non-empty sub-region generated by splitting the region through the point in all k directions. The size of a quad tree node is $O(2^k)$ and the tree has n such nodes. For large k the storage requirements are infeasible.

The k dimensional tree (abbreviated to "kd tree") [Bent75a, Bent79a] is a related data structure which avoids the problem: the kd tree has O(n) nodes with up to two children each regardless of k. This is achieved by splitting sub-regions in one direction only. Typically, a node on level i would represent a partition splitting attribute i mod k.

A balanced kd tree has O(log n) levels instead of the quad tree's O((log n)/k) levels. A balanced kd tree can be built in time O(n log n) [Bent79a] but a balanced quad tree is not always possible.

Neither the quad tree nor the kd tree are suitable for dynamic applications. Both can degenerate resulting in degradation of performance. No method for maintaining the balance of a kd tree is known.

The k dimensional Btree (K-D-B tree) [Robi81] is a multidimensional version of the Btree [Baye72]. Internal nodes store representations of regions and leaves store points. The insertion algorithm is more complicated than for a standard Btree. The deletion algorithm has not been fully worked out.

Experiments show that storage utilization following a sequence of insertions is 50% to 70% but there is no lower bound nor is the average known.

Multidimensional clustering [Liou77] partitions the space on each attribute in turn until each cell contains no more than c points. The positions of the partitions are stored in a list which, when searched, yields cell addresses. The entire directory is scanned to evaluate a query. The method is not truly dynamic. Insertions and deletions from the cells can be made as long as there is room and cells may become empty. But there is no provision for changing cell boundaries and updating the cell directory other than by reorganization.

In multidimensional paging [Merr78] each k-1 dimensional partition spans the entire k dimensional space. Thus a grid is created. The selection of the boundary positions is more difficult than with multidimensional clustering. The marginal distributions of the points are needed to determine good boundary positions. Due to the global nature of the partitions overflow is sometimes unavoidable (given a lower bound on load factor). Multipaging was originally a static data structure but a dynamic version has been proposed [Merr82].

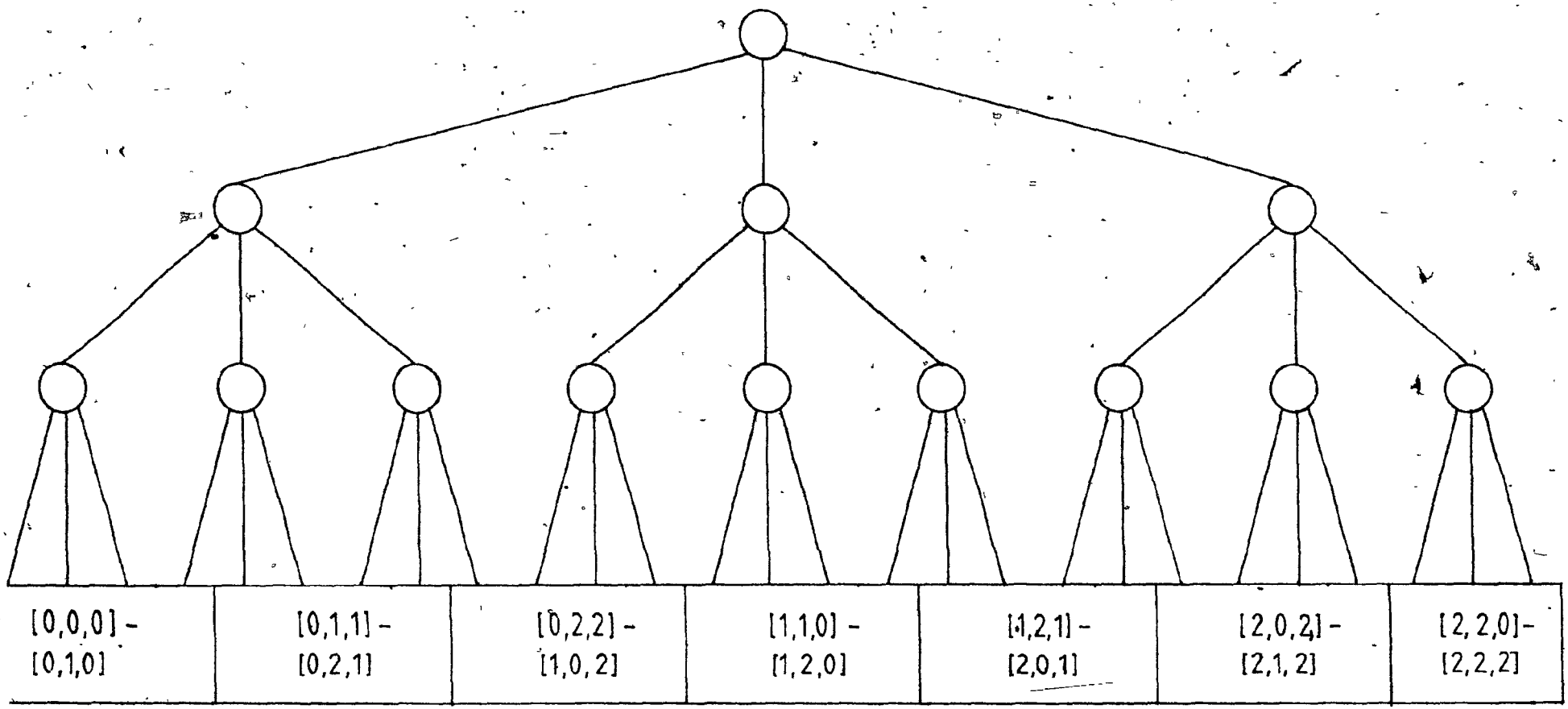
EXCELL, the extendible cell method [Tamm80], uses a very simple partitioning, (also a grid), which is independent of the data distribution. All cells have the same dimensions. The grid is made fine enough to avoid overflow, (there is no lower bound on storage utilization). EXCELL uses extendible hashing [Fagi79] with a hash function that interleaves the bits of the binary representations of the attribute values, (see chapter 3 section 2.1). EXCELL has severe worst case problems. A related

structure, HCELL [Tamm81] has worst case problems which are less likely and less severe.

HCELL partitions the space as does EXCELL but does it recursively. EXCELL provides a grid fine enough so that every cell has no more than c points. HCELL may use a coarser grid. Cells with more than c points are handled using another HCELL directory.

Table 1 summarizes this survey. A data structure is considered to be static if its performance can degenerate following updates (and then has to be rebuilt). "Bias" refers to the general scheme in which the space is partitioned. It is a qualitative measure of the degree to which the data structure favors searching on some attributes over others. All of the data structures with low bias can be made more biased but high bias data structures cannot be made less so.

A high bias data structure has poor performance for searching on some attributes. For example, consider the multiple attribute tree of figure 1. It classifies tuples on three attributes, A_0, A_1, A_2: $dom(A_0) = dom(A_1) = dom(A_2) = \{0, 1, 2\}$. To process any 1-match query on A_0 requires the accessing of 3 of the 6 pages. But 6 accesses are required to process any 1-match query on A_2. The problem is that the "efficiency" of the search decreases as the attribute being queried gets farther away from the root. ("Efficiency" is discussed in chapter 3 section 1.6.2.)



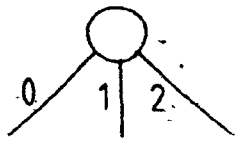


Figure 1. The multiple attribute tree: A high bias multidimensional data structure.

Table 1:

Data structure	Query type	Updating	Bias	Reference
EXCELL	R	D	L	[Tamm80]
HCELL	R	D	L	[Tamm81]
inversions	R	S/D	H	[Knut73]
kd tree	R	S	L	[Bent75a]
K-D-B tree	R	D	L	[Robi80]
Multiple attribute tree	R	S	H	[Kash77, Gopa80]
MDB tree	R	D	H	[Sche81]
Multidimensional clustering	P	S	L	[Liou77]
Multiple key hashing	P	S	H	[Roth74]
Multipaging	R	S/D	L	[Merr78, Merr82]
Quad tree	R	S	L	[Fink74]
Trie	R	D	H	[Rive74]

R: Range

P: Partial Match

S: Static

D: Dynamic

S/D: Static and dynamic versions exist

L: Low

H: High

The most desirable kind of data structure can process range queries, is dynamic and has low bias. Table 1 has several entries of this type. However, each such data structure is based on one particular storage structure for searching. For

example, the kd tree is based on the binary tree but there is no AVL version of the kd tree. In chapter 3 we will discuss a class of data structures for associative searching. A data structure in this class can be created given any data structure that supports random and sequential accessing. Another desirable feature of these data structures is that they provide a certain ordering of the tuples, facilitating algorithms based on merging.

2. Data structures used in implemented systems

The main features of our design for the physical database will be compared with corresponding features of two implemented systems: System R and INGRES. These systems and others were surveyed in 1979 [Kim79].

We concentrate on these systems because they are the most complete centralized systems implemented: for example, none of the other systems mentioned in the survey include a concurrency control system.

Implementations of distributed databases are based on centralized databases so their PDBs contain no new ideas. For example, System R* [Will82], distributed INGRES [Ston79] and SDD-1 [Roth80] are all based on centralized systems.

In this section, the data structures used by System R and INGRES are discussed. (Other features of these systems are discussed in chapter 4.) The data structures used by INGRES are similar to those used in other systems. Most of the following information is from two overviews of System R [Astr76, Blas81] and an overview of INGRES [Ston76].

2.1. The data structures of System R

The data structures of System R are quite complicated. There are several components which we describe below.

2.1.1. Segments

A segment is a set of logical pages. All database objects, (e.g. relations, inversions) are stored in segments. Each object is completely contained in a segment. Each segment may contain several objects.

A page map is used to locate the physical page associated with a given logical page. Physical pages are allocated to segments dynamically. Various operations on segments deallocate pages. Logically sequential pages are kept physically sequential whenever possible: this is easy to ensure when the segment is initialized. When a physical page is updated, the old copy is kept for recovery purposes and a "nearby" physical page is allocated, (e.g. the two pages are on the same cylinder). This mechanism "almost always" works [Lori77] and when it doesn't it is likely to correct itself following updates after out of date pages have been reclaimed.

System R has its own memory management system. Pages are explicitly freed. The least recently used free page is swapped out on a page fault. Segment pages and "blocks" of page maps are managed in separate parts of primary memory.

2.1.2. Relations

Relations are the basic objects of System R. They can be created and destroyed dynamically. Tuples consist of fixed length and varying length fields (attributes). New attributes may be added dynamically without a total reorganization of the storage structure containing the relation. (The value of an added attribute is undefined in a tuple until it is explicitly changed.)

Associated with each tuple is a tuple identifier (TID) which is not visible outside the PDB. Pointers stored in inversions and "links" (see section 2.1.5) refer to TIDs.

The tuples of a relation can be "scanned" in several ways. They can be scanned in a system defined order, according to the sequence of index values in some inversion or in an order specified by a unary link (see section 2.1.5).

2.1.3. Pages of relations

The organization of a System R page (allocated to a relation) is more complicated than in many other database systems. There are several reasons for this:

- A page can store tuples from more than one relation.
- Tuples can grow and shrink (due to the varying length fields).
- Links, (pointers associated with tuples), can be created and destroyed dynamically.

A TID (see section 2.1.2) consists of a logical page address and a byte offset within the page. At the specified location is a pointer to a tuple within the page. This indirection allows tuples to be shifted within the page. Only the intra-page pointer is affected; the TID is unchanged. In case of overflow,

the tuple is replaced on the page by a (shorter) pointer to an overflow record. Thus, given a TID, a tuple can almost always be found in one page access but never more than two.

2.1.4. Images

In System R terminology, an inversion is an "image". Images can be created and destroyed dynamically. The images are stored in B+trees (see [Come79]) residing in the same segment as the relation being indexed. The leaves are doubly linked to support fast sequential access. Each entry in a leaf consists of an index value and the TIDs for all tuples containing the value.

Up to one image per relation can be clustered: tuples with logically sequential index values are stored sequentially in physical storage. This expedites sequential processing using the clustered image.

2.1.5. Links

Links are used to connect tuples from one or two relations into a doubly linked list.

Unary links connect tuples from one relation into an order which does not necessarily correspond to lexicographic ordering (of the attribute values associated with the link).

Binary links strongly resemble DBTG sets [CODA71]. As with images, the decision to include a certain binary link lies with the database administrator but all maintenance is done by System R. Binary links allow the efficient implementation of certain operations. For example, a natural join can be processed very efficiently if a binary link exists on the access set involved in the join. That is, each tuple in one relation will be on the same

link as the tuples in the other relation with matching values. A binary link may be clustered, further increasing the speed of a natural join: the "owner" and "member" tuples will all be physically close.

Links can be created and destroyed dynamically. Addition of a link causes the pages to be repacked when the tuples are placed in the lists since room for the pointers must be allocated.

Obviously, there are many ways to evaluate queries on these data structures. Scans, images, links or some combination of these may be used. An optimizer considers several possibilities and selects the cheapest (based on estimates of the cost of each strategy). The optimizer has been described in [Seli79].

2.2. The data structures of INGRES

The data structures used by INGRES are much simpler than those of System R. This is, in part, due to the decision to implement INGRES using the file system and memory management facilities of the UNIX operating system [Ritc74].

Relations can be created and destroyed dynamically. All attributes are fixed length and new attributes cannot be added dynamically.

Relations are stored in UNIX files. A file consists of 512 byte pages. The concept of a "nearby" page is meaningless in UNIX so physical sequentiality cannot be guaranteed.

Relations can be stored in any of five organizations. Either an "ISAM-like" file or hashing can be used. Raw or compressed data can be stored in either kind of file. The fifth organization

is a randomly ordered sequential file.

The keyed organizations are accessed via TIDs resembling those of System R. If a primary page becomes full then a linear list of overflow records is set up.

Inversions® can be created and destroyed dynamically. No facility resembling the links of System R was included, (one of the reasons has to do with the UNIX file system).

Chapter 3

Data Structures for Range Searching

In this chapter, we describe and analyze a class of "multidimensional data structures" (MDSs) which support the efficient evaluation of range and partial match queries. Our approach will be to transform tuples into integers and store the integers in an "indexed-sequential data structure", (e.g. a binary tree). An indexed-sequential data structure, (ISDS), has the following two properties:

1) Any record can be located by a random access in time f(n), where n is the number of records. (Typically the ISDS is a tree of some kind in which case f(n) = O(log n).)

2) A record's successor (according to some total ordering) can be located by a sequential access in time not exceeding f(n).

Since we are dealing with large files stored on disk, our unit of time measurement will be one disk access.

Many well known file organizations are ISDSs. For example: ISAM [IBM66], the Btree [Baye72] and its variants [Come79]. But traditional hash files which do not preserve order are excluded.

This approach has the following consequences:

- Many existing file systems can be adapted quickly and easily to deal with multidimensional data.
- New ISDSs immediately yield new MDSs.

This chapter is organized as follows. Section 1 describes the "kd trie", a data structure based on the "space" model of a relation, (see chapter 1 section 4). In section 2 the transformation of an ISDS to a certain kind of MDS is explained. This transformation follows directly from an "interpretation" of the kd trie. Section 3 describes a pair of new ISDSs whose performance is at least comparable to that of the Btree, (and is likely to be better). Since they are ISDSs they yield new MDSs.

In sections 2 and 3 we will use the following notation: $\langle s_1 : n_1 \mid s_2 : n_2 \mid \ldots s_m : n_m \rangle$ denotes the string

$$\underbrace{s_1\ s_1 \ldots s_1}_{n_1}\ \underbrace{s_2\ s_2 \ldots s_2}_{n_2} \ldots \underbrace{s_m\ s_m \ldots s_m}_{n_m}$$

where each s_i is a string of one or more characters. If $n_i = 1$ then "$s_i : n_i$" may be abbreviated to "s_i".

1. Multidimensional tries used for range searching

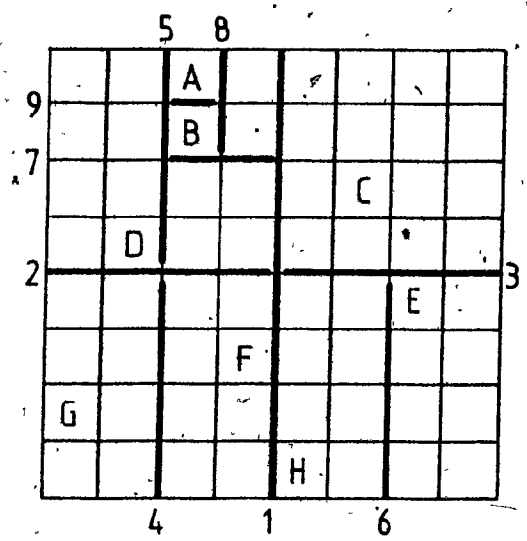
We propose a data structure, the kd trie, for range searching. This data structure is related to the kd tree [Bent75a, Bent79a]. ("kd" is an abbreviation for "k dimensional".) In most cases, the kd trie has superior performance for searching and updating: the time requirements for these operations are at least as good as for the kd tree and the kd trie is smaller, (resulting in fewer accesses to secondary storage). Most importantly, the kd trie leads to the discovery of a class of data structures for range searching, (see section 2).

Most of the material in this section has appeared in [Oren81, Oren82a].

1.1. The kd trie

As discussed in chapter 1 section 4, a k-ary relation can be represented by a k dimensional space of bits in which each tuple is represented by an on bit. We will refer to these on bits as "points". The kd trie is a concise representation of this space.

Figure 1 shows a two dimensional space of bits and the 2d trie



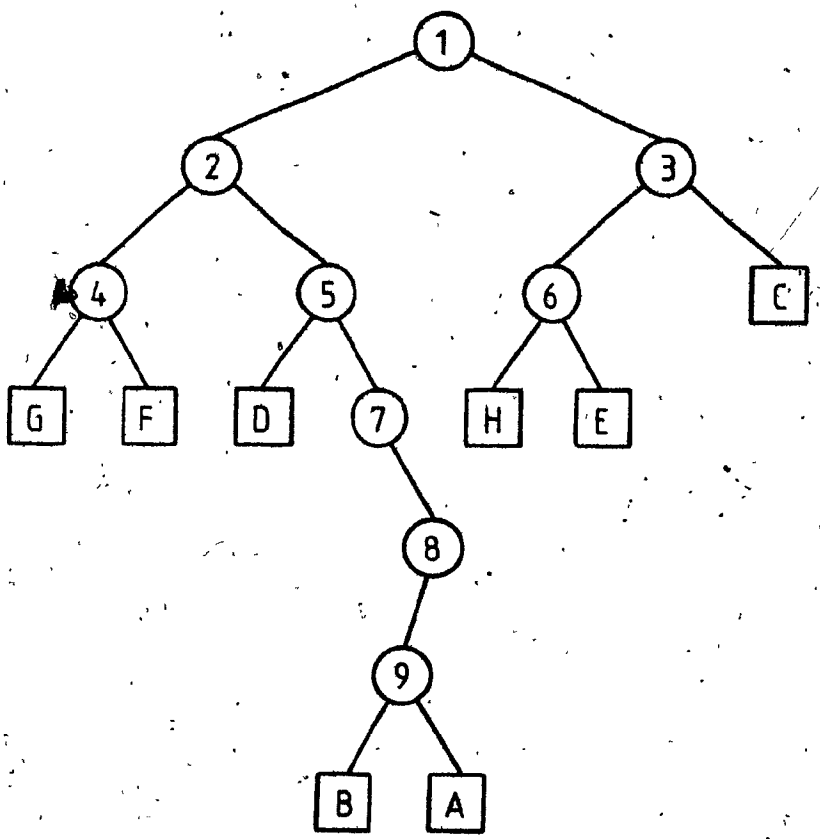


Figure 1. A 2d space and the 2d trie representing it.

representing it, (the tuples are labelled for ease of reference only). The thick lines represent partitions. The numbers associated with these lines refer to the labels of the internal (round) nodes of the 2d trie. The partitions follow a simple pattern: each partition splits a region into two sub-regions of equal size. The orientation of a partition being placed in a region R is perpendicular to that boundary of R which was most recently placed. (For example, partition 7 is perpendicular to partition 5.) A node of the 2d trie represents a region of the space. All nodes on level i split attribute i mod 2, (the root is at level 0). A null link indicates that a sub-region is empty, (i.e. it contains no points). A non-null link points to the sub-trie describing the sub-region or to an external node which stores the tuples corresponding to the points in the sub-region. The generalization to k dimensions is simple: All nodes on level i split attribute i mod k.

The splitting terminates in one of two ways:

1) In a "pure" kd trie, splitting continues as far as possible, that is, until each leaf represents an occupied one bit region. The coordinates of the point are not stored anywhere since they can be derived when the path to the leaf is traversed. (The derivation is similar to a binary search.)

2) In a "hybrid" kd trie, splitting continues until a sub-region contains no more than s points. External nodes store the tuples themselves. (Figure 1 shows a hybrid 2d trie with s = 1.)

Unless stated otherwise, when we say "kd trie" we refer to the hybrid version.

The depth of the pure kd trie is determined by the size of the space representing the relation. If the space contains D bits

then the depth of the pure kd trie is $h = \lceil \log(D) \rceil$. This is also the number of bits required to represent a tuple: each tuple is represented by the path from the root to the leaf corresponding to the tuple. There is no theoretical bound on h but in practice it would be measured by the hundreds or thousands: a tuple requiring more than a few thousand bits to represent is unusual.

No internal node in a hybrid kd trie can be deeper than level h. Knuth [Knut73] has proven that the average depth of a trie is O(log(n)) under the assumption that the keys are uniformly distributed real numbers in [0,1), represented in binary. Devroye [Devr82] has provided a shorter proof of this result and has also shown that the expected depth is O(log(n)) for a very large class of distributions, including all distributions that can arise in practice.

The kd trie is related to the non-homogeneous kd tree [Bent79a] and algorithms for the latter apply to the former. Both data structures can evaluate boolean combinations of range queries (without resorting to a scan of all tuples).

A discriminator is a k-1 dimensional hyperplane which splits one k dimensional region into two regions. The discriminator is perpendicular to the axis which represents the attribute being split. Both kd trees and kd tries select the attributes to be split cyclically, (i.e. 0, 1, ... , (k-1), 0, 1, ... , (k-1), ...), but other methods are possible [Bent76, Bent79a].

The kd tree and kd trie differ in their methods of selecting the position within a region of the discriminator: A discriminator in a kd tree evenly divides the set of points in a region. This strategy results in a well-balanced kd tree. A kd trie discriminator is independent of the data; it evenly divides

a region. (Compare figures 1 and 2.)

A consequence of this difference is that a given set of tuples can be represented by any one of a large number of kd trees but the kd trie is unique: it is completely determined by the data. This observation has two implications concerning the performance of the kd trie relative to that of the kd tree.

1) Since kd tries cannot be rebalanced in any way, (other than by transforming the data), there is nothing that can be done to improve the performance of a bad kd trie. On the other hand, a bad kd tree can always be rebalanced.

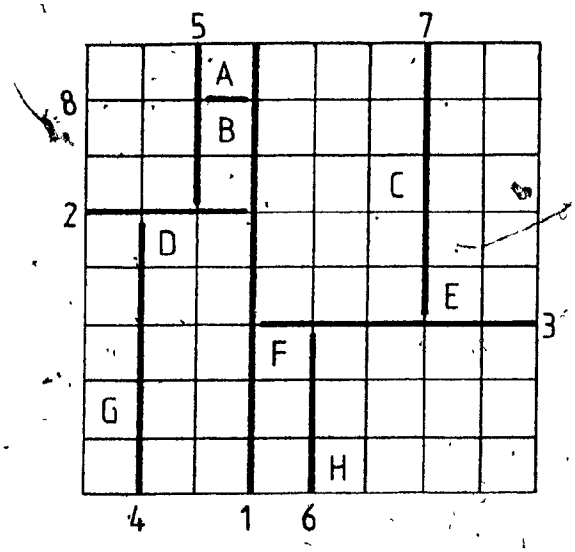
2) Updates of a kd trie cause degeneration only if they lead to a distribution of tuples that determines a bad kd trie, (we describe such tries in section 1.3). Updates can cause a kd tree to degenerate regardless of the tuples contained in the updated relation.

Another difference between the two data structures is that the discriminator values need to be stored in kd tree nodes but not in kd trie nodes. Kd trie discriminators can always be calculated.

The kd trie resembles Hardgraves quatree which is used to represent sets of integers in a given range [Hard76].

1.2. Dynamic operations on the kd trie

The tuples of a hybrid kd trie are stored in external nodes. Each external node has a capacity of s tuples. Insertions and deletions usually affect the external nodes only. When an external node overflows, it is replaced by a chain of nodes, (see figure 3).



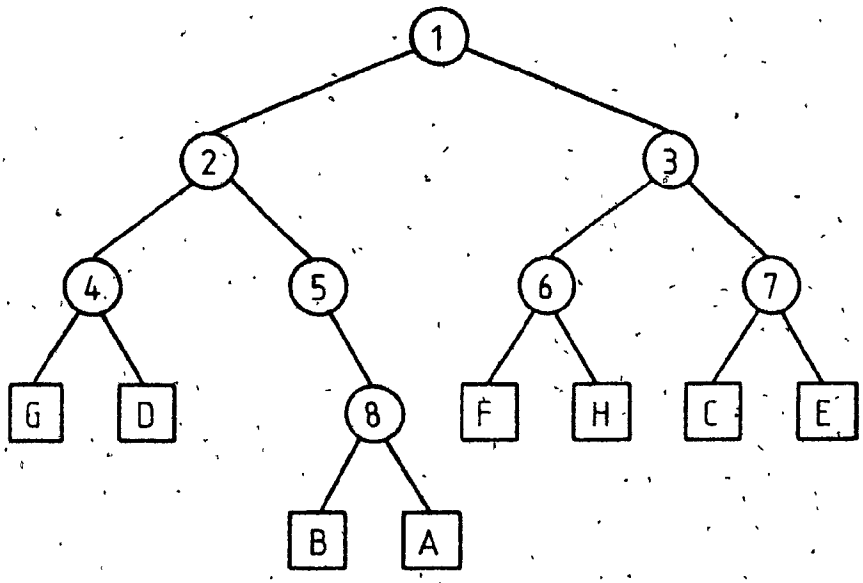


Figure 2. A 2d space and one of the 2d trees representing it.

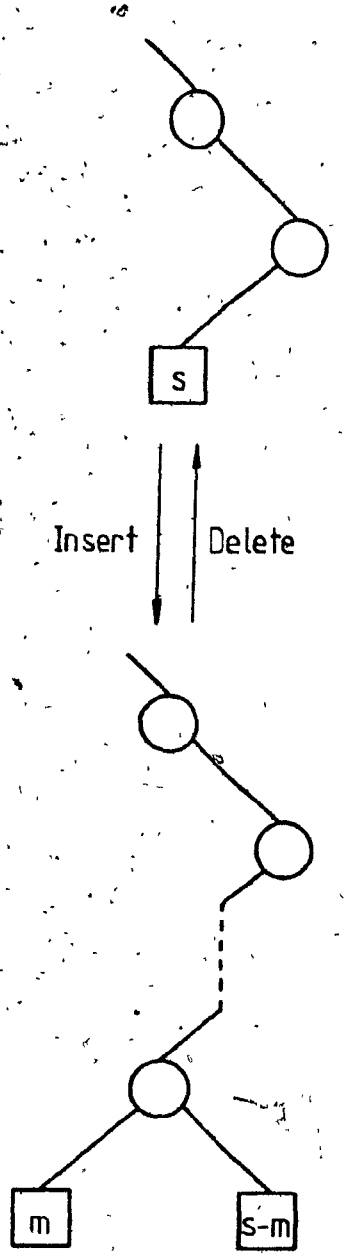


Figure 3. The effect of overflow on an external node. The s tuples split into two groups of size m and s-m, ($0 \leq m < s$). The tuple which caused the overflow will be stored in one of the new external nodes.

Each node in the chain (except the last) corresponds to a bit position whose value is identical in all the tuples of the external node which overflowed. The tuples do not all agree in the bit corresponding to the last node of the chain. Thus, the tuples of the external node which overflowed are split into two groups.

This chain has fewer than h nodes (where h is the number of bits needed to represent a tuple). If each bit position needed to resolve overflow has an even chance of being 0 or 1 in each tuple then we can easily calculate the expected length of the overflow chain, (see section 1.5.1). The expected length is bounded by 28/9 and as s increases, it approaches 1.

When, due to deletions, two fraternal external nodes become empty enough to be combined, the process is reversed.

The kd trie grows and shrinks gracefully. The kd tree, on the other hand, cannot be updated without some problems. It can degenerate and then rebalancing is required to avoid poor performance.

1.3. Performance of the kd trie

Tables 1 and 2 list the average and worst case space and time requirements of the hybrid kd trie. The requirements of the non-homogeneous kd tree are given for comparison. (The results concerning the kd tree are from [Bent75a] except where stated otherwise.) The following abbreviations are used in tables 1 and 2:

h: Number of bits required to represent a tuple.

F: Number of tuples satisfying the query.

k: Number of dimensions.

L: Lifespan of one rebalancing.

n: Number of tuples.

t: Number of attributes being queried, $t < k$.

Table 1: Average case	hybrid kd trie	non-homogeneous kd tree
Size	O(n)	O(n)
Creation	O(n log(n)) *	O(n log(n)) [Bent79a]
Exact Match Query	O(log(n))	O(log(n))
Partial Match Query	$O(tn^{1-t/k})$	$O(tn^{1-t/k})$
Range Query	O(log(n)+F)	O(log(n)+F) [Bent79a]
Insertion	O(log(n))	O(log(n) + n log(n)/L)
Deletion	O(log(n))	O(log(n) + n log(n)/L)

* The expected depth of a hybrid trie is O(log(n)) [Knut73].

Table 2: Worst case	hybrid kd trie	non-homogeneous kd tree
Size	O(hn)	O(n)
Creation	O(hn)	O(n log(n)) [Bent79a]
Exact Match Query	O(h)	O(log(n))
Partial Match Query	$O(hn^{1-t/k})$	$O(tn^{1-t/k})$
Range Query	O(hn)	$O(tn^{1-1/k})$ [Lee77]
Insertion	O(h)	O(log(n) + n log(n)/L)
Deletion	O(h)	O(log(n) + n log(n)/L)

The kd tree costs for insertion and deletion include the term

$$n \log(n) / L .$$

This term represents the cost of rebalancing, (the numerator), amortized over the "lifespan" of that operation (the denominator, L, which can also be thought of as "mean time between rebalancings"). For a static file, L is infinitely large.

The average case results for the kd trie have derivations similar to those for the non-homogeneous kd tree. They appear in section 1.5.2.

Two points concerning random kd tries should be emphasized.

1) The average case results of Knuth [Knut73] and Devroye [Devr82] apply to (1d) tries. Our application of these results to kd tries is valid since the two types of tries are indistinguishable: a trie can be "interpreted" as a kd trie given k.

2) The notions of "randomness" for tries and trees are not the same. In [Devr82] Devroyé regards the data stored in tries as real numbers in [0, 1) (selected from a given distribution) expressed in binary. In [Bent75a] Bentley assumes that all permutations of the data are equally likely. It would seem then that the average case results for tries and trees cannot be compared. However, the two notions can be reconciled: The tree results hold for all distributions of data (since the results are independent of the distribution). Similarly, the trie results hold for all permutations of the data.

The worst case results for the kd trie have trivial derivations; they appear in section 1.5.3.

In the average case, the kd trie is at least as good as the kd tree. We will show in section 1.4 that the kd trie has a more compact representation. Thus, for large relations on secondary

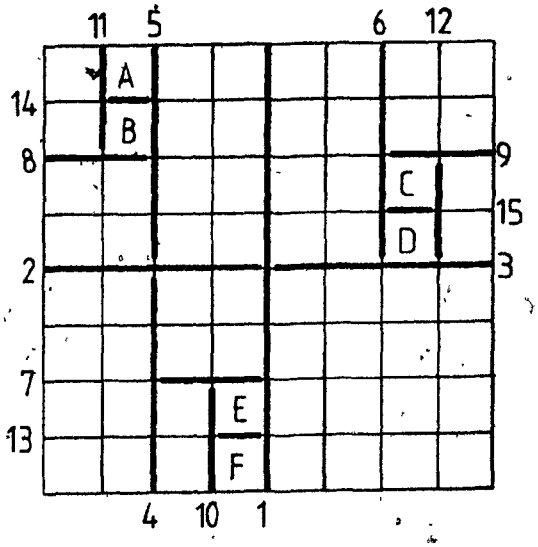
storage, the kd trie will out-perform the kd tree. In the worst case, the kd tree is superior. To encounter the worst case times for the kd trie, the data being queried must have many clusters of at least s+1 points each, (where s is the capacity of an external node), and each cluster must be confined to certain small regions, (those regions represented by the kd trie nodes at or slightly above level h-log(s)). (See figure 4.) The probability of this occurring can be decreased by increasing s. s cannot be made arbitrarily large: in practice, considerations such as memory capacity and page size place limits on s.

1.4. Implementation of the kd trie

In this section we describe a storage structure for the hybrid kd trie. Because internal nodes do not store discriminators, an extremely compact representation of the kd trie is possible. A search on this structure yields a set of pointers to external nodes which hold the tuples satisfying the query. Each external node has a capacity of s tuples.

An internal kd trie node can be represented by two bits: The only information needed is whether each link is null. The addressing information of a link is unnecessary if an "implicit" representation is used.

Each internal node is, then, represented by two bits, one corresponding to each link. A bit is on iff the corresponding link is not null. The nodes of each level are stored consecutively. We will refer to this storage structure as the "bit string" representation. The pure kd trie corresponding to the data of figure 1 would be represented by the following bit



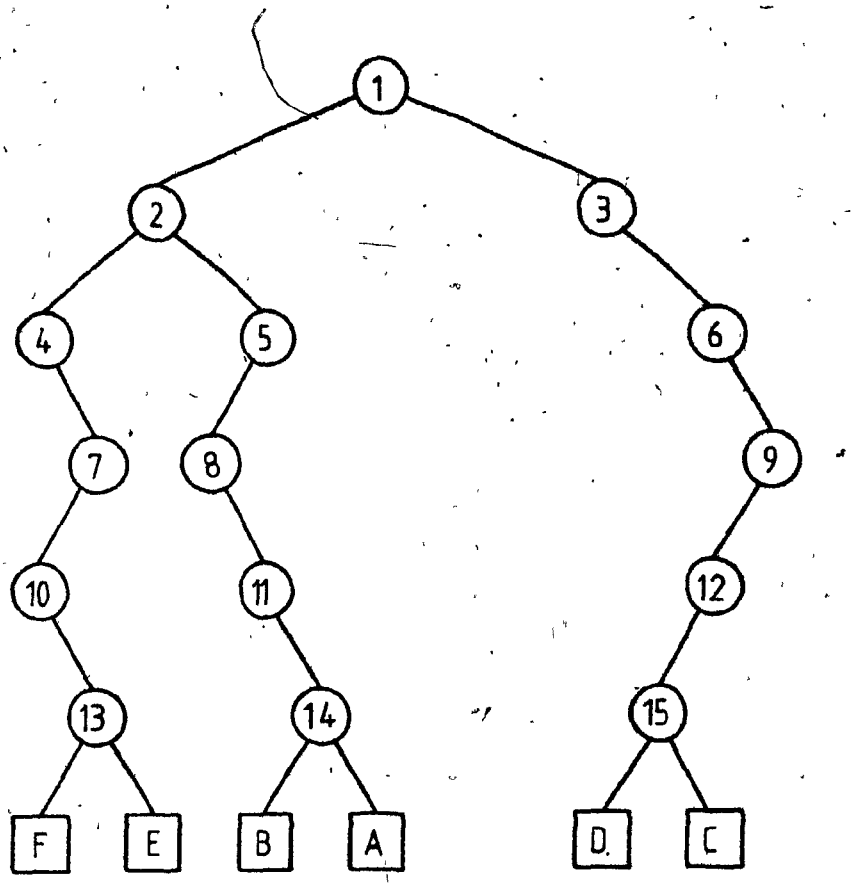


Figure 4. An example of a "bad" distribution of data and the corresponding worst case 2d trie.

string: (A "0" indicates a null link, a "1" indicates a non-null link. Beneath each leaf's links are the labels of the tuples from figure 1.)

level	bit string
0	11
1	11 11
2	11 11 11 10
3	10 01 10 01 10 01 10
4	10 01 01 10 10 10 01
5	01 10 10 11 10 01 01
	G F D BA H E C

Now consider the ith on bit of level j which is at position C_j. The node which is pointed to by the link represented by bit C_j has a displacement of 2(i-1) bits in the next level. Unfortunately, finding that the C_j th bit was the ith on bit involved scanning C_j bits. This scanning must be done for every level and is expensive. However, by organising the bit string into "blocks", the number of bits scanned in each level is reduced to an arbitrary constant (dependent on the block size).

Figure 5 shows a pure kd trie organized into blocks. Each block contains for several consecutive levels, a contiguous segment of each level's bit string. In addition, either all the children of the nodes of segment x are in the same block as x or none of them are. Now instead of scanning levels we scan segments of levels.

Extra storage is required to support blocking. A small, constant number of integers associated with each block provides information which permits linking across block boundaries.

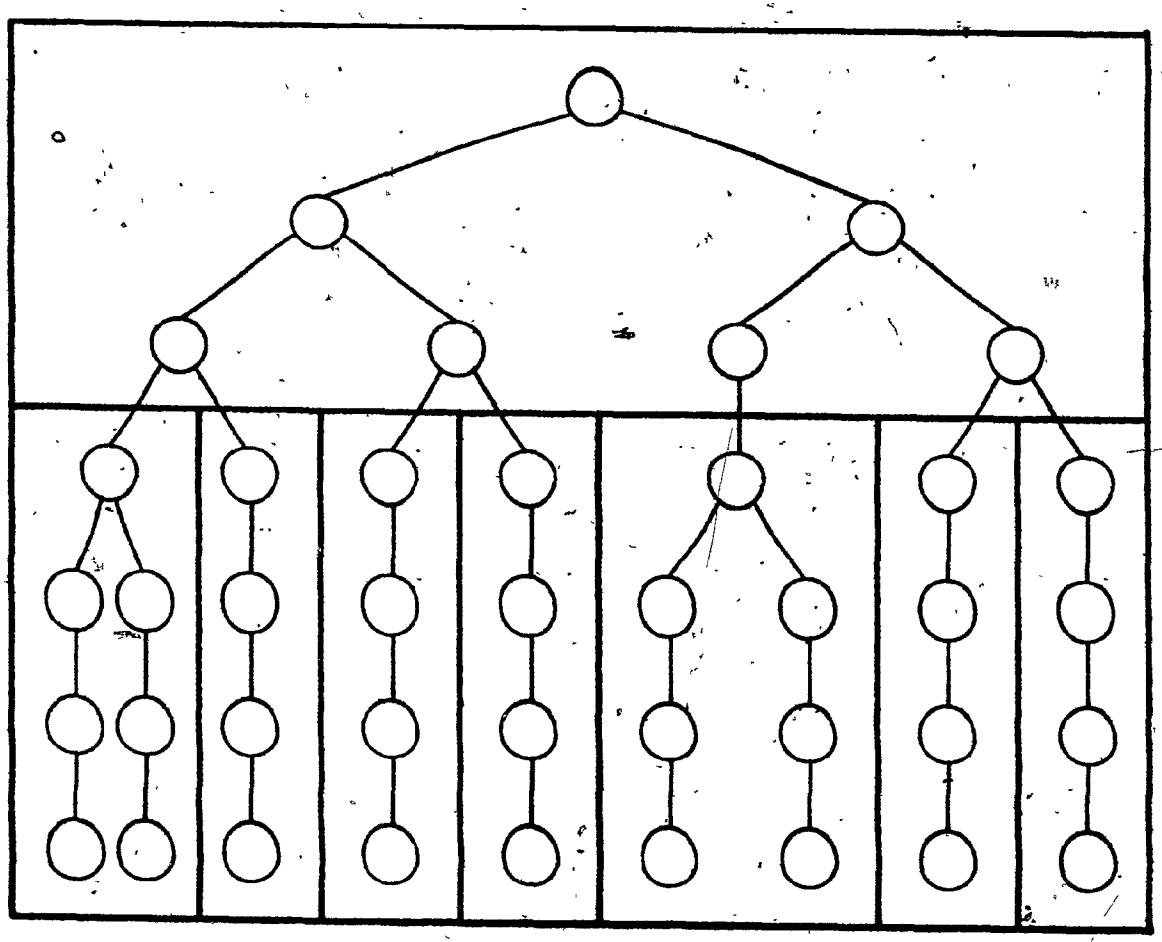


Figure 5. Partitioning of a pure trie. Block capacity is seven nodes.

The storage scheme described is suitable for a pure kd trie which stores all leaves on the same level. In the hybrid version, a leaf may appear on any level. A simple modification of the bit string representation will handle a hybrid kd trie.

No internal node in a kd trie has two null links so the only possible nodes in the bit string representation are "01", "10" and "11". The unused combination, "00", can be used to indicate a link from an internal node to an external node. We call such a node a terminator. The terminator is followed in the bit string by a pointer to an external node. Blocking and implicit addressing are possible even with terminators and pointers present in the bit string. The kd trie of figure 1 would be represented by the following bit string:

level	bit string
0	11
1	11 11
2	11 11 11 00[->C]
3	00[->G] 00[->F] 00[->D] 01 00[->H] 00[->E]
4	10
5	11
6	00[->B] 00[->A]

Note: "00[->X]" indicates a terminator followed by a pointer to the page containing X, (occupying 2 or 3 bytes typically).

This storage structure allows efficient implementation of the update operations of section 1.2: Blocking ensures that the

changes to a level's bit string are localized (to the block containing the segment being changed).

Given this storage structure, a kd trie almost certainly occupies much less space than does a kd tree (even if the latter uses an "implicit" representation as suggested in [Bent79a]). (See table 3.) A major benefit of this compactness is that a large portion of the kd trie can fit in primary memory at once. Compared to the kd tree, relatively few accesses to secondary storage would be required. Again, the worst case corresponds to a highly unlikely clustering of points.

Table 3.
Space requirements

	hybrid kd trie	non-homogeneous kd tree
Node size	2 bits	≃10 bytes [Bent79a] = 80 bits
Number of pointers to external nodes	O(n/s)	O(n/s)
Number of nodes (expected)	O(n)	O(n)
Number of nodes (worst case)	O(hn)	O(n)

1.5. Kd trie analysis

We now give the derivations of the kd trie results in sections 1.2 and 1.3.

1.5.1. The expected length of the overflow chain

We assume that each bit position needed to resolve overflow has an even chance of being 0 or 1 in each tuple. Then the probability that r bit positions do not suffice to resolve

overflow, (i.e. the s+1 tuples all agree in the bit corresponding to the ith node of the chain, i=1, 2, ... , r), is $p_r=2^{-rs}$. The expected length of the chain is

$$E = \lim_{h\to\infty} \sum_{r=1}^{h} r\, p_{r-1} (1 - p_r)$$

since h can be arbitrarily large. Thus

$$E = \frac{1}{(1-2^{-s})^2} - \frac{1}{(1-4^{-s})^2}$$

E is bounded by 28/9, (obtained at s = 1), and as s increases, E approaches 1.

1.5.2. Average case kd trie results

In analyzing the quad tree, Bentley and Stanat [Bent75b] derived results for the "perfect" quad tree. They then compared the results with the behaviour of randomly built quad trees and found close agreement.

We will consider an analogous variation of the kd trie: the perfect kd trie. The perfect kd trie has 2^i nodes on level i, i = 0, 1, ... , log(n) - 1, (assuming $n = 2^j$ for integer $j \geq 0$).

A perfect kd trie has depth log(n). According to [Devr82] the depth of the average ld trie is slightly above this. This theoretical result is demonstrated by the data of Fredkin [Fred60]. Furthermore, a kd trie has the same size and shape as a ld trie: The kd trie is essentially a ld trie whose keys are permutations of the bits of the k-tuples. These facts, considered together, suggest that the perfect kd trie is a good

model for the average kd trie.

1.5.2.1. Size of the kd trie

The perfect kd trie has 2^i nodes on level i, i = 0, 1, ... , log(n) - 1. The size of the perfect kd trie is

$$S = \sum_{i=0}^{\log(n)-1} 2^i$$

$$= n - 1 = O(n)$$

1.5.2.2. Cost of exact match queries

An exact match query specifies a single value for each attribute. At each level, exactly one node is examined (until a null link or external node is reached). The number of nodes visited is at most

$$V_{EM} = O(\log(n))$$

1.5.2.3. Cost of partial match queries

A partial match query specifies values for t < k attributes. Suppose (pessimally) that attributes 0, 1, ... , k-t-1 are not specified and that the remaining attributes, k-t, k-t+1, ... , k-1 are specified in the query.

Since there is no selection on attributes 0 through k-t-1, all the children of all the nodes on these levels must be visited. In a perfect kd trie, each internal node has two children. Thus the

number of nodes visited in the first k-t levels is

$$V_{PM}(k-t) = 1 + 2 + \ldots + 2^{k-t-1}$$

In the next t levels, selection occurs on each level and only one child of each node will be visited. The contribution from each of these t levels is 2^{k-t} nodes. The total number of nodes visited in the first k levels is

$$V_{PM}(k) = (1 + 2 + \ldots + 2^{k-t-1}) + t2^{k-t}$$

Now, each of the sub-tries whose root was visited in the kth level can be analyzed as if it were the root of a kd trie. Thus, in the second "band" of k levels, $2^{k-t}V_{PM}(k)$ nodes are visited. There are log(n)/k bands in the perfect kd trie so that the total number of nodes visited in a partial match query is bounded by

$$V_{PM} = (1 + 2 + \ldots + 2^{k-t-1}) + t2^{k-t} + 2^{k-t}[(1 + 2 + \ldots + 2^{k-t-1}) + t2^{k-t}] + \ldots + 2^{(\frac{\log(n)}{k} - 1)(k-t)}[(1 + 2 + \ldots + 2^{k-t-1}) + t2^{k-t}]$$

$$= \left[\sum_{i=0}^{\frac{\log(n)}{k} - 1} 2^{i(k-t)}\right]\left[\sum_{i=0}^{k-t-1} 2^i + t2^{k-t}\right]$$

$$= (2^{k-t} - 1 + t2^{k-t}) \frac{n^{1-t/k} - 1}{2^{k-t} - 1}$$

$$= O(tn^{1-t/k})$$

This is an upper bound because we assumed an arrangement of attributes that maximized V_{PM}.

1.5.2.4. Cost of range queries

We will first obtain results for a perfect 2d trie and then generalize to k dimensions. A 2d range query specifies a range of values for each attribute. In our view of a relation, this corresponds to specifying a range of values on each axis. The query dimensions are x and y where x, y $\in$ [0, 1] denote the portion of each axis that is covered by the query. The number of nodes visited is bounded by (cf. [Bent75b])

$$V_R = \sum_{i=0}^{\frac{\log(n)}{2} - 1} [(x2^i+1)(y2^i+1) + (x2^{i+1}+1)(y2^i+1)]$$

$$= xy(n-1) + (3x + 2y)(n^{1/2}-1) + \log(n)$$

Because the 2d trie is perfect, xyn is the number of tuples accessed by the query. This quantity was called F in section 4. Thus

$$V_R = O(\log(n) + F)$$

We now generalize to k dimensions. The query has dimensions $x_0, x_1, \ldots, x_{k-1}$, $x_i \in [0, 1]$, $i = 0, 1, \ldots, k-1$. The number

of nodes visited is

$$V_R = \sum_{i=0}^{\frac{\log(n)}{k}-1} \sum_{j=0}^{k-1} \left[\prod_{m=0}^{j-1} (x_m 2^{i+1}+1) \right] \left[\prod_{m=j}^{k-1} (x_m 2^{i}+1) \right]$$

But $(x_m 2^{i+1} + 1) < 2(x_m 2^i + 1)$, so

$$V_R < \sum_{i=0}^{\frac{\log(n)}{k}-1} \sum_{j=0}^{k-1} 2^{j-1} \prod_{m=0}^{k-1} (x_m 2^i+1)$$

$$= (2^{k-1}) \sum_{i=0}^{\frac{\log(n)}{k}-1} \prod_{m=0}^{k-1} (x_m 2^i+1)$$

The terms dominating V_R are

$$(n-1) \prod_{m=0}^{k-1} x_m = O(F)$$

and (from the precise formula for V_R)

$$\sum_{i=0}^{\frac{\log(n)}{k}-1} k = O(\log(n))$$

The number of nodes visited in a range query of a perfect kd trie is

$$V_R = O(\log(n) + F)$$

1.5.2.5. Creation, insertion and deletion costs

All of these operations perform exact match queries. An insertion traces a path of length O(log(n)) through the kd trie. A deletion (in which all attribute values of the tuple being deleted are known) must perform an exact match query to locate the tuple. The actual updating of the kd trie retraces all or, (more likely), part of this path. Thus, insertion and deletion cost O(log(n)) each. Finally, a kd trie can be created by performing n insertions for a cost of O(n log(n)).

1.5.3. Worst case kd trie results

The worst case kd trie has been described in section 1.3. Figure 6 shows the profile of such a kd trie and illustrates the terms "head" and "tail". The first O(log(n)) levels comprise the head of the kd trie. This is the part containing all nodes with two children. The tail consists of long chains of nodes which do not bifurcate. Resolution of points very close together in the space occurs near the bottom of the tail.

1.5.3.1. Size of the kd trie

The width of the tail is n/(s+1) (where s is the number of tuples in each external node). The length of the tail is O(h-log(n)) nodes and it is much longer than the head. The tail determines the size of the kd trie:

$$S = O(hn)$$

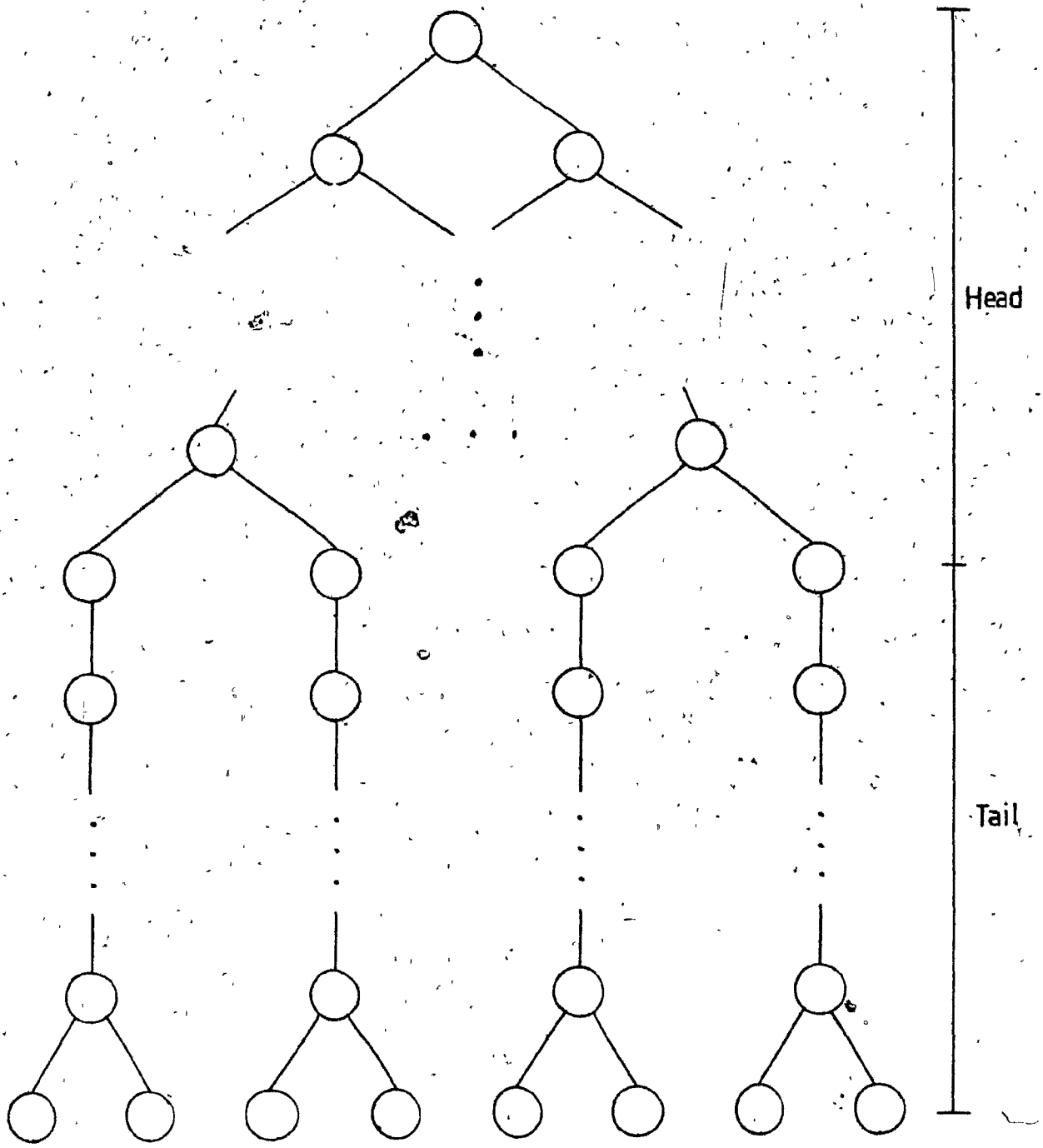


Figure 6. Head and tail of a worst case trie.

1.5.3.2. Cost of exact match queries

At each level, exactly one node is examined (until a null link or external node is reached). The number of nodes visited is at most

$$V_{EM} = O(h)$$

1.5.3.3. Cost of partial match queries

Let v_i denote the number of nodes visited on level i of the kd trie during a partial match query. Then, in the worst case

$$v_i = \begin{cases} 2v_{i-1}, & \text{level } i-1 \in \text{head and attribute } i-1 \text{ is queried.} \\ v_{i-1}, & \text{level } i-1 \in \text{head and attribute } i-1 \text{ is not queried.} \\ v_{i-1}, & \text{level } i-1 \in \text{tail.} \end{cases}$$

$v_{\log(n)}$ is the number of nodes examined in the first level of the tail.

$$V_{\log(n)} = (2^{k-t})^{\frac{\log(n)}{k}} = n^{1-t/k}$$

In the worst case, the chain descending from each of the nodes visited on level log(n) must be completely traversed. This cost is

$$V_{PM}(\text{tail}) = O(hn^{1-t/k})$$

From the average case results

$$V_{PM}(\text{head}) = O(tn^{1-t/k})$$

nodes have been examined in the head. The total number of nodes visited is

$$V_{PM} = V_{PM}(\text{head}) + V_{PM}(\text{tail}) = O(hn^{1-t/k})$$

1.5.3.4. Cost of range queries

Given a pessimal combination of data and query, the cost of a range query can be O(hn):

The query boundaries coincide with partitions corresponding to nodes near the bottom of the kd trie. Furthermore, O(n) points are immediately adjacent to these boundaries (on one side or the other) so that the entire tail, containing O(hn) nodes, must be examined to determine whether each point is in the query region.

1.5.3.5. Creation, insertion and deletion costs

Proceeding as in the average case analysis, insertion and deletion have the same cost as does an exact match query: O(h). The kd trie can be built by n insertions at a cost of O(hn).

1.5.4. Number of pages accessed

The set of pages can be viewed as the first level of the k+1st band of levels in the perfect kd trie.

1.5.4.1. Partial match queries

The number of pages accessed is

$$P_{PM} = 2^{\frac{\log(p)}{k}(k-t)} = p^{1-t/k}$$

where p is the number of pages storing the data.

1.5.4.2. Range queries

The number of pages accessed is

$$P_R = \prod_{m=0}^{k-1} (x_m p^{1/k} + 1)$$

$$= O(V_p)$$

where V is the "volume" of the query: the fraction of the space covered by the query.

1.6. Experimental results

An implementation of the kd trie was programmed in C and experiments were carried out on a PDP-11/45 running UNIX V6. The motivation for experimentation is that our analysis assumed a particular distribution of data: that the data yields a "perfect" kd trie. This corresponds to a distribution as shown in figure 7: the space is partitioned into cells and each cell contains one point. It was hoped that this would adequately model a uniform distribution of data. This strategy is similar to that of Bentley and Stanat in their analysis of the quad tree [Bent75b].

Our first experiment, a "control", involved a 2d trie storing points in a 1024 x 1024 space. I.e. $dom(A_0) = dom(A_1) = \{0, \ldots, 1023\}$. Page capacity was 10 tuples. Various range and partial match queries were evaluated on files of 100, 200, ... , 3000 tuples. For all experiments, attribute values were generated using a random number generator with uniform distribution. For each query we measured the number of internal node visits, number of data pages accessed and efficiency (discussed below). In addition storage utilization of the data pages was measured.

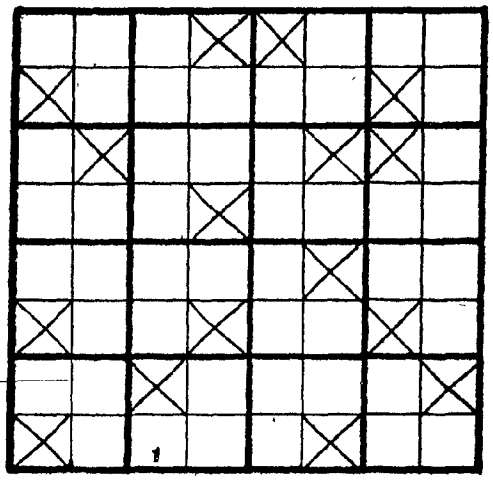


Figure 7. Distribution of data yielding a perfect kd trie.

Next, the effects of varying dimension and page capacity were studied.

1.6.1. Number of internal node visits

1.6.1.1. Partial match queries

The number of internal node visits for a partial match query in a perfect kd trie is

$$V_{PM} = (2^{k-t} - 1 + t2^{k-t}) \frac{n^{1-t/k} - 1}{2^{k-t} - 1}$$

(see section 1.5.2.3). Recall that this is an upper bound for all partial match queries. It is exact (for a perfect kd trie) for the pessimal case in which the last t attributes are queried. Table 4 compares the results of experiments with the expected values. The values are in close agreement.

Table 4.

number of tuples	number of attributes	number of attributes queried	V_R (obs.)	V_R (est.)
428	2	1	55	59
289	3	1	91	100
289	3	2	29	28
143	4	1	79	86
143	4	2	40	40
143	4	3	17	17

1.6.1.2. Range queries

An upper bound for the number of internal node visits for a range query in a perfect kd trie was derived in section 1.5.2.4. The formulae for k = 2, 3 and 4 are, respectively:

$$V_{R_2} = x_0x_1(n-1) + (3x_0+2x_1)(n^{1/2}-1) + \log(n)$$

$$V_{R_3} = x_0x_1x_2(n-1) + (7x_0x_1+5x_0x_2+4x_1x_2)\frac{n^{2/3}-1}{3} + (5x_0+4x_1+3x_2)(n^{1/3}-1) + \log(n)$$

$$V_{R_4} = x_0x_1x_2x_3(n-1) + (15x_0x_1x_2+11x_0x_1x_3+9x_0x_2x_3+8x_1x_2x_3)\frac{n^{3/4}-1}{7} + (11x_0x_1+9x_0x_2+7x_0x_3+8x_1x_2+6x_1x_3+5x_2x_3)\frac{n^{1/2}-1}{3} + (7x_0+6x_1+5x_2+4x_3)(n^{1/4}-1) + \log(n)$$

Tables 5, 6 and 7 compare the results of experiments with the expected values. Recall that this is an upper bound (for the perfect kd trie); it is a poor estimate.

Table 5.

n = 428

x_0	x_1	V_R (obs.)	V_R (up. bd.)
1/2	1/2	159	165
1/8	1/2	61	63
1/8	1/32	18	19
1/32	1/32	11	12

Table 6.

n = 289

x_0	x_1	x_2	V_R (obs.)	V_R (up. bd.)
1/4	1	1	127	226
1/4	1	1/4	67	103
1/4	1/4	1/16	24	31
1/4	1/16	1/16	14	21

Table 7.

n = 143

x_0	x_1	x_2	x_3	V_R (obs.)	V_R (up. bd.)
1	1/4	1	1	83	264
1/4	1/4	1	1	44	114
1/2	1/2	1/2	1/2	97	97
1/4	1/4	1/4	1/4	26	26
1/4	1/4	1/4	1/16	25	21

1.6.2. Efficiency and number of pages accessed

The efficiency of a search measures the amount of work done in evaluating a query relative to the minimum possible. For example, Robinson, in analyzing experiments on the K-D-B tree [Robi81] defines it to be

$$\frac{\frac{N'}{N} P}{P'}$$

where N is the number of tuples, N' is the number of tuples satisfying the query, P is the number of pages in the file and P' is the number of pages accessed to retrieve the required tuples. The numerator is the minimum number of pages that could be accessed in evaluating the query, (assuming a load factor of 100%). We now discuss various factors that have some effect on efficiency and on the absolute number of pages accessed.

1.6.2.1. Volume of the query

The volume of a range query is the fraction of the space covered by the query. A consistent observation, in all experiments, was that as volume decreased, so did efficiency. Table 8 shows the data for queries of a given aspect ratio. That is, the ratio of the lengths of the sides stays constant but the volume is allowed to vary.

The reason for this is as follows: pages whose tuples lie inside the query, ("internal" pages), contribute all of their tuples to the result. Pages whose tuples lie in a region overlapping a boundary of the query, ("boundary" pages), contribute only some of their tuples. As the volume decreases, the fraction of the latter type of page increases, resulting in lower efficiency. The low efficiency at low volumes is not a serious problem since only a small fraction of the pages are actually accessed.

Table 8.

n = 428, aspect = 1:4.

volume	efficiency	fraction of pages accessed
1/4	0.88	28%
1/16	0.67	9%
1/256	0.29	1%
1/1024	0.17	1%

1.6.2.2. Aspect ratio

From the result of section 1.5.4.2, the aspect ratio, ($x_0 : x_1$ for 2d data), should have an effect on the number of pages accessed in evaluating a range query. For k = 2

$$P_R = x_0 x_1 n + (x_0 + x_1) n^{1/2} + 1$$

It is the $x_0 + x_1$ term that is affected by the ratio, (for a fixed volume V, $x_0 x_1 = V$). Table 9 shows the observed results and the estimates.

Table 9.

n = 428, volume = 1/1024.

x_0	x_1	pages read (obs.)	pages read (est.)
1/64	1/16	2.7	3.0
1/16	1/64	3.7	3.0
1/32	1/32	2.7	2.7
1/1024	1 *	16.0	22.1
1	1/1024 *	25.0	22.1

* Partial match queries

1.6.2.3. File size and page capacity

It was noticed that efficiency increased with file size. This is not due to pages becoming more densely filled: load factor hovers around 70% as the file grows from 100 to 3000 tuples, (see section 1.6.2.4).

The explanation has to do with the number of boundary pages relative to the number of internal pages. As the file grows, the number of internal pages grows more quickly than the number of boundary pages (for a given query). This explains why the trend was not so pronounced for partial match queries: there are no boundary pages and this keeps the efficiency low. (See table 10.)

Page capacity has an effect on efficiency for a similar reason. A file of a given size is stored on fewer pages if page capacity increases but this is due to the smaller number of pages which results in relatively fewer internal pages. Of course, the decrease in efficiency is accompanied by a smaller absolute

number of page reads since the pages are larger.

Table 10.

Efficiency

X_0	X_1	n = 500	n = 1000	n = 2000	n = 3000
1/4	1	0.67	0.79	0.82	0.88
1/2	1/8	0.43	0.52	0.60	0.61
1/16	1/16	0.21	0.31	0.36	0.40
1/1024	1 *	0.01	0.03	0.02	0.03
1	1/1024 *	0.00	0.01	0.01	0.01

* Partial match query

1.6.3. Load factor

The load factor was observed to be fairly constant at about 70% under a wide variety of conditions. As points were added in the control experiment, load factor was measured after every 100 insertions. The range was from 67% to 73%. At the end of the experiment, (3000 insertions), the load factor was 71%.

For 3 and 4 dimensional data, with 2000 and 1000 tuples resepectively, the load factors were 70% and 69%.

Load factor varied with page capacity but no pattern was discernible, (see table 11).

Table 11.

capacity	load factor
10	71%
20	67%
30	73%
40	64%
50	77%

2. A class of data structures for range searching

In this section we describe a class of data structures for range searching. This class includes a type of kd tree whose balance can be maintained efficiently, (unlike Bentley's kd tree [Bent75a, Bent79a]), and a multidimensional Btree which is simpler than Robinson's K-D-B tree [Robi81] and Scheuermann and Ouksel's MDB tree [Ouks81, Sche82].

Our approach is to transform the tuples into integers which will be stored in a "one dimensional" data structure such as a binary tree, (i.e. the multidimensional tuples are ordered in a way to be described below).

We require that the underlying data structure be indexed-sequential: our data structures are multidimensional generalizations. The advantages of this approach have been discussed in the introduction of this chapter.

The class of indexed-sequential data structures includes some of the most widely used file organizations: e.g. ISAM [IBM66], the Btree [Baye72] and its variants [Come79]. "Indexed-sequential data structures" (ISDSs) were defined in the

introduction of this chapter. This section, with a few changes, appears as [Oren82b].

2.1. The z ordering

An inorder traversal of the kd trie of figure 8 corresponds to the alphabetical ordering of the tuple labels A, B, C, D, E, F, G, H, I. This motivates the total ordering which can be defined for any set of (multidimensional) tuples: the z order.

A kd tree (for example) could not be used to define such an ordering because a given kd tree representing a set of tuples is not unique. The inorder traversals of two such kd trees are not necessarily the same.

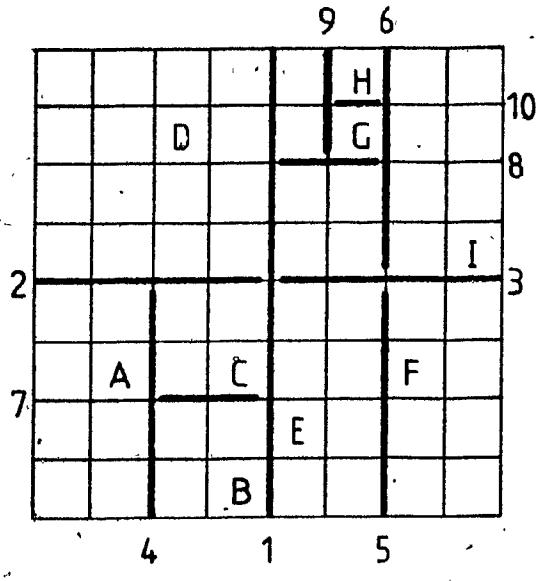
The kd trie provides an objective ordering, (i.e. an ordering dependent on the data only): if tuple t precedes t' in the inorder traversal of a kd trie containing both tuples, then t precedes t' in z order, ($t \leq_z t'$).

The z ordering of tuples can be seen as a path, the "z curve", which passes through all the bits in the space. Figure 9 demonstrates the z curve in 2 dimensions. The path starts in the lower left corner and ends in the upper right corner.

Suppose that each of the domains (from which the attribute values are drawn) contain 2^d elements. Then, if the attribute values of $P = [P_0, \ldots, P_{k-1}]$ are represented in binary, $P = [p_{00}p_{01}\ldots p_{0,d-1}, \ldots, p_{k-1,0}p_{k-1,1}\ldots p_{k-1,d-1}]$ where p_{ij} is the jth bit of P_i's binary representation. The z order is defined by

$$P \leq_z Q \iff \text{shuffle}(P) \leq \text{shuffle}(Q)$$

where $\text{shuffle}(X) = x_{00}x_{10}\ldots x_{k-1,0}x_{01}x_{11}\ldots x_{k-1,1}\ldots x_{0,d-1}x_{1,d-1}\ldots x_{k-1,d-1}$ is the integer created by the concatenation of the



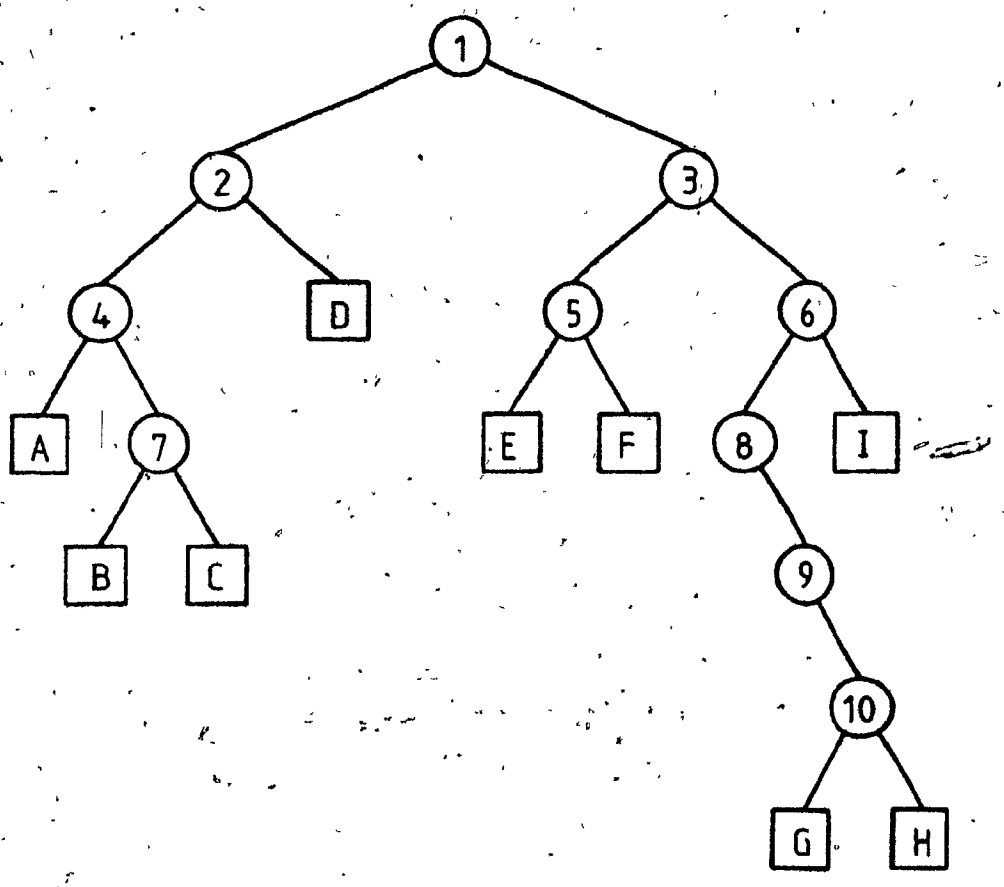


Figure 8. A 2d space and the 2d trie representing it.

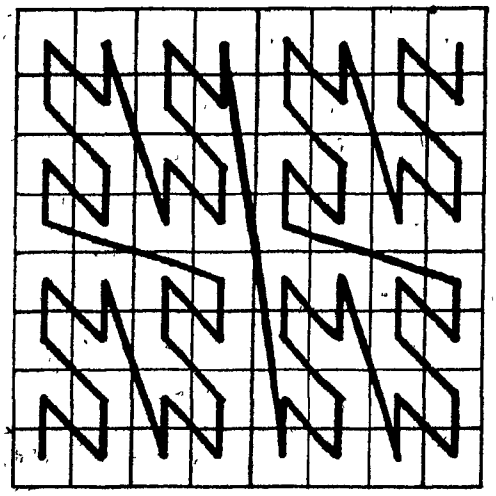


Figure 9. The z curve in 2 dimensions.

indicated bits. Thus point A of figure 8 corresponds to the tuple $[001_2, 010_2]$ and shuffle(A) $= 000110_2 = 6$. Point B is the tuple $[011_2, 000_2]$ and its shuffle value is $001010_2 = 10$, so A $\leq_z$ B.

The kd trie is essentially a trie storing shuffled tuples. The trie is an ISDS. We will show that any ISDS storing shuffled tuples can be used for the evaluation of range queries.

While z ordering can be defined in terms of an inorder traversal of the kd trie, the kd trie does not have to be constructed to generate z ordered tuples. The function shuffle (given below) performs the interleaving of the bits. The hash function used by Tamminen in EXCELL [Tamm80] is equivalent to shuffle.

In the example given above, the interleaving was performed cyclically: The ith bit of shuffle(t) is from attribute i mod k of tuple t. This corresponds to a cyclic splitting pattern for the kd trie. We can generalize to patterns other than cyclic. Another generalization is that the domains need not be the same size. Both of these generalizations can be accomplished by setting the values in the attr array. For example, in a 4 x 8 space with attr[0]=0, attr[1]=1, attr[2]=1, attr[3]=0 and attr[4]=1, shuffle($[00_2, 111_2]$) $= 01101_2$ and shuffle($[10_2, 001_2]$) $= 10001_2$ so $[00_2, 111_2] \leq_z [10_2, 001_2]$. By modifying the shuffle function in this way, it is possible to obtain "high bias" MDSs, (see chapter 2 section 1.2).

Clearly, unshuffling is also possible. That is, given shuffle(t) (and the attr values) the attribute values of t can be determined. The unshuffle function (given below) is applied to tuples retrieved in a search before they are reported.

shuffle(tuple) Shuffle is an array of $h = w_0 + \ldots + w_{k-1}$ bits where $w_i = |dom(A_i)|$, $i = 0, \ldots, k-1$. Tuple is an array of k integers, one for each attribute. The ith attribute value is represented by a string of w_i bits labelled (from MSB to LSB) 0, 1, ... , $w_i - 1$. Attr[i] indicates the attribute of the tuple from which the ith bit of the shuffle value should be selected. Count is an array of k counters: count[a] is the next bit position of the ath attribute to be included in the shuffled tuple.

```
for a := 0 to k-1
   count[a] := 0
end

for i := 0 to h-1
   a := attr[i]
   shuffle[i] := (bit count[a] of tuple[a])
   count[a] := count[a] + 1
end
return

end shuffle
```

unshuffle(stuple)
Stuple is an array of h bits representing the shuffled tuple. Unshuffle is an array of k integers (of $w_0, \ldots, w_{k-1}$ bits respectively), one for each attribute, which will store the unshuffled tuple.

```
for a := 0 to k-1
   count[a] := 0
end

for i := 0 to h-1
   a := attr[i]
   (bit count[a] of unshuffle[a]) := stuple[i]
   count[a] := count[a] + 1
end
return

end unshuffle
```

2.2. Evaluating range queries

In this section we describe an algorithm for the evaluation of range queries. This algorithm can be applied to any indexed-sequential data structure that stores tuples in z order. (The idea of storing shuffled tuples in unidimensional data structures is attributed to McCreight by Bentley in [Bent75a].)

In each step of the search algorithm a search region (SR) is generated and tested for overlap with the query region (QR). Three cases can occur:

1) The SR is outside the QR: The SR does not contain any points whose tuples satisfy the query.

2) The SR is inside the QR: All of the points inside the SR correspond to tuples satisfying the query. The tuples are retrieved, unshuffled and reported.

3) The SR overlaps (but is not inside) the QR: The SR is split into two smaller SRs which are searched recursively.

In case (3) the various SRs are constructed. The method of splitting is exactly the same method used to split regions in the kd trie. Therefore, the tuples retrieved in case (2) are consecutive in z order. (Recall that the tuples in the external nodes descended from any kd trie node are consecutive in the z order.)

The retrieval of tuples in case (2) is simple given an ISDS: Suppose that the SR is $[l_0:u_0, \ldots, l_{k-1}:u_{k-1}]$. Then the tuples to be retrieved have shuffle values in the range [shuffle($[l_0, \ldots, l_{k-1}]$), shuffle($[u_0, \ldots, u_{k-1}]$)]. The first tuple of the z order whose shuffle value is in this range can be located by a random access to the ISDS storing the shuffled tuples. The others can be retrieved by sequential accesses.

The search algorithm is given below in greater detail. The algorithm is invoked by calling Rangesearch(QR, $[0:D_0-1, \ldots, 0:D_{k-1}-1]$, 0).

```
Rangesearch(QR, SR, level)
Level is the depth of recursion. A tuple is represented by h
bits so level ≤ h. If level = h then the size of the
corresponding SR is one bit and either SR ⊆ QR or SR ∩ QR = ∅.

if SR ∩ QR = ∅    (* case 1 *)
then    (* do nothing *)
   pass
else if SR ⊆ QR    (* case 2 *)
then    (* retrieve all tuples in the SR *)
   t := randac(loval(SR))
   while tuple(t) ≤ hival(SR)
      report unshuffle(tuple(t))
      t := seqac(t)
   end
else    (* SR ∩ QR ≠ ∅,    case 3 *)
   Rangesearch(QR, left(SR, attr[level]), level+1)
   Rangesearch(QR, right(SR, attr[level]), level+1)
end if
return

end Rangesearch


left(SR, a)
(* The range of attribute a of SR is l_a : u_a *)
u_a := (l_a + u_a - 1) / 2
return(SR)

end left


right(SR, a)
(* The range of attribute a of SR is l_a : u_a *)
l_a := (l_a + u_a + 1) / 2
return(SR)

end right
```

If SR = $[l_0:u_0, \ldots, l_{k-1}:u_{k-1}]$ then loval(SR) = shuffle($[l_0, \ldots, l_{k-1}]$) and hival(SR) = shuffle($[u_0, \ldots, u_{k-1}]$). Tuple(t) returns the shuffled tuple located at address t. Randac(x) locates the first tuple of the relation in the z order whose shuffled value is at least x and returns the address of that tuple. Seqac(t) finds the successor of the tuple located at

address t. If tuple(t) has no successor then seqac(t) returns an address such that tuple(t) = ∞. Left(SR,a) splits the SR on attribute a and returns the left sub-region. Right(SR,a) returns the corresponding right sub-region. For example, if the ath attribute of SR is $1011000_2:1011111_2$ then the ath attribute of left(SR, a) is $1011000_2:1011011_2$ and the ath attribute of right(SR, a) is $1011100_2:1011111_2$.

2.3. Refinements to Rangesearch

The Rangesearch algorithm of section 2.2 has some problems. In this section we discuss these problems and some solutions.

The most serious problem is that Rangesearch may generate a vast number of SRs, each of which generates a random access. Most of these SRs will not generate page faults given a typical page management system, but the amount of CPU work involved will be overwhelming. Figure 10 illustrates the problem. Near the query boundaries, many very small SRs, (one and two bit regions in figure 10), may be generated. The number of these SRs is determined by the location and dimensions of the query and is independent of the data stored. (In section 2.7 we derive an expression for the number of random accesses generated by a certain 2d query. In the worst case, a query of size $X_0 \times X_1$ can generate $O(X_0 + X_1)$ random accesses.) The problem arises because many splits may be required before SRs not overlapping the QR are generated.

Another problem with Rangesearch is that the stack of SRs generated by the recursive calls may become quite large. The maximum depth of recursion, h, is also the number of bits

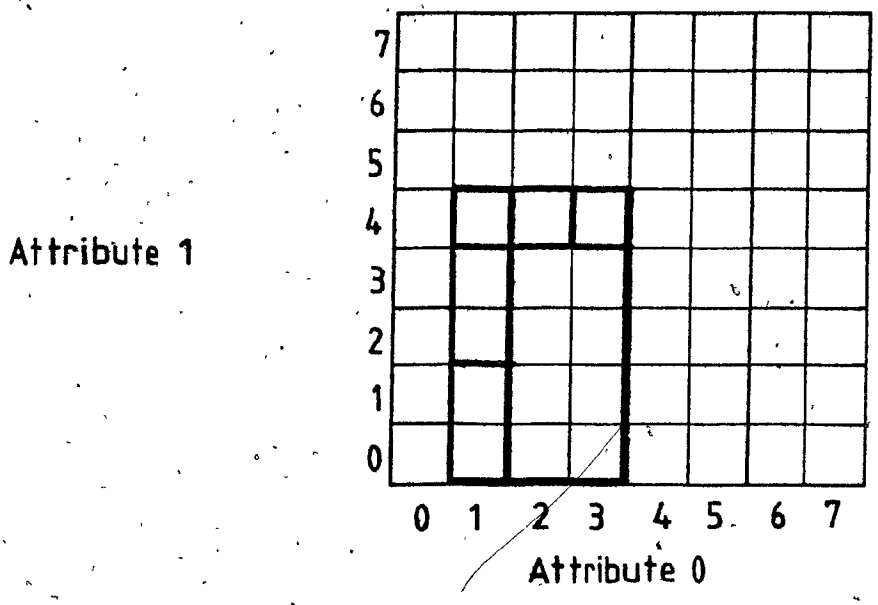


Figure 10. A query region and its component search regions.

required to represent a tuple. Each SR occupies 2h bits for a total of $2h^2$ bits. Since, in practice, h may be in the thousands, the size of the stack can be a problem.

2.3.1. Eliminating the stack

The stack of SRs can be eliminated. By storing just the top SR, previous SRs can be derived. Thus the stack can be represented in 2h bits.

Let SR(i) denote the SR on the ith level call of Rangesearch. From SR(i), two SRs can be derived: The left child of SR(i) is LSR(i) and the right child of SR(i) is RSR(i). Given SR(i), either SR(i+1) = LSR(i) or SR(i+1) = RSR(i).

The stack of SRs can be eliminated since SR(i) can be reconstructed from SR(i+1). If SR(i) = $[l_0:u_0, \ldots, l_a:u_a, \ldots, l_{k-1}:u_{k-1}]$ where a = attr[i] then LSR(i) = $[l_0:u_0, \ldots, l_a:u'_a, \ldots,$ and RSR(i) = $[l_0:u_0, \ldots, l'_a:u_a, \ldots, l_{k-1}:u_{k-1}]$. SR(i) can be reconstructed from LSR(i) since $u_a = 2u'_a - l_a + 1$, (recall that $u'_a = (l_a+u_a-1)/2$). Similarly, SR(i) can be reconstructed from RSR(i): $l_a = 2l'_a - u_a - 1$. Now, the stack of SRs has been reduced to a stack of bits. The ith bit indicates whether SR(i+1) = LSR(i) or SR(i+1) = RSR(i).

Even this small stack can be eliminated. Using the notation given in the introduction of this chapter, the binary representation of $l_a:u_a$ (or $l'_a:u'_a$) is

$$\langle *:d-c-1 \mid Y \mid 0:c\rangle \; : \; \langle *:d-c-1 \mid Y \mid 1:c\rangle$$

where $c \geq 0$, $d = w_a$ is the number of bits required to represent values from the ath domain, each * is either 0 or 1 and Y is 0 or 1. The corresponding range in the parent region is $l_a:u_a$ =

$$\langle *:d-c-1 \mid 0 \mid 0:c\rangle \; : \; \langle *:d-c-1 \mid 1 \mid 1:c\rangle$$

If Y = 0 then the child was a left child; if Y = 1 then the child was a right child.

This view of ranges provides another way to calculate LSR(i) and RSR(i). If l_a = <*:d-c-1 | 0 | 0:c> and u_a = <*:d-c-1 | 1 | 1:c> then u'_a = <*:d-c-1 | 0 | 1:c> and l'_a = <*:d-c-1 | 1 | 0:c>. That is, the calculation can be achieved by complementing a single bit.

To summarize, the stack is unnecessary since the SR(i) from which SR(i+1) was derived can be reconstructed. The technique presented here will be of use in solving the more serious problem of reducing the number of random accesses.

2.3.2. Reducing the number of random accesses

2.3.2.1. Non-homogeneous ISDSs

A non-homogeneous tree-based data structure stores all the records in the leaves. Internal nodes store only discriminators which guide the search, (e.g. a B+tree). The leaves can be linked together into a list. Sequential accesses are then very cheap, requiring at most one page access and usually none.

A leaf page of a non-homogeneous ISDS contains tuples whose shuffle values are $s_1, \ldots, s_p$ where p is the number of tuples on the page and $s_1 < \ldots < s_p$. Many small SRs are likely to fall between successive tuples: Along a query boundary of size x there can be up to O(x) small SRs, (see section 2.7). Typically, the space representing a relation is so sparse that the vast majority of these SRs contain no tuples satisfying the query.

To avoid dealing with these SRs, the Rangesearch algorithm

could test all the tuples on a retrieved page for inclusion in the QR. The problem with this approach is that the state of the search algorithm, as described by SR and level, is out of date after the page has been processed. Somehow, these variables must be reset so that the search can resume.

The last value on the page is s_p. This tuple, (or any other), can be regarded as a one bit region. If $s_p = [a_0, \ldots, a_{k-1}]$ then the corresponding (one bit) region is $[a_0:a_0, \ldots, a_{k-1}:a_{k-1}]$. To begin reconstruction of the state of the search algorithm we set SR = $[a_0:a_0, \ldots, a_{k-1}:a_{k-1}]$ and level = h. (In general, an m bit region is explored at level h-log(m). The current SR contains 1 bit.) Using the process described in section 2.3.1, ancestral SRs can be reconstructed. This is analogous to "climbing up" a trie. Attr indicates which attribute's range to modify at each step of the reconstruction. Each time a parent SR is reconstructed, level is decremented by one. This process continues until the SR overlaps the QR and the child of SR was a left child. (If there is no overlap then we are in a part of the trie that should not be explored. If there is overlap and the child was a right child then both children of the SR have been explored.) With SR and level reset, the search resumes with RSR(level). The modified Rangesearch algorithm is given below.

Figure 11 illustrates the search using the data of figure 8 and the QR of figure 10. The tuples have been stored on pages 1-4 in z order. The tuples have been placed at positions in the diagram representing their shuffle values. The query is broken up into SRs (displayed beneath the tuples). In the original version of Rangesearch, each SR would be searched in turn yielding tuples

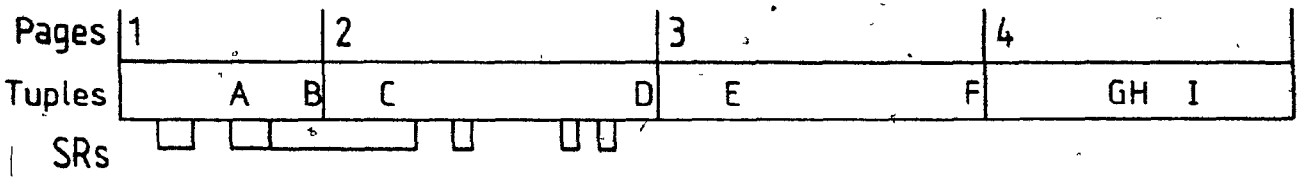


Figure 11. The relationship between tuples, pages and search regions.

A, B and C. The modified version of Rangesearch would proceed as follows:

- The leftmost SR is generated, causing page 1 to be read. All the tuples on the page are checked for inclusion in the QR. Both A and B are included in the QR although neither belongs to the SR which generated the page read.
- Reconstruction of the state of the search yields an SR which causes page 2 to be read. Tuples C and D are checked against the QR. C is inside the QR but D is not. Note that the three small SRs between C and D were not generated.
- Reconstruction from tuple D causes the search to terminate since D is past the last SR. Pages 3 and 4 are not accessed.

Rangesearch(QR)

Rangesearch is now specified iteratively since the reconstruct algorithm "jumps across" levels of recursion. Dir indicates whether we are climbing up (towards the root) or down the trie. Childside is left or right depending on whether the child of the current SR was LSR or RSR.

```
SR := [0:D0-1, ... , 0:Dk-1 -1]
level := 0
dir := down
repeat until level = 0 and childside = right
   switch dir
   case down:
      if SR ⊆ QR
         process(SR, t)    (* get all the tuples in the SR and *)
                           (* scan the remainder of the last page read *)
         reconstruct(t, SR, level, childside)
         dir := up
      else if SR ∩ QR ≠ ∅   (* explore the left SR first *)
         SR := left(SR, attr[level])
         level := level + 1
            (* return the new SR and increment level *)
      else   (* SR ∩ QR = ∅ *)
         dir := up
         (* restore is done below: case up and SR ∩ QR = ∅ *)
      end if
   case up:
      if SR ∩ QR ≠ ∅   (* Have just reconstructed or restored *)
         dir := down
         SR := right(SR, attr[level])
         level := level + 1
            (* return the new SR and increment level *)
      else   (* SR ∩ QR = ∅ *)
         restore(SR, level, childside)
      end if
   end switch
end
return

end Rangesearch
```

process(SR, t)
Retrieve all tuples in the SR, starting on the page accessed by loval(SR). (This may require reading more pages.) Then, test all tuples on the last page read for inclusion in QR. L(t) is the address of the last tuple on the page containing t.

```
t := randac(loval(SR))
while tuple(t) ≤ hival(SR)
   report tuple(t)
   t := seqac(t)    (* may cause a new page to be read *)
end

(* Finish up the last page read *)
end-of-page := L(t)
while t ≤ end-of-page
   if tuple(t) ∈ QR then report tuple(t)
   if t < end-of-page then t := seqac(t)
end
return

end process
```

restore(SR, level, childside)

```
level := level - 1
a := attr[level]

(* The range of attribute a of SR, l :u , expressed in binary is *)
(* <*:d-c-1 | Y | 0:c>  :  <*:d-c-1 | Y | 1:c>. *)

if Y = 0
then
   childside := left
   u_a := 2 u_a - l_a + 1
else
   childside := right
   l_a := 2 l_a - u_a - 1
return

end restore
```

reconstruct(t, SR, level, childside)

```
[a_0, ... , a_k-1] := tuple(L(t))

(* reconstruct from last tuple of page *)
SR := [a_0:a_0, ... , a_k-1 :a_k-1]
level := h
repeat until SR ∩ QR ≠ ∅ and childside = left
   restore(SR, level, childside)
end
return

end reconstruct
```

2.3.2.2. Homogeneous ISDSs

A homogeneous ISDS stores records in both the internal and leaf nodes of a tree-based data structure, (e.g. a Btree). For such data structures, the technique of section 2.3.2.1 is not always correct: the tuples on one page $s_1, \ldots, s_p$ do not represent all the tuples t, such that $s_1 \leq \text{shuffle}(t) \leq s_p$ unless the page is a leaf. That is, an SR between s_1 and s_p may actually cover tuples on a descendent page. Unless hival(SR) is the shuffle value of a tuple of the relation appearing on a non-leaf page the processing of the SR terminates on a leaf page. It will, therefore, be an extremely rare event that the search terminates on a non-leaf: the probability that hival(SR) corresonds to a tuple is very small since, in practice, the space is so sparse. Furthermore, in practice, the ISDS is a balanced tree of high degree so most of the pages are leaves.

For a homogeneous ISDS then, the version of the process algorithm given below would be used. If the search terminates on a leaf page, the rest of the page is scanned as in section 2.3.2.1. If the search terminates on a page which is not a leaf, (this will rarely occur), then the leaf page containing the successor of the last tuple seen is scanned.

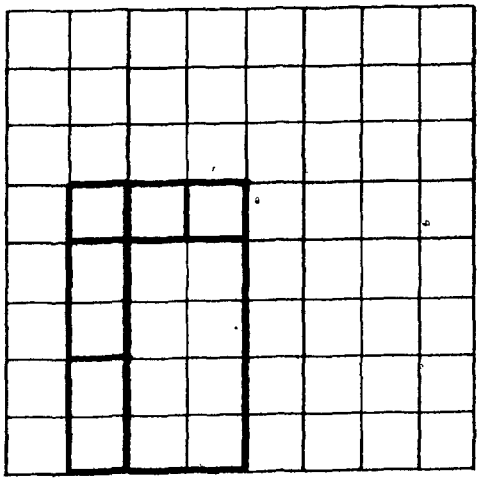
```
process(t, SR)
Page(t) is the page containing tuple t.

t := randac(loval(SR))
while tuple(t) ≤ hival(SR)
   report tuple(t)
   t := seqac(t)    (* may cause a new page to be read *)
end

if page(t) is not a leaf
then t := seqac(t)    (* the successor is on a leaf page *)

end-of-page := L(t)
while t≤ end-of-page
   if tuple(t) ∈ QR then report tuple(t)
   if t < end-of-page then t := seqac(t)
end
return

end process
```

2.3.2.3. Query expansion

Query expansion is another method for reducing the number of random accesses generated by a QR. It is applicable regardless of the ISDS used. The basic idea is to embed the QR in a larger region, QR', which generates fewer random accesses, (but possibly more sequential accesses). (See figure 12.) The Rangesearch algorithm is modified to process QR' and filter out tuples in QR' - QR.

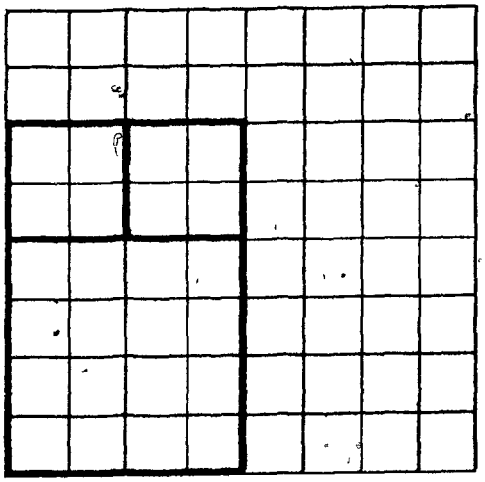
Our discussion of query expansion follows from the analysis of a certain QR in section 2.7. The QR considered is $[0{:}X_0-1, 0{:}X_1-1]$. The binary representations of X_0 and X_1 determine R(QR), the number of random accesses generated by the QR.

For the special case $X = X_0 = X_1$, we show in section 2.7 that $R(QR) = O(2^q)$ where q is the distance, (number of bit positions), between the leftmost and rightmost 1 in the binary representation of X. (In section 2.7 we also discuss the case $X_0 \neq X_1$.) This result says, in a sense, that R(QR) is independent of the size of the query: $R([0{:}X-1, 0{:}X-1]) = R([0{:}2X-1, 0{:}2X-1])$. This is so



Original query. [001 011, 000 100]

6 SRs



Expanded query: [000.011, 000·101]

3 SRs

Figure 12. Query expansion.

because doubling X is equivalent to a "left shift" which does not affect q.

In order to reduce R(QR), QR can be replaced by QR′ such that QR C QR′ and q(QR) > q(QR′). For example, if QR = [0:X-1, 0:X-1] where X = 00110101_2 then QR′ = [0:X′-1, 0:X′-1] where X′ = 00111000_2 satisfies our requirements.

There is a relationship between the bits of X and the partitions creating the SRs. Increasing X to X′ corresponds to moving the boundaries of the query outward so that larger (and fewer) SRs will be needed to process the query.

A modified version of Rangesearch which employs query expansion is given below.

There is a cost to be paid for using QR′ instead of QR: a larger portion of the space is covered and it is likely that more tuples will be retrieved. Simple calculations based on the formula for R(QR) suggest that it is well worth expanding the query by a large amount, (assuming a uniform distribution of tuples). For example, if X = <0:r | 10 | *:d-r-2> is expanded to X′ = <0:r | 11 | 0:d-r-2> then the portion of the space covered increases as much as 125% but the number of random accesses is exactly 5.

```
RangesearchQE(QR)
Find QR′    (* with properties given in text *)
Rangesearch(QR′)    (* tuples are placed in report file *)
For each t ∈ report
   if t ∉ QR then delete t from report
end
return

end RangesearchQE
```

2.4. Multidimensional data structures based on z order, (ZMDSs)

The Rangesearch algorithm can be used in conjunction with any ISDS. A large number of such data structures are known. The randac and seqac procedures (called by Rangesearch) must be supplied for each such data structure.

Multidimensional data structures based on z ordering (ZMDSs) can be generated by applying the shuffle, unshuffle and Rangesearch algorithms to one of these ISDSs, (see figure 13). We now discuss a number of ZMDSs. Many of these are new data structures for range searching.

2.4.1. Zkd binary search

The simplest ISDS is the array of ordered data. By storing shuffled tuples, the array can be used for the evaluation of range queries. Randac(x) is a binary search for the smallest entry greater than or equal to x. Seqac(t) increments t (a subscript pointing to an array location) by 1.

Of course, this data structure can only be used efficiently for static files.

2.4.2. Zkd tree

By using the binary tree as the ISDS, a more dynamic MDS is obtained. It is not the same as Bentley's kd tree [Bent75a]: the inorder traversal of the kd tree does not necessarily yield the tuples in z order. (There are also some zkd trees which do not correspond to any kd tree.) As discussed in section 2.5, maintenance of any order is a useful property.

All the modifications of binary trees apply to zkd trees. In particular, if the AVL tree [Knut73] is used instead of the

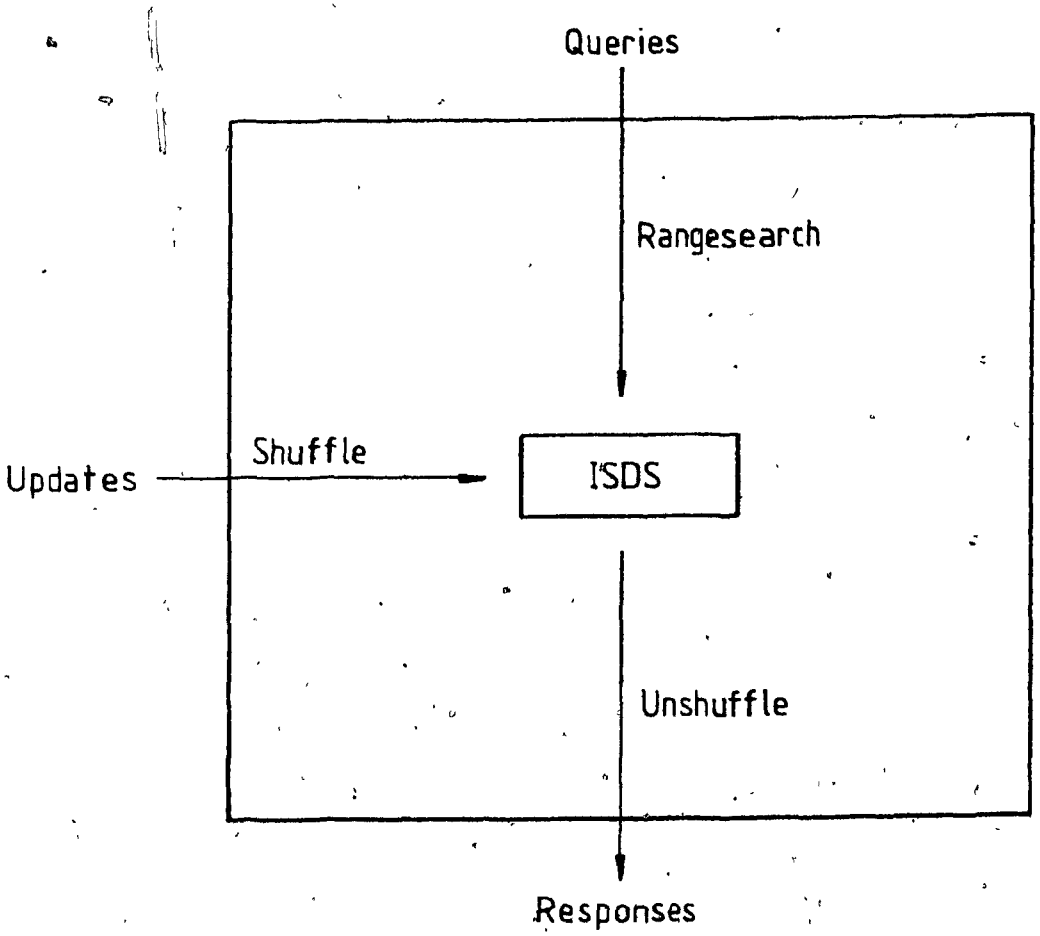


Figure 13. Converting an ISDS into a ZMDS.

binary tree then the result is a zkd tree which does not degenerate. Except for rebuilding, no method is known for maintaining the balance of kd trees.

2.4.3. Zkd Btree

The classical ISDS for secondary storage is the Btree [Baye72]. The derived ZMDS is the zkd Btree. As with binary trees, any variant of the Btree, (see [Come79] for a survey), can be used in place of the standard Btree.

The zkd Btree is much simpler that Robinson's K-D-B tree [Robi81] and Scheuerman and Ouksel's MDB tree [Sche82]. Furthermore, it inherits from the Btree an expected load factor of about 70% [Yao78], and a worst case load factor of 50% [Baye72]. The K-D-B tree does not have either of these properties.

2.4.4. Kd trie

The kd trie is based on Fredkin's trie [Fred60] which is an ISDS.

2.4.5. EXCELL

Tamminen's extendible cell method (EXCELL) [Tamm80] is closely related to the kd trie but uses a different ISDS. The ISDS on which EXCELL is based is extendible hashing of Fagin et al. [Fagi79]. 1d EXCELL is obtained by using extendible hashing with the hash function h(k) = k. Multidimensional EXCELL is the 1d version augmented by shuffle, unshuffle and Rangesearch. (Note: Tamminen did not give an algorithm for evaluating range queries on EXCELL.) A related data structure, HCELL [Tamm81], (see

chapter 2 section 1.2), is also a ZMDS.

2.4.6. Multiple attribute trees

The MDB tree of Scheuermann and Ouksel [Sche82] is descended from the doubly-chained tree [Suss63], the multiple attribute tree [Kash77] and the modified multiple attribute tree [Gopa80]. These can all be seen as ZMDSs which use a "trivial" shuffle function: using the notation of section 2.1, if $X = [x_{00}x_{01}\ldots x_{0,d-1}, \ldots, x_{k-1,0}x_{k-1,1}\ldots x_{k-1,d-1}]$ then the trivial shuffle function is

$$\text{TrivialShuffle}(X) = x_{00}x_{01}\ldots x_{0,d-1}\ldots x_{k-1,0}x_{k-1,1}\ldots x_{k-1,d-1}$$

The four data structures use different ISDSs but they all transform the data using TrivialShuffle. Customized search algorithms, different from Rangesearch, were proposed for these data structures.

2.5. Other uses of z ordering

Data in files are often ordered to allow efficient implementation of algorithms requiring merging. Since all ZMDSs store tuples in z order, the efficient merging of files of multidimensional data stored in ZMDSs is possible, (even if the operands are based on different ISDSs).

In particular, linear time implementation of the set operations is possible. This is not possible with multidimensional data structures that do not preserve order, (e.g. the kd tree and multidimensional clustering).

2.6. Performance

Many of the performance characteristics of a ZMDS are inherited from the underlying ISDS. These properties include storage utilization and dynamic behaviour (e.g. time required for insertion).

Some general statements can be made about the performance of ZMDSs. In section 2.7 we analyze the expected cost of partial match and range queries for non-homogeneous ZMDSs given a certain distribution of tuples. We believe that the results are also valid for certain homogeneous ZMDSs. (The zkd Btree for example. Since most pages are leaves, it is "almost" non-homogeneous.) The results are the same as for the kd tree, the kd trie (see section 1.5.4) and EXCELL: for a relation stored on P pages, a partial match query costs $O(P^{1-t/k})$ page reads where $t < k$ is the number of attributes specified in the query; a range query costs $O(VP)$ page reads where V is the "volume" of the query, (the fraction of the space covered by the query).

2.7. Analysis of the ZMDSs

2.7.1. Number of search regions in a query region

The problem considered here is to determine R(QR), the number of SRs generated in the exploration of a given QR.

We consider square 2d spaces and assume that attr[i] = i mod 2. Each domain contains 2^d elements. We consider a special case, QR = [0:x-1, 0:y-1] where x and y are integers between 1 and 2^d inclusive. Let $u_0 \ldots u_{d-1}$ and $v_0 \ldots v_{d-1}$ be the binary representations of x and y respectively.

Let A(x,y) = R(QR).

Suppose the first split of the space is vertical and it partitions the domain of x into the sets {00...00, 00...01, ... , 01...11} and {10...00, 10...01, ... , 11...11}. If $x < 2^{d-1} = r_0$ then this split is outside the QR (see figure 14a) otherwise it passes through the QR (see figure 14b). Clearly, $x < 2^{d-1}$ iff $u_0 = 0$.

Now, after the first split we have

$$A(x,y) = \begin{cases} A(r_0,y) + A(x-r_0,y), & u_0=1 \\ A(x,y), & u_0=0 \end{cases}$$

$$= u_0 A(r_0,y) + A(x-u_0 r_0,y) \qquad (1)$$

To clarify the analysis we will rename the first term "$B(r_0,y)$" so that it corresponds to region B of figure 14b. The analysis of the B term will be simpler than that of the A term: when we do further splits it will generate other B terms, not both A and B terms. This is because the first argument of B is always a power of 2 whereas the first argument of A may not be. Splitting B vertically corresponds to division of the first argument by 2, yielding another power of 2, (hence, a B term, not an A term).

The A region is next split at position r_0 in the horizontal direction. Proceeding as above, the split may or may not pass through the region. Two cases are possible (see figure 15) and we have (after renaming one of the A regions a C region as above)

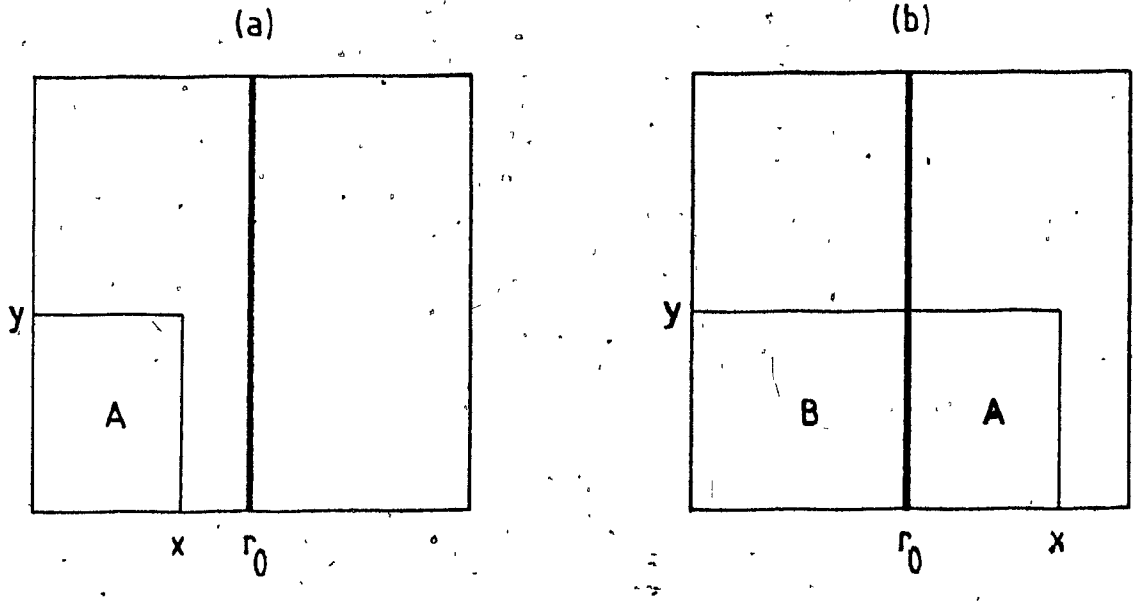


Figure 14. Vertical split of the space.

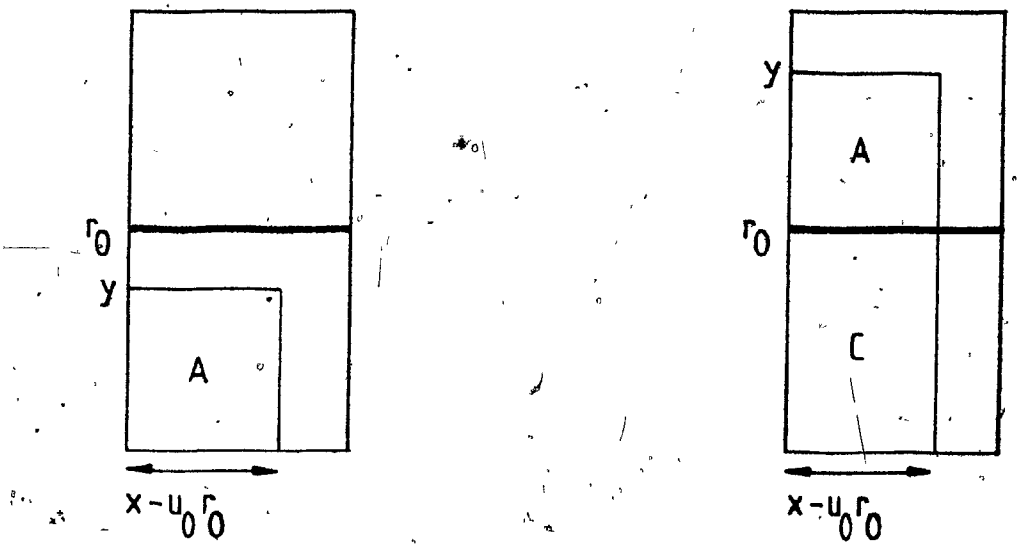


Figure 15. Horizontal split of the space in the vicinity of an A region.

$$A(x-u_0r_0,y) = v_0C(x-u_0r_0,r_0) + A(x-u_0r_0,y-v_0r_0) \qquad (2)$$

Combining (1) and (2):

$$A(x,y) = u_0B(r_0,y) + v_0C(x-u_0r_0,r_0) + A(x-u_0r_0,y-v_0r_0) \qquad (3)$$

If $u_0 = 1$ and $v_0 = 1$ then these two splits have created three regions (see figure 16).

The A region can be analyzed exactly as was the original QR. For example, after two more splits:

$$A(x,y) = u_0B(r_0,y) + v_0C(x-u_0r_0,r_0) + u_1B(r_1,y-v_0r_0) + v_1C(x-u_0r_0-u_1r_1,r_1) + A(x-u_0r_0-u_1r_1,y-v_0r_0-v_1r_1) \qquad (4)$$

(r_1 is defined below.) Clearly, $A(0,0) = 0$. We can now give an expression for $A(x,y)$ in terms of the Bs and Cs.

$$A(x,y) = \sum_{i=0}^{d-1} u_iB(r_i, y - \sum_{j=0}^{i-1} v_jr_j) + \sum_{i=0}^{d-1} v_iC(x - \sum_{j=0}^{i} u_jr_j, r_i) \qquad (5)$$

We now derive expressions for the B and C terms.

The B region was created by a vertical split. The next split is horizontal. As above, two cases can occur. Consider $B(r_0,y)$, (see figure 17). In the first case one SR and another B term is

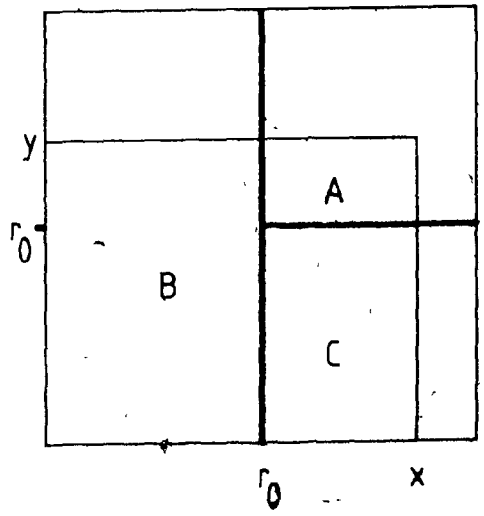


Figure 16. After the first two splits.

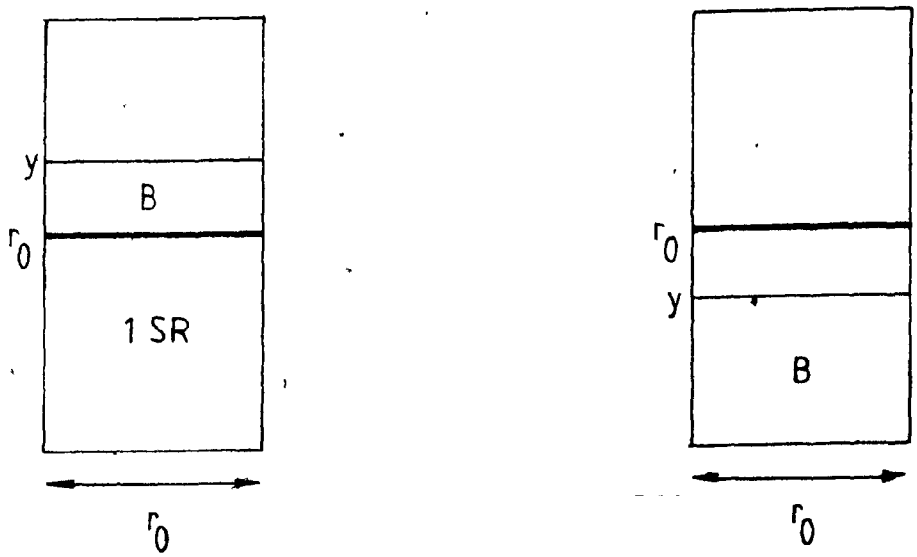


Figure 17. Horizontal split of the space in the vicinity of a B region.

generated. The split is inside the region iff $v_0 = 1$.

$$B(r_0,y) = \begin{cases} 1 + B(r_0,y-r_0), & v_0=1 \\ B(r_0,y), & v_0=0 \end{cases}$$
$$= v_0 + B(r_0,y-v_0r_0) \qquad (6)$$

The next split is perpendicular to the query boundary and is certain to split the B region into two B regions. So after a horizontal and a vertical split of a B region two cases are possible, (see figure 18). (Note that the SR is not split further.) In either case

$$B(r_0,y) = v_0 + 2\ B(r_1,y-v_0r_0) \qquad (7)$$

where $r_1 = r_0/2$. Clearly, $B(r_1,0) = 0$ where $r_i = 2^{d-1-i}$. The second argument to B is never negative. The v_i coefficient insures this since v_i is zero whenever subtraction of r_i would give a negative result. So we have

$$B(r_0,y) = \sum_{k=0}^{d-1} v_k 2^k \qquad (8)$$

(The value of $B(r_0,y)$ is the integer created by writing the binary representation of y backwards!)

Similar analysis for the other B terms yields

$$B(r_i,y-\sum_{j=0}^{i-1} v_j r_j) = \sum_{k=i}^{d-1} 2^{k-i} v_k \qquad (9)$$

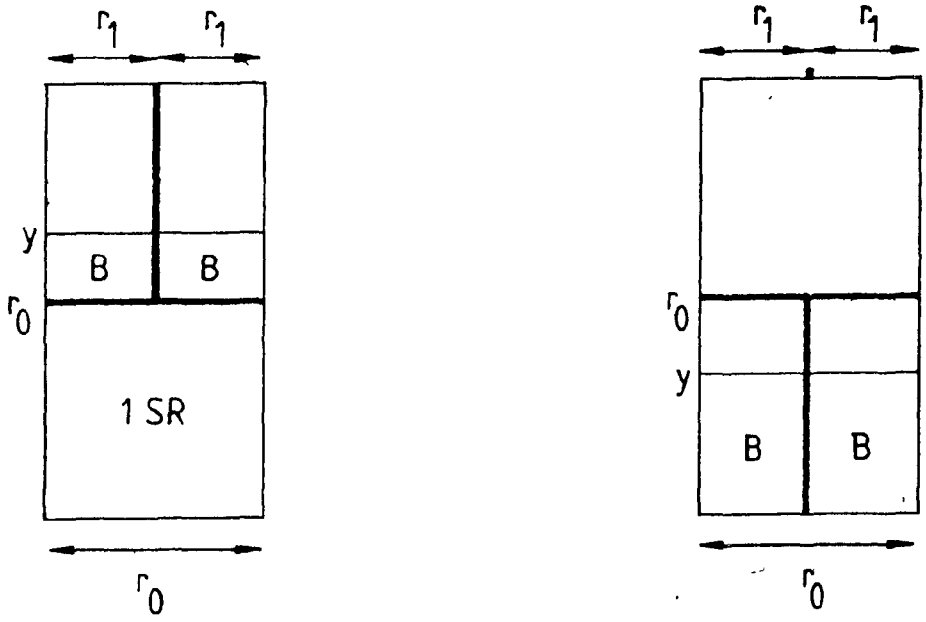


Figure 18. Vertical split of the space in the vicinity of of a B region.

The analysis of the C terms is similar to that of the B terms. The result is

$$C(x-\sum_{j=0}^{i} u_j r_j, r_i) = \sum_{k=i+1}^{d-1} 2^{k-1-i} u_k \qquad (10)$$

Combining (5), (9) and (10)

$$A(x,y) = \sum_{i=0}^{d-1} [2^{-i} u_i \sum_{k=i}^{d-1} 2^k v_k + 2^{-(i+1)} v_i \sum_{k=i+1}^{d-1} 2^k u_k] \qquad (11)$$

To understand (11) better we will discuss those factors most heavily influencing A(x,y). Consider the special case x = y. Then the following two results are derivable from (11):

- If x has 1s in bit positions p and p+q and 0s elsewhere then $A(x,x) = 3(2^{q-1})+2$. (12)

- If x has 1s from position p to position p+q inclusive and 0s elsewhere then $A(x,x) = 3(2^{q+1}) - 2q - 5$. (13)

Furthermore, for all x, y, changing any 0 to a 1 in x or y increases A(x,y). (14)

These three facts establish $R(QR) = O(2^q)$ for square queries where q is the distance (i.e. number of bit positions) between the first and last 1 in x: (12) is a lower bound. Applying (14)

to any 0 in positions p+1 through p+q-1 inclusive increases A(x,x) and (13) is an upper bound. Both (12) and (13) are $O(2^q)$. It is interesting to note that A(x,x) in (12) and (13) is independent of p. This implies that scaling by factors of 2 has no effect of R(QR) (for square queries). These results can be generalized to non-square queries (in square spaces). The proof proceeds as in (12) - (14) and is simple and tedious. The result is that R(QR) is dominated by an $O(2^r)$ term where r is the longest distance (i.e. number of bit positions) between a 1 bit in x and a 1 bit in y. (E.g. if $x = 01101000_2$ and $y = 00000110_2$ then r = 6.)

2.7.2. ZMDS Performance

We present evidence that the expected performance of any ZMDS on secondary storage is the same as for the kd trie:

- A partial match query on t attributes costs $O(P^{1-t/k})$ where P is the number of pages storing the data.
- A range query costs O(VP) where V is the "volume" of the query: the fraction of the space covered by the query.

These results depend on some properties of the z curve.

2.7.2.1. Z curve properties

We will analyze z curves in k dimensions where each attribute has 2^d possible values: 0, ... , 2^d-1.

For now, consider a point P(x,y) in a 2d space such that x, y $< 2^d$. Let s(P) or s(x,y) denote the shuffle value of (x,y). We observe that s(x+1,y)-s(x,y) is independent of y. Similarly, s(x,y+1)-s(x,y) is independent of x, (see figure 19). The

								s(x, y+1) − s(x, y)
21	23	29	31	53	55	61	63	1
20	22	28	30	52	54	60	62	3
17	19	25	27	49	51	57	59	1
16	18	24	26	48	50	56	58	11
5	7	13	15	37	39	45	47	1
4	6	12	14	36	38	44	46	3
1	3	9	11	33	35	41	43	1
0	2	8	10	32	34	40	42	
s(x+1, y) − s(x, y)	2	6	2	22	2	6	2	

Figure 19. Gap sizes.

quantities s(x+1,y)-s(x,y) and s(x,y+1)-s(x,y) are "gaps".

Let us label X as attribute 0 and Y as attribute 1. Also, let T(u) be the length of the longest suffix of the binary representation of u consisting of 1s only, (e.g. $T(010111_2) = 3$). Then G(i,T(u)) is the gap along attribute i, i=0, 1, between attribute i values u and u+1.

In the example given above:

	i		
	G	0	1
	0	2	1
T(u)	1	6	3
	2	22	11

We also notice that G(0,r) = 2 G(1,r).

The observations made here will be generalized and proven. A formula for G(i,r) will be given. First, we sketch the derivation for k = 2. We then give a more rigorous treatment for $k \geq 2$.

2.7.2.1.1. Proof for k = 2

We want to establish that

$$s(x+1,y) - s(x,y) = G(0,T(x)) \tag{1}$$

and

$$s(x,y+1) - s(x,y) = G(1,T(y)) \tag{2}$$

and derive a formula for G(i,r), i = 0, 1. The proof is inductive.

First consider a 2 x 2 space, (see figure 20).

By inspection, $G(0,0) = 2$ and $G(1,0) = 1$. (1) and (2) have been established for the trivial case.

Next, consider a region of size $2^{r+1} \times 2^{r+1}$, (see figure 21). A, B, C and D are the four bits at the center of this region. Their coordinates are

A: $(2^r-1, 2^r)$

B: $(2^r-1, 2^r-1)$

C: $(2^r, 2^r)$

D: $(2^r, 2^r-1)$

Our induction hypothesis (IH) is that (1) and (2) hold in Bs region for $G(i,s)$, $s < r$, $i = 0, 1$.

It can be shown that the IH also holds in the other three quadrants, (we will prove this for $k \geq 2$ later).

We will extend the IH to $G(i,r)$ in the following way:

- Compute $g_L = s(A) - s(B)$ and $g_R = s(C) - s(D)$.
- Show that $g_L = g_R$ $(= g)$.
- Show that $s(x,2^r) - s(x,2^r-1) = g$ for $0 \leq x \leq 2^{r+1}-1$, (i.e. all along the $G(1,r)$ boundary).
- These results establish (2) and show that $G(1,r) = g$.
- Repeat these steps (using C, A, D, B in place of A, B, C, D respectively) to establish (1) and derive $G(0,r)$.

The coordinates of A, $(2^r-1, 2^r)$, expressed in binary are

$$(\langle 0{:}d-r-1 \mid 0 \mid 1{:}r\rangle, \langle 0{:}d-r-1 \mid 1 \mid 0{:}r\rangle)$$

and

$$s(A) = \langle 00{:}d-r-1 \mid 01 \mid 10{:}r\rangle$$

or, in base 4 notation:

$$s(A) = \langle 0{:}d-r-1 \mid 1 \mid 2{:}r\rangle$$

Similarly, in base 4 notation,

$$s(B) = \langle 0{:}d-r-1 \mid 0 \mid 3{:}r\rangle$$

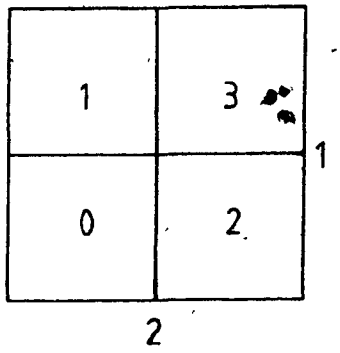


Figure 20. Gap sizes for a 2 x 2 region.

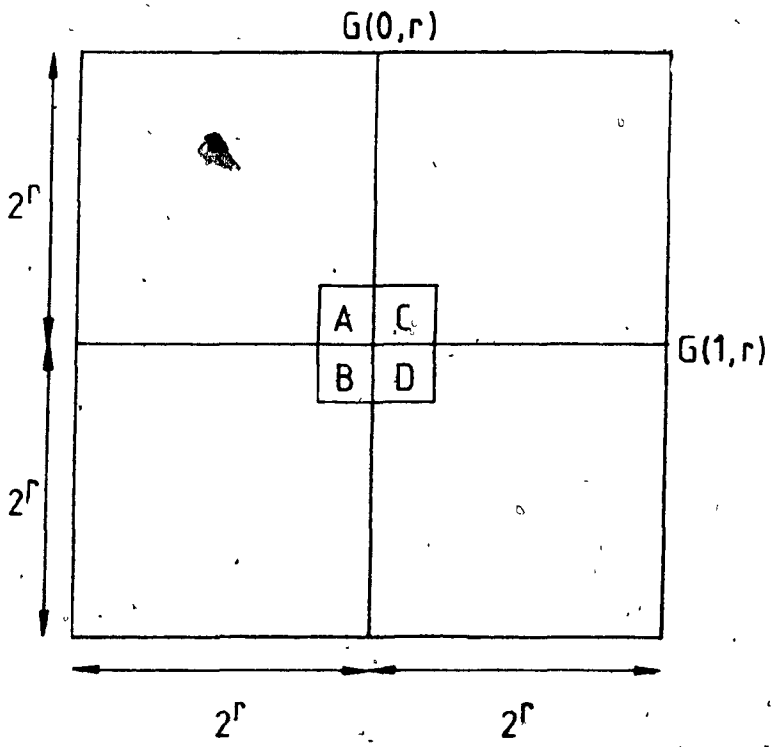


Figure 21. Gap sizes for a 2^{r+1} x 2^{r+1} region.

The difference in the shuffle values is $g = s(A) - s(B)$. This can be easily calculated using "grade-school" notation (note that $\langle X:m\rangle = \langle X:m-1 \mid X\rangle$):

$$\begin{array}{l} \langle 0:d-r-1 \mid 1 \mid 2:r-1 \mid 2\rangle \\ -\ \underline{\langle 0:d-r-1 \mid 0 \mid 3:r-1 \mid 3\rangle} \\ \langle 0:d-r-1 \mid 0 \mid 2:r-1 \mid 3\rangle \end{array}$$

So

$$g_L = \langle 0:d-r \mid 2:r-1 \mid 3\rangle$$

$$= 2 \sum_{j=1}^{r-1} 4^j + 3$$

$$= (1 + 2\times 4^r)/3$$

Repeating these steps for points C and D, (working in base 4):

$$s(C) = \langle 0:d-r-1 \mid 3 \mid 0:r\rangle$$

$$s(D) = \langle 0:d-r-1 \mid 2 \mid 1:r\rangle$$

and

$$g_R = s(C) - s(D)$$

$$= \langle 0:d-r-1 \mid 0 \mid 2:r-1 \mid 3\rangle$$

$$= g_L$$

Let $g = g_L = g_R$.

Now consider E and F, the points immediately to the left of A and B, (see figure 22). From the induction hypothesis:

$$s(E) = s(A) - G(1,T(2^r-2))$$

$$s(F) = s(B) - G(1,T(2^r-2))$$

where 2^r-2 is the x coordinate of E and F. (It can be shown that

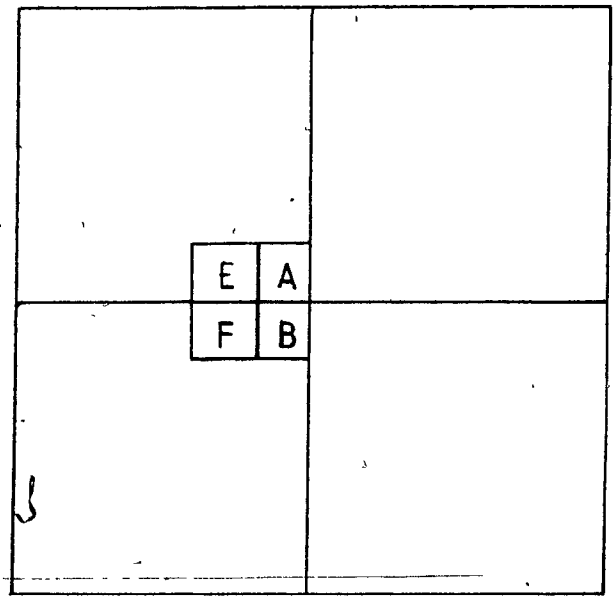


Figure 22. Applying the induction hypothesis.

$T(2^r-2) < T(2^r-1) = r$; thus the IH can be applied.) Therefore

$$s(E) - s(F) = s(A) - s(B)$$
$$= g.$$

We can continue stepping left, (and right from C and D) using the IH at each step to show that

$$s(x,2^r) - s(x,2^r-1) = g, \quad 0 \leq x \leq 2^{r+1}-1.$$

We have established (2) and shown that $G(1,r) = (1 + 2x4^r)/3$.

Proceeding as above, we can establish (1) and show that $G(0,r) = (2 + 4x4^r)/3 = 2\,G(1,r)$.

2.7.2.1.2. Proof for $k \geq 2$

We want to show that for any pair of adjacent points, P and Q, whose coordinates differ (by one) only in the ith position, ($P = (p_0, \ldots, p_{k-1})$, $Q = (q_0, \ldots, q_{k-1})$, $p_j = q_j$, $j \neq i$, $p_i - q_i = 1$),

$s(P) - s(Q)$ depends only on i and p. (3)

We also want a formula for $G(i,T(p_i))$, $i = 0, \ldots, k-1$.

The proof follows the outline of the previous one. We start with the trivial case, a 2^k-bit hyper-cube. Each point has k coordinates, each of which is either 0 or 1. Now write the coordinates of each point as a bit string of length k. These bit strings are the shuffle values $0, 1, \ldots, 2^k-1$. Consider a pair of neighbouring bits. If their coordinates differ in attribute i then the difference in their shuffle values is 2^{k-1-i}. Thus $G(i,0) = 2^{k-1-i}$.

We now perform the induction step. Generalizing from the 2d case, consider a space of $2^{k(r+1)}$ bits and the 2^k bits in the center of the space. The coordinates of one of these bits are $(2^r - e_0, \ldots, 2^r - e_{k-1})$ where each e_j is either 0 or 1.

Our IH is that (3) holds for G(i,s), i = 0, ... , k-1, s < r in the sub-region $(0{:}2^r-1, \ldots, 0{:}2^r-1)$, (corresponding to the quadrant containing bit B in the previous section). The IH holds in all other sub-regions if

$$G(i,T(x+2^r)) = G(i,T(x))$$

for i = 0, ... , k-1, x = 0, ... , 2^r-2. It is sufficient to show that

$$T(x+2^r) = T(x).$$

Intuitively, addition of 2^r does not affect the r least significant bits, (those bits representing x). More formally: Since $x < 2^r-1 = \langle 0{:}d-r-1 \mid 0 \mid 1{:}r\rangle$, $x = \langle 0{:}d-r-1 \mid 0 \mid *{:}r\rangle$ where each * is either a 0 or a 1 (but at least one is a zero). Clearly,

$$x+2^r = \langle 0{:}d-r-1 \mid 1 \mid *{:}r\rangle.$$

By inspection, $T(x) = T(x+2^r)$ since both x and $x+2^r$ have the same suffix, $\langle *{:}r\rangle$, which contains at least one zero.

Now that the IH has been established in all sub-regions, consider a pair of points, P and Q that are neighbours but lie in different sub-regions due to the split of attribute i. Each p_j and q_j is either 2^r or 2^r-1. We will arbitrarily set $p_i = 2^r$ and $q_i = 2^r-1$. s(P) and s(Q) can be expressed in base 2^k notation:

$$s(P) = u_0 \ldots u_{d-1}$$

$$s(Q) = v_0 \ldots v_{d-1}$$

where each u_j and v_j is a base 2^k digit, and can therefore be represented by k bits. Then, since

$$(2^r-1)_2 = \langle 0:d-r-1 \mid 0 \mid 1:r \rangle$$

and

$$(2^r)_2 = \langle 0:d-r-1 \mid 1 \mid 0:r \rangle$$

s(P) and s(Q) can be depicted as

	0	...	d-r-2	d-r-1	d-r	...	d-1
s(P)	---0---	...	---0---	---1---	---0---	...	---0---
s(Q)	---0---	...	---0---	---0---	---1---	...	---1---

More formally (let u_{jm} and and v_{jm} denote the mth bit of u_j and v_j respectively):

$$u_{jm} = v_{jm},\ j = 0, \ldots, d-1;\ m \neq i$$

$$u_{ji} = v_{ji} = 0,\ j = 0, \ldots, d-r-2$$

$$u_{d-r-1,i} = 1$$

$$v_{d-r-1,i} = 0$$

$$u_{j1} = 0,\ j = d-r, \ldots, d-1$$

$$v_{j1} = 1,\ j = d-r, \ldots, d-1$$

Computing the difference in these shuffle values and expressing the result in base 2^k:

$$g = s(P) - s(Q)$$

$$= \langle 0:d-r-1 \mid (2^{k-1-i}-1) \mid (2^k-2^{k-1-i}-1):r-1 \mid (2^k-2^{k-1-i}) \rangle$$

$$= 2^{k-1-i}2^{kr} + \sum_{j=1}^{r-1} [2^k-2^{k-1-i}-1]2^{kj} + (2^k-2^{k-1-i})$$

$$= \frac{2^{k-1-i}}{2^k-1} [(2^k-2)2^{kr} + 1]$$

Since this result is independent of e_j, $j \neq i$, it applies to all 2^{k-1} pairs of points separated by the split of attribute i.

The result also applies to any pair of points on either side of the same boundary within the same $(2^r)^k$-bit sub-cube:

In travelling from P to some other point in the same sub-cube, P′, we cross gaps of size G(w,t), $0 \leq w \leq k-1$, $t < r$. (That $t < r$ is obvious from the binary representations of the coordinates.) The path V from P to P′ can be represented by a sequence of crossed boundaries, (w_p,t_p). By the IH, the same gaps are encountered in travelling from Q to Q′, (the neighbor of P′ across the boundary corresponding to G(i,r)). Thus,

$$s(P') = s(P) + \sum_{p \in V} G(w_p,t_p)$$

and

$$s(Q') = s(Q) + \sum_{p \in V} G(w_p,t_p)$$

so $s(P') - s(Q') = s(P) - s(Q) = g$. This establishes (3) and shows that

$$G(i,r) = \frac{2^{k-1-i}}{2^k-1}\,[(2^k-2)2^{kr} + 1]$$

2.7.2.2. Cost of partial match queries

The data is partitioned into pages. Suppose the shuffle values of the tuples are distributed so that every set of b consecutive z numbers: mb, mb+1, ... , (m+1)b - 1, contain c

tuples, (where c is the capacity of one page). Then the pages impose a partitioning on the sequence of z numbers (and on the space). Each page covers

$$b = \frac{c2^{kd}}{n}$$

bits of the space, where n is the number of tuples in the relation. Figure 23 shows the partitioning for b = 5. Some pages cover two distinct regions of the space as indicated by the page numbers (e.g. pages 2 and 7 in figure 23).

In processing a partial match query, all the pages covering the query must be retrieved. (E.g. for the partial match query x = 3, in the above diagram, pages 3, 4, 6 and 7 would be retrieved.)

We will establish an upper bound for the number of page reads required to process a partial match query (given that each page covers b bits).

Consider a 2d space and a partial match query on attribute 0. The gap sizes encountered along attribute 1 are 1, 3, 1, 11, 1, 3, 1, (as in figure 19).

In visiting the bits of the space covered by the query, gaps of various sizes are encountered. Each gap may contain zero, one or more page boundaries. In crossing a gap without a page boundary, a page read is not generated. Crossing a gap with at least one page boundary generates exactly one page read. Suppose for concreteness that b = 5. Then gaps of size 1 and 3 may or may not generate page reads. The gap of size 11 certainly will (since b < 11). This observation, combined with the results of the previous section, imply a certain regularity in the partitioning

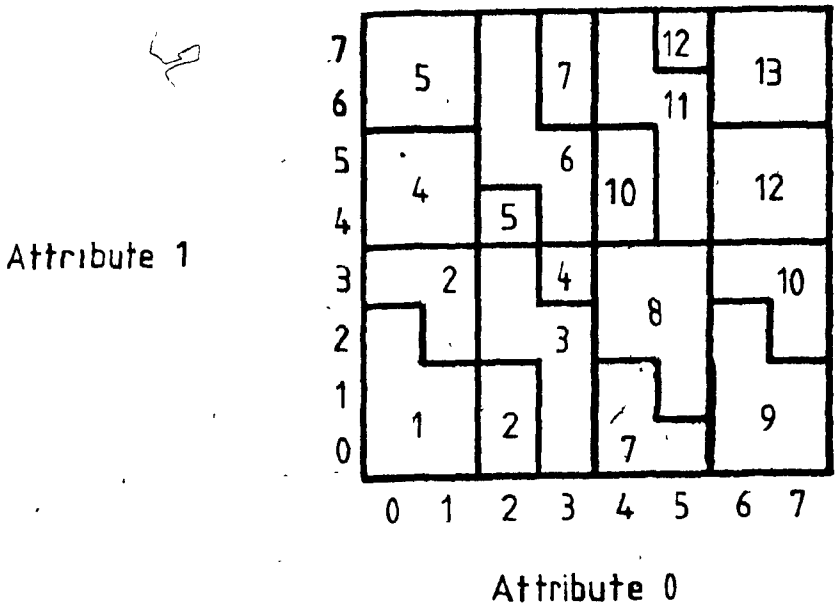


Figure 23. Partitioning the space into pages.

of the space.

The space can be partitioned by a regular grid. The boundaries of this grid represent gaps of size greater than or equal to b. When crossed they are certain to generate page reads. Figure 24 demonstrates this grid for b = 5. Each sub-region of this grid is a "chunk". In order to derive an upper bound for the cost of a partial match query we will proceed as follows:

- Find a constant which is an upper bound for the number of pages in a chunk.
- Find the number of chunks accessed in the processing of a partial match query.

The product of these quantities yields the result (since, to process a chunk, we have to read every page of the chunk in the worst case).

We can place more grid lines in the space to simplify the analysis. The effect of this is to "force" more page reads. This technique is valid since we are interested in an upper bound. We therefore place extra grid lines so that each chunk is a hyper-cube, (see figure 25).

2.7.2.2.1. Size of a chunk

The finest sub-division of an axis (before "extra" grid lines were placed) was on attribute 0 (since G(i,r) increases as i decreases). Solving

$$G(0,x) = b$$

yields the size of a chunk's side.

$$\frac{2^{k-1}}{2^k-1}\left[(2^k-2)2^{kx} + 1\right] = b$$

$$x = \frac{1}{k}\log\left[\frac{b(2^k-1) - 2^{k-1}}{2^{k-1}(2^k-2)}\right]$$

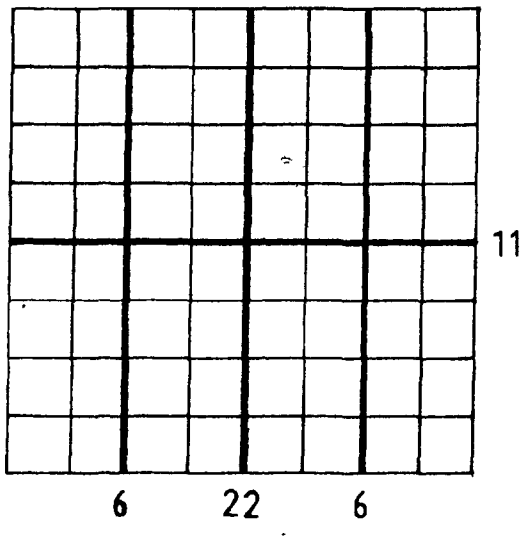


Figure 24. Grid for b = 5.

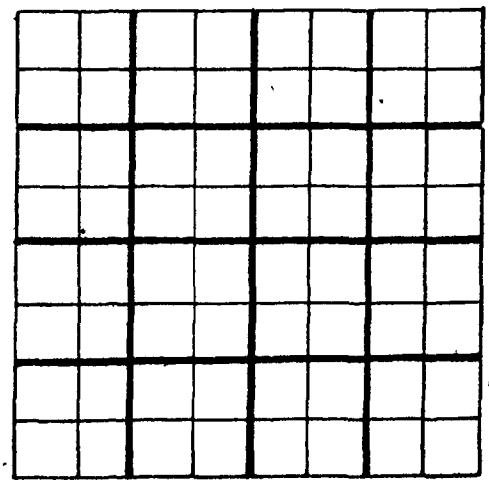


Figure 25. Grid for b = 5 with extra grid lines.

where the side has length 2^x. To get an upper bound on chunk size define

$$u = \lceil x \rceil$$

(since the x in G(0,x) is an integer). Thus

$$u \leq x + 1$$

and the size of the chunk is

$$S \leq 2^{ku}$$

$$\leq \frac{2^{k+1}-2}{2^k-2} b$$

That is, each chunk contains, (depending on k), no more than 2 or 3 pages, since, within each chunk, the z numbers are consecutive. This is guaranteed since each chunk is a kd hypercube resulting from the even splitting of each axis.

2.7.2.2.2. Number of chunks read

Each axis has been divided into 2^{d-u} pieces (during the construction of the chunks). A partial match query on t attributes covers

$$PM = 2^{(k-t)(d-u)}$$

chunks. Using the value of u given above,

$$PM = 2^{(k-t)(d-1)} \left[\frac{2^{k-1}(2^k-2)}{b(2^k-1) - 2^{k-1}} \right]^{1-t/k}$$

Since $2^{k-1} \leq 2^k-1$, $b(2^k-1) - 2^{k-1} \leq (b-1)(2^k-1)$. Also, (2^k-2) /

$(2^k-1) \leq (2^k-1) / 2^k$.

$$PM \leq 2^{(k-t)(d-1)} \left[\frac{2^k-1}{2(b-1)} \right]^{1-t/k}$$

In practice, $2^k << b$ so $(2^k-1) / (2b-2) < (2^k+1) / 2b$

$$PM \leq 2^{(k-t)(d-1)} \left(\frac{2^k+1}{2}\right)^{1-t/k} \left(\frac{1}{b} \right)^{1-t/k}$$

Recall that $b = 2^{kd} c/n$ and $P = n/c$.

$$PM \leq 2^{-(k-t)} \left(\frac{2^k+1}{2} \right)^{1-t/k} P^{1-t/k}$$

Each chunk contains no more than S/b pages:

$$PM \leq \frac{S}{b} 2^{-(k-t)} \left(\frac{2^k+1}{2} \right)^{1-t/k} P^{1-t/k}$$

$$= O(P^{1-t/k}) \text{ pages.}$$

This cost is the same as for the kd trie.

2.7.2.3. Cost of range queries

The number of chunks covered is

$$R \leq \prod_{i=0}^{k-1} (f_i 2^{d-u_i} + 1)$$

where f_i is the fraction of attribute i covered by the query.

$$R \leq 2^{k(d-u)} \prod_{i=0}^{k-1} f_i + \text{low order terms}$$

$2^{k(d-u)}$ is the total number of chunks and $\prod f_i$ is V, the "volume" of the query. Since each chunk has no more than S/b pages,

$$R = O(VP).$$

3. An ISDS based on linear hashing

Linear hashing [Litw80] is a dynamic hashing method. It provides access to the primary page of any bucket in one access. But linear hashing is not an ISDS because it is not order preserving. That is, the successor of r, a record in the file, is not related to the address of r. Other hashing methods can be made order-preserving in a trivial way: by using the hash function $h(r) = \lfloor r/s \rfloor$ where r is an integer and s is a scaling factor. Linear hashing requires the use of hash functions that make this strategy impossible.

In this section, we propose two variations of linear hashing which are order-preserving. Both random and sequential accessing will then be possible. Such a data structure is very useful: it is functionally equivalent to a Btree but should have better performance because it is based on hashing.

Since these variations of linear hashing are order-preserving, they are ISDSs and the techniques of section 2 can be used to create the corresponding ZMDSs. Since this process has been discussed in detail, we restrict our attention to ld data: the data to be stored is a set of integers in a given range.

Little will be said about the performance of these new data structures. We expect that the performance will be typical of hashing methods, (e.g. a random access should usually cost one disk access). In the worst case, the behaviour is almost never worse than that of a Btree. The only exception is that an insertion will occasionally cost $O((\log(n))^2)$ if the data is highly clustered.

3.1. A variation of linear hashing

3.1.1. Linear hashing

A description of a special case of linear hashing is a prerequisite. The data to be stored consists of records of d bits each. Record r can be regarded as the integer $<r_0 | \ldots | r_{d-1}>$ where r_i is the ith bit of the record. The records are to be stored in buckets 0, 1, ... The number of a bucket will also be the address of the bucket's primary page. Overflow pages are allocated from a separate address space.

The file is accessed using hash functions of the form

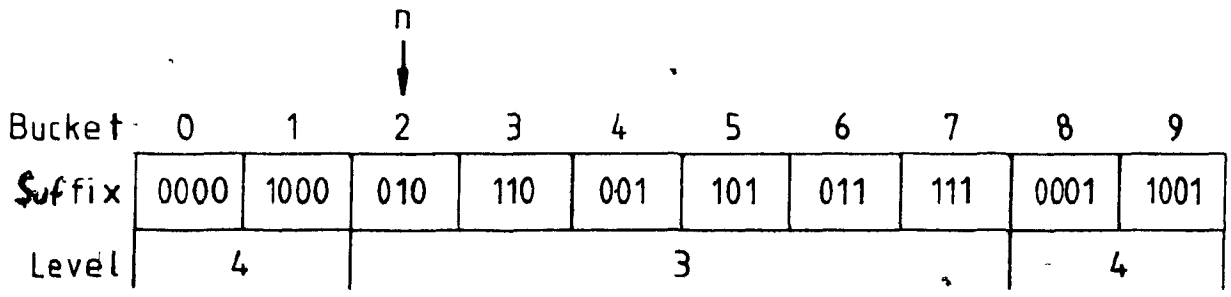
$$h_i(r) = r \bmod 2^i$$

The value of i is one of two consecutive integers, m and m+1, where m is the level of the file. A pointer to the file, n, indicates whether h_m or h_{m+1} should be used, (see figure 26).

Buckets 0 through n-1 and 2^m through $2^m + (n-1)$ are at level m+1; buckets n through $2^m - 1$ are at level m. n is a pointer to the next bucket to be split. It travels from left to right so that every bucket is split in turn. When bucket n is split, its records are distributed between buckets n and $2^m + n$, both of which will then be at level m+1, (since n was incremented). When n reaches 2^m all buckets are at level m+1; n is reset to 0 and starts travelling right again.

A record is assigned to a bucket based on the value of r_{d-m}; the mth least significant bit. So all records in a level m bucket agree in their last m bits.

A bucket is split (and n is incremented) whenever a record hashes to a full primary page, (i.e. there is a collision). The bucket that is split is not, in general, the one involved in the



m = 3
n = 2

Figure 26. Linear hashing.

collision. But eventually, every bucket will be split and (hopefully) the overflow pages will be reclaimed. If splitting creates empty buckets, (because either all or none of the records moved), and the load factor threatens to become "too low", the split is suppressed.

Litwin claims that a linear hash file can also shrink [Litw80] although he does not give the deletion algorithm. It is not difficult to imagine how deletions might be handled. For example, when the overflow pages of any bucket become empty, buckets n and $n + 2^m$ could be combined and n decremented.

To locate a record, r, Randac(r) (for "random access") is called to locate the bucket. We are not concerned with searching within the bucket. Randac returns the number of the bucket containing r.

Randac(r)
Offset(r) locates the rank of record r within the bucket.

```
B := h_m(r)
if B < n
then    (* B has been split to give two buckets on level m+1 *)
   B := h_{m+1} (r)
end
return(B,offset(r))

end Randac
```

Generally, the range of $h_m(r)$ is $[0, 2^m-1]$. That is, h_m and h_{m+1} both hash to $[0, 2^m-1]$. Consider the buckets in [0, n-1]. Since $n < 2^m$, both h_m and h_{m+1} hash to [0, n-1]. But any bucket in this range has been split so h_{m+1} is the correct function to use.

One attractive feature of linear hashing is that it grows smoothly; one bucket at a time. The growth is "linear". The directory of extendible hashing, on the other hand, grows exponentially: it doubles in size periodically (but there are few

of these expansions). In addition, buckets of extendible hashing split when they become full, requiring an update of one directory entry.

3.1.2. Order-preserving linear hashing, (OPLH)

Consider the partitioning imposed by the hash function $h_m(r) = r \bmod 2^m$. All of the records in a given bucket (at level m) agree in LSB(r,m), the m least significant bits, $<r_{d-m} \mid \ldots \mid r_{d-1}>$.

If, instead, the records agreed in the most significant bits, each bucket would store all of the records of the file that fall in a certain range. The hash table would then be order preserving. Let left(s,k) and right(s,k) denote, respectively, the k leftmost and rightmost characters of string s. Mir($<c_1 \mid c_2 \mid \ldots \mid c_v>$) is the "mirror image", $<c_v \mid \ldots \mid c_2 \mid c_1>$ where each c_i is a single character.

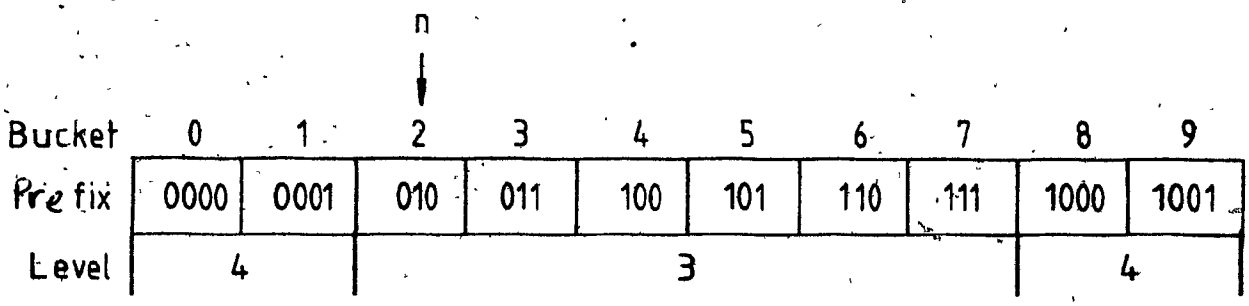
The simplest way to partition the file on the basis of the most significant bits is to store record r in bucket h (mir(r)). That is, the bits are reversed before hashing. Clearly

$$h_m(\text{mir}(r)) = \text{right}(\text{mir}(r),m)$$
$$= \text{mir}(\text{left}(r,m))$$

The bucket number is obtained by reversing the bits of the m bit prefix of the record. Searching and splitting work exactly as for linear hashing. This has to be true since, in effect, we are dealing with another file in which each record, r, has been replaced with mir(r).

If the bits of the prefix were not reversed, i.e. $h_m(r) = \text{left}(r,m)$, then splitting works differently. This alternative is explored in section 3.2.

Figure 27 shows an example of an order-preserving linear hash



m = 3
n = 2

Figure 27. Order-preserving linear hashing.

file. Notice that the mirror image of the m (or m+1) bit prefix of a record at level m (or m+1) matches the m (or m+1) bit representation of the bucket number. So bucket $3 = 011_2$ is at level 3 and stores records with prefix 110_2.

Burkhard has independently discovered OPLH [Burk82]. He also applies shuffling to yield an MDS but his search algorithm is different from the Rangesearch algorithm (and variations) given in section 2. Burkhard does not discuss "sequential" OPLH, (see section 3.2), nor does he address certain problems with OPLH which are dealt with in detail in section 3.4.

3.1.3. Algorithms

The variation of linear hashing described above supports random and sequential accessing. Randac, given in section 3.1.1, can be used except that the argument to h_m and h_{m+1} is mir(r) instead of r.

Suppose that there are N(b) records in bucket b. The address of a record is (b,i) where b is a bucket number and $1 \leq i \leq N(b)$. If $i \geq N(b)$ then the successor of R(b,i), the record at (b,i) is R(b,i+1). The number of records in all buckets is NR. N is the number of buckets.

Consider the problem of finding the successor of R(b,N(b)). The level of the bucket is known: the level, m_b, is m if $b < n$, m-1 otherwise. This bucket represents all records in the range $[\langle mir(b) \mid 0{:}d\text{-}m_b\rangle, \langle mir(b) \mid 1{:}d\text{-}m_b\rangle]$, (recall that b is an m_b bit number). The smallest record above this range is

$$S = \langle mir(b) \mid 1{:}d\text{-}m_b\rangle + 1$$
$$= \langle (mir(b)+1) \mid 0{:}d\text{-}m_b\rangle$$

A search for S (using Randac) will locate b', the bucket

containing the successor of R(b,N(b)). There is one problem: b' may be empty. Repeating the above procedure until a non-empty bucket is found yields the bucket containing the successor of R(b,N(b)). The complete algorithm for sequential accessing is given below.

```
Seqac(p)
p is a pointer to a record, (b,i).

if i < N(b)
then
   i := i + 1
else
   (* find next bucket *)
   repeat until NotEmpty(b)
   if n ≤ b ≤ 2^m -1
   then   (* Level m-1; b is an m-1 bit number. *)
      S := <(mir(b)+1) | 0:d-(m-1)>
   else   (* Level m; b is an m bit number. *)
      S := <(mir(b)+1) | 0:d-m>
   end
   p := Randac(S)
end
return

end Seqac
```

The bucket splitting and joining algorithms are Split and Join.

```
Split()
Bit(<c_1 | ... | c_v>,i) is c_i.

for i := 1 .. N(n)    (* Bucket n is being split. *)
   r := R(n,i)
   if bit(r,m) = 0
   then
      (* the record does not move *)
   else
      move r to bucket n + 2^m
   end
end

n := n + 1
if n = 2^m
then    (* All buckets are at level m *)
   n := 0
   m := m + 1
end
return

end Split
```

```
Join()

n := n - 1
if n < 0
then    (* go down one level *)
   m := m - 1
   n := 2^m - 1
end

n' := n + 2^m
for i := 1 to N(n')
   move R(n',i) to bucket n
end
return

end Join
```

3.2. Another variation of linear hashing, ("Sequential" OPLH)

As suggested in section 3.1.2, another order-preserving variation of linear hashing can be obtained by using the hash function $h_m(r) =$ left(r,m). The method of section 3.1.2 has the following property: $b_m = \langle x_0 \mid \ldots \mid x_{m-1} \rangle$ is the number of a bucket on level m, (each x_i is 0 or 1), and $b_{m+1} = \langle 0 \mid x_0 \mid \ldots \mid x_{m-1} \rangle$ is the number of a bucket on level m+1. Since b_{m+1} is

derived from b_m during a split of the latter, the two buckets will never be present at the same time. This property does not hold if the new hash function, $h_m(r) = left(r,m)$ is used. It is possible to have b_m and b_{m+1} present simultaneously and this requires that the organization of the hash file be reconsidered.

Consider the situation of figure 28. When bucket 0 is split, the third bit will be used to separate the records (since level 3 is being started). The buckets created will be 000_2 and 001_2. But there are now two buckets with the same number, 1: 001_2 and 01_2. Generally, bucket n is being replaced by buckets 2n and 2n + 1. If $n \leq 2^{m-1} - 1$ then $2n + 1 \leq 2^m - 1$ so the created buckets will have the same address as existing buckets.

There is a simple solution to this problem: place all level m+1 buckets in locations following the last level m bucket. Applying this strategy, the file of figure 28, after being split twice, would appear as in figure 29.

Locating a bucket is slightly more complicated: a bucket number can no longer be used as an absolute address. It is now a (base, offset) pair. The base is the absolute address of the first bucket of the level and the offset is the bucket number itself, (thus b_m and b_{m+1} can be distinguished: the offsets are the same but the bases are different).

In figure 29, the base of level 2 is 0 and the base of level 3 is 4. In general, let B_i be the base address of level i. If the file starts at level 0 and $B_0 = 0$ then $B_i = B_{i-1} + 2^{i-1}$ so $B_i = 2^i - 1$.

This organization, as with standard linear hashing, grows by one bucket at each split: when two buckets are added at level m+1, the bucket being split (at level m) is discarded, (e.g.

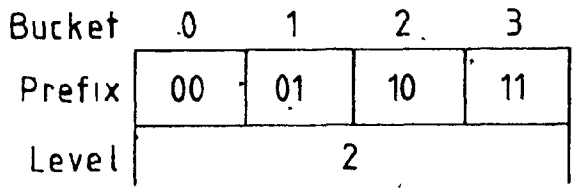


Figure 28. Using $h_m(r)$ = left(r,m)

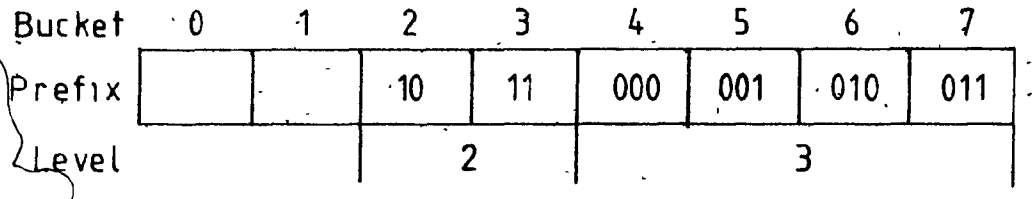


Figure 29. Level m+1 buckets follow level m buckets.

buckets 00_2 and 01_2 of figure 28 do not appear in figure 29). The problem is that the file is constantly "moving forward". In practice, a file has a fixed number of buckets. If these buckets, 0, ... , F-1 are treated circularly, (bucket 0 follows bucket F-1), then the problem is eliminated. The modifications of the algorithms to deal with this circularity are simple. But for simplicity of presentation we continue to work with an open ended sequence of pages, 0, 1, ...

The Randac algorithm of section 3.1.1 can be used but the hash function must be modified to incorporate both components of bucket addresses.

$$h_m(r) = B_m + \text{left}(r,m)$$

returns the required absolute address.

Seqac requires only a slight modification. Bucket b is on level m if $b < B_m$, (instead of $n \leq b \leq 2^m - 1$).

Split places the contents of bucket n (at level m) in buckets 2n and 2n + 1 (at level m+1). The absolute addresses of these buckets are $B_m + n$, $B_{m+1} + 2n$ and $B_{m+1} + 2n + 1$ respectively.

An attractive feature of this variation is that, except for one discontinuity, consecutive ranges are in consecutive buckets (and the corresponding primary pages are physically adjacent). This implies better performance during sequential processing since the disk arm will not have to move very much. The method of section 3.1.2 does not have this property. Because physical sequentiality of logically sequential pages is guaranteed by this method we call it "sequential" OPLH, (SOPLH).

3.3. Overflow

OPLH places severe restrictions on the hash functions that can be used. The hash functions that have been used are left(r,m) and mir(left(r,m)). It is possible that these values, prefixes of records, will be clustered. That is, left(r,m) may not scatter the data very well. So overflow will be more common than with other hashing methods.

So far, almost nothing has been said about how overflow is dealt with. Both Litwin and Burkhard suggest the use of overflow chains [Litw80, Burk82]. A Btree (or variant) is a much more appropriate data structure in the present context. Since OPLH is indexed-sequential, the data structure representing an overflowing bucket should be also.

We have also said very little about the performance of OPLH. Our only concern will be to prevent the performance from being worse than for a Btree, (whenever possible). If the data is distributed uniformly, the performance is better.

We will use a B+tree instead of a Btree: it is a non-homogeneous data structure. I.e. all the records are in the leaves. This results in simplified algorithms. To simplify matters further, a bucket which has not overflowed will be regarded as a B+tree containing one leaf, (i.e. all pointers are null).

A bucket which has not overflowed stores all of its records on the primary page. A bucket which has overflowed stores the root of the B+tree on the primary page and uses pages from a separate address space for the descendents.

The use of a B+tree (instead of a linear list of overflow pages) complicates operations on buckets: Split and Join. It is

essential that these operations preserve the properties of B+trees, (the load factor in particular). In section 3.4 these B+tree operations and others will be referred to. The implementation of these and other B+tree operations will be discussed in section 3.7.

3.4. Multi-level OPLH, (MLOPLH)

3.4.1. Problems with OPLH

An OPLH file may contain an arbitrary number of sparsely filled buckets. This can result in poor performance for sequential accessing. Consider the situation shown in figure 30. To retrieve all the tuples whose prefix is 00_2 buckets 0, 4 and 8 must be accessed. These three disk accesses yield two tuples. If the entire file were at level 2 then bucket 0 would contain the tuples which would be retrieved in one access, (since primary page capacity is 4). But if the file were at level 2, other searches, (e.g. for prefix 100_2), would be more expensive. Furthermore, it would take six joins to reach level 2. So the problem is not solved by joining more frequently.

The situation demonstrated in figure 30 is characterized by the appearance of several sparsely filled buckets. It can occur following a sequence of splits which distribute the tuples unevenly or following repeated deletions concentrated in a few buckets. Since it can occur as a result of deletions, suppressing splits does not solve the problem either.

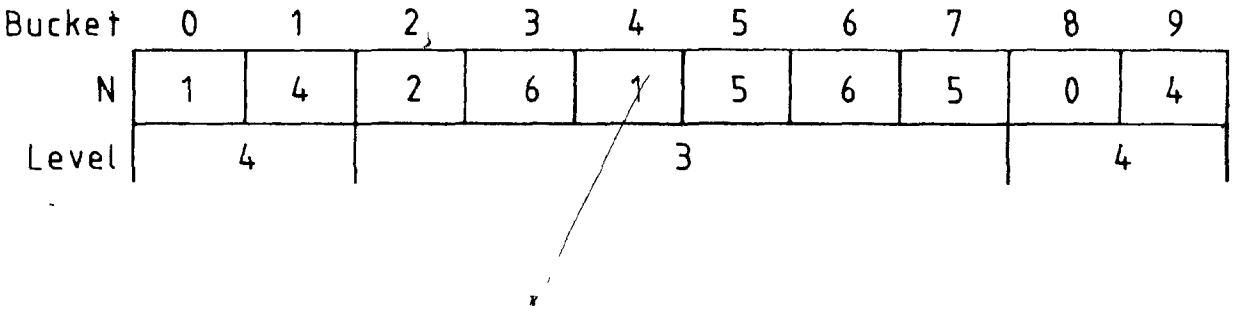


Figure 30. OPLH with sparse buckets. (A bucket is sparse if it contains 0 or 1 record.)

3.4.2. Adding more levels to OPLH

Linear hashing, as described in section 3.1.1, is based on the binary trie: it classifies records according to a sequence of bits, (the suffix). Other data structures based on this idea are extendible hashing [Fagi79], EXCELL [Tamm80], HCELL [Tamm81], trie hashing [Litw81] and, of course, the trie [Fred60, Knut73]. What all of these data structures have in common is the notion of "level". The level of a record is the number of bits used in its classification. Records are usually grouped into buckets (as we are doing). The level of a bucket is the level common to all records in the bucket.

The trie stores a record at the lowest level providing a classification which avoids bucket overflow. The same is true of extendible hashing, EXCELL and HCELL but each of these has a directory with all entries at the same level. Linear hashing and OPLH use no more than two consecutive levels.

The problem with OPLH, described above, would be alleviated if parts of it could be stored at lower levels than normal. E.g. if, in figure 30, the contents of buckets 0, 4 and 8 could be stored in a level 2 bucket, (corresponding to prefix 00_2), leaving the rest of the file at levels 3 and 4, the problem would be solved. Next, we discuss a "multi-level" version of OPLH, MLOPLH. A multi-level version of sequential OPLH (SOPLH) is briefly discussed in section 3.5.3.

3.4.3. Sub-normal buckets

A bucket is sparse if it contains no more than a given number of records (which is a fraction of the capacity of the primary page). A sparse bucket will not be permitted to exist. It will be combined with its brother to form a sub-normal bucket: a bucket whose level is lower than normal. If level(b) < NormalLevel(b) then b is sub-normal. (Level(b) is the level of the bucket and NormalLevel(b) is m or m+1. b and b´ are brothers if level(b) = level(b´) and $|b - b'| = 2^{level(b)-1}$.

Consider the file of figure 30. Buckets 0, 4 and 8 are sparse. To eliminate the problem, buckets 0 and 8 are joined, (they are brothers at the same level), yielding the level 3 bucket for prefix 000_2. Joining buckets 0 and 4 yields a level 2 bucket at address 0, (see figure 31). Note that buckets 4 and 8 could not have been joined first: they were at different levels.

3.4.3.1. Degenerate splits

A split may yield one or two sparse buckets. It is not feasible to refrain from splitting until the situation changes: all further splits are also delayed. Instead, the bucket that should have been split can remain at its current level and n, (the pointer to the next bucket to be split), is advanced. This is a degenerate split. For example, suppose that a split of bucket 2 in figure 31 yields a sparse bucket. The degenerate split leaves the bucket at level 3, (it is then sub-normal; see figure 32).

Bucket	0	1	2	3	4	5	6	7	8	9
N	2	4	2	6	-	5	6	5	-	4
Level	2	4	3	3	-	3	3	3	-	4
Normal level	4		3						4	

Figure 31. Multi-level OPLH: sparse buckets have been eliminated.

Bucket	0	1	2	3	4	5	6	7	8	9	10
N	2	4	2	6	-	5	6	5	-	4	-
Level	2	4	3	3	-	3	3	3	-	4	-
Normal level		4				3				4	

Figure 32. Bucket 2 has undergone a degenerate split.

3.4.3.2. Forcing joins

A bucket can also become sparse following a deletion. When this occurs, the bucket is joined with its brother. For example, if a record is deleted from bucket 2 of figure 31, it becomes sparse. It is then joined with its brother, bucket 6, (see figure 33).

3.4.3.3. A sparse bucket may become non-sparse

A sub-normal bucket can, due to insertions, yield higher level buckets that are both non-sparse. For example if the record deleted from bucket 2 were put back, it would be correct to distribute the records of bucket 2 in figure 33 returning to the situation of figure 31. This operation is similar to, but not the same as a split.

3.4.4. Algorithms

The modifications of OPLH given in sections 3.4.3.1 - 3.4.3.3 are extensive. We now give all the algorithms needed for the implementation of MLOPLH.

3.4.4.1. Random accessing

Since records are not always in the buckets they should be in, (e.g. due to a forced join), some mechanism is required for locating a record. Reading an empty bucket (e.g. bucket 4 in figure 31) is an indication that the records of the bucket have moved to the brother bucket, (and down one level). There is a problem with this approach: In an MLOPLH file containing buckets at many levels, a record may have been moved down several levels. One disk read is required to discover that the records of a

bucket have been moved down one level.

To avoid the disk reads, an array of bits, Moved, can be used: it contains one bit for each bucket. Moved(b) is true iff the contents of bucket b have been moved (to bucket brother(b)). The array entries for the file of figure 33 are shown in figure 34.

Since the space requirement of Moved is only one bit per bucket, it can be kept in primary memory even if the file is quite large.

The Randac algorithm uses Moved to find the level of a given record.

Randac(r)
L is the level of the bucket whose range contains r. Offset locates the rank of the record within the bucket.

```
if h_m(mir(r)) < n
then L := m+1
else L := m

(* Skip over empty buckets *)
while Moved(h_L(mir(r)))
   L := L - 1
end

return((h_L(r), offset(r)))

end Randac
```

3.4.4.2. Sequential accessing

As before, Seqac must construct a successor record and perform a random access. But since there are no sparse buckets the loop which skips over empty buckets can be removed.

Bucket	0	1	2	3	4	5	6	7	8	9
N	2	4	7	6	-	5	-	5	-	4
Level	2	4	2	3	-	3	-	3	-	4
Normal level	4		3						4	

Figure 33. Buckets 2 and 6 have been joined.

	0	1	2	3	4	5	6	7	8	9
Moved	0	0	0	0	1	0	1	0	1	0

Figure 34. The Moved array.

```
Seqac(p)
p is a pointer to a record, (b, i).

if i < N(b)
then i := i + 1
else
   (* b is a number of level(b) bits *)
   S := <(mir(b)+1) | 0:d-level(b)>
   p := Randac(S)
end
return

end Seqac
```

3.4.4.3. Splitting

The Split algorithm must be modified to deal with the creation of sparse buckets. The details of this will be dealt with by the Distribute algorithm. (Distribute will also handle sub-normal buckets becoming "more normal".) The split may be degenerate; Distribute is not called if DistribOK indicates this condition, (see section 3.7.1).

```
Split()

if level(n) = m
then   (* the bucket is not sub-normal *)
   if DistribOK(n) then Distribute(n)   end
else
   (* The bucket is sub-normal. The records will be distributed *)
   (* as soon as insertions to the bucket ensure a *)
   (* non-degenerate distribution of the records. *)
end

(* Increment number of buckets (N) and pointer to next *)
(* bucket to be split (n). *)
N := N + 1
n := n + 1
if n = 2^m
then
   n := 0
   m := m + 1
end
return

end Split
```

3.4.4.4. Joining

Join must be modified to deal with sub-normal buckets. Collect combines the contents of brother buckets (and will be used to deal with buckets which become sparse due to deletion).

Join()

```
N := N - 1
n := n - 1
if n < 0
then
   m := m - 1
   n := 2^m - 1
end

if level(n) = m+1
then   (* the bucket is not sub-normal *)
   Collect(n)
else
   (* The bucket is sub-normal and already at or below level m. *)
end
return

end Join
```

3.4.4.5. Insert a record

When a record is inserted, the receiving bucket may be able to distribute its records without creating sparse buckets. AddToBucket performs the update of the bucket's data structure and is discussed in section 3.7.

Insert(r)
r is the record being inserted. p is the address of the smallest record greater than or equal to r. N(b) is the number of records in bucket b. NR is the number of records in the file.

```
p := Randac(r)
b := bucket(p)   (* the bucket portion of the address *)
AddToBucket(r,p)
N(b) := N(b) + 1
NR := NR + 1
if TimeToSplit(NR, N) then Split()   end
if DistribOK(b) then Distribute(b)   end
return

end Insert
```

3.4.4.6. Deletion of a record

If deletion creates a sparse bucket it is deleted by calling Collect. The algorithm is analogous to Insert.

```
Delete(p)
p points to the record being deleted.

b := bucket(p)
RemoveFromBucket(p)
N(b) := N(b) - 1
NR := NR - 1
if TimeToJoin(NR, N) then Join()   end
if sparse(b) then Collect(b)   end
return

end Delete
```

3.4.4.7. Distribution of records in a bucket

Distribute is called by Split and Insert when the records of a bucket b at level level(b) are to be distributed to brother buckets at level level(b) + 1. The records are classified according to the value of the level(b) + 1st bit. SplitBucket(b) performs this classification.

Note that Distribute is recursive. Consider the situation of figure 35. Bucket 0 stores all the records whose prefix is 0_2 but the distribution of these records is biased. As soon as two records with prefix 01_2 are inserted, bucket 0 can be split to yield buckets 0 and 1 (corresponding to prefixes 00_2 and 10_2 respectively). Now, bucket 0 has 24 records. Assuming an even distribution, the distributions shown in figure 36 can occur, (the prefixes are shown). These distributions are performed by the recursive calls.

Bucket	0	1	2	3	4	5	6	7	8	9
N	24*	3	-	4	-	3	-	2	-	3
Level	1	4	-	3	-	3	-	3	-	4
Normal level	4		3						4	

* 00: 24
01: 0

Figure 35. A large cluster of records in bucket 0.

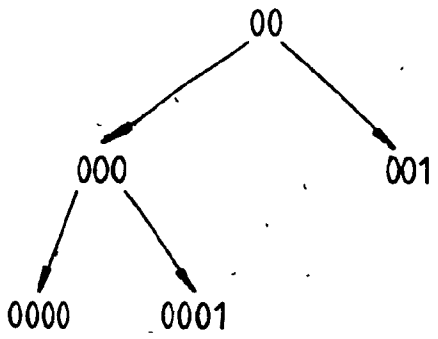


Figure 36. Distribution of the bucket 0 records.

Distribute(b)

```
if level(b) ≤ m
then    (* distribution is possible *)
   oldlevel := level(b)
   b´ := brother(b)
   SplitBucket(b)
   level(b) := oldlevel + 1
   level(b´) := oldlevel + 1
   if DistribOK(b) then Distribute(b)   end
   if DistribOK(b´) then Distribute(b´)   end
else
   (* brother(b) does not exist yet *)
end
return

end Distribute
```

3.4.4.8. Collection of brother buckets

Collect is called by Join and Delete to combine a sparse bucket with its brother. JoinBuckets manipulates the buckets' data structures.

Collect is also recursive. Consider a Collect of bucket b at level L and b´ = brother(b) at level L´ > L. Before the Collect can occur, b´ must be at level L. To ensure this, b´ is collected, even though neither b´ nor brother(b´) (≠ b) is sparse.

```
Collect(b)

b´ := brother(b)

(* Bring b´ to level of b *)
if level(b´) < level(b)
then    (* interchange b and b´ *)
   b :=: b´
end
while level(b) < level(b´)
   Collect(b´)
end

(* Place result in lower bucket b *)
if b´ < b then b :=: b´    end
JoinBuckets(b,b´)
level(b) := level(b) - 1
return

end Collect
```

3.5. Problems with MLOPLH

By eliminating sparse buckets, the problem of potentially poor performance for sequential processing has been solved. For searching, MLOPLH is not worse than the Btree. It can be expected to have better performance since most buckets will not overflow.

Updates occasionally generate calls to Distribute or Collect. These calls can generate a lot of work; a scan of the entire file is required in extreme cases. In this section, the problem is explained and a solution is proposed.

3.5.1. The problem

Consider the situation of figure 37. A rash of deletions from bucket 0 have caused it to become sparse. It must be collected. Its brother is bucket 1 which is at level 4. Bucket 1 must therefore be collected, putting it at level 3. By the time that bucket 1 reaches level 1, the contents of buckets 3, 5, 7 and 9 will have been moved to bucket 1. Half of the buckets have been

Bucket	0	1	2	3	4	5	6	7	8	9
N	1	7	-	8	-	7	-	8	-	8
Level	1	4	-	3	-	3	-	3	-	4
Normal level	4		3						4	

Figure 37. Half of all the buckets are involved when bucket 0 is collected.

affected.

A similar problem plagues Distribute. Generally, Collects and Distributes involving buckets at very low levels: 0, 1, 2, etc. involve very large fractions of the file.

The cost of moving records from one bucket to another is a minor concern: the records in brother buckets b and b' can be merged in time O(log(N(b)) + log(N(b'))), (see section 3.7). This is possible because all the records in bucket b are smaller than those in bucket b', (and only the "edges" of the B+trees have to be modified).

The major concern is the number of buckets whose contents have to be moved. For example, collecting a bucket on level L may cause as many as about $N / 2^L$ buckets to be accessed.

3.5.2. The solution

Clearly, the solution involves placing a lower bound on the level of a bucket. For example, if the lowest level permitted is 5, then a bucket being collected will be at least on level 6 and no more than 1/64th of the buckets will be involved in any Collect.

A consequence of this strategy is that a few sparse buckets may exist. In general, if the lowest level permitted is L then there may be as many as 2^L sparse buckets.

A few modifications to MLOPLH are required to make this work:

- The file is initialized with 2^L empty buckets, (instead of one empty bucket).
- Sparse buckets must be kept track of. The address and level of each must be known. When a new sparse bucket is created for any reason, and the number of sparse buckets exceeds 2^L, the sparse

bucket at the highest level is collected, (this is the cheapest one to collect).

- The modifications ot the Split, Join, Distribute and Collect algorithms are straightforward.

It is interesting that there is a tradeoff between the worst case costs of sequential processing and updating: an update may require accessing $N / 2^L$ buckets and there are up to 2^L sparse buckets which can slow down sequential processing.

3.5.3. Sequential MLOPLH

All of the modifications of OPLH through section 3.5.2 can also be applied to SOPLH. The only difference is that brother buckets b and b', $b < b'$, when collected would reside in bucket b' instead of bucket b.

3.6. Performance

MLOPLH is a complicated but potentially faster alternative to the Btree. Random accessing is definitely faster in MLOPLH: in the worst case a Btree containing the entire file has to be searched. Sequential accessing will be about the same for both data structures unless a large number of sparse buckets are permitted in an MLOPLH file.

It is difficult to compare the performance of dynamic operations. The frequency of "exceptional" updates (e.g. involving page splits for a Btree and bucket distribution for MLOPLH) may not be the same for the two data structures. Also, it is difficult to compare the costs of executing the "exceptional" updates.

It is clear that in some cases MLOPLH may access a fixed fraction of the buckets: this is definitely slower than the slowest Btree update.

It is also clear that a lot of work is necessary before MLOPLH can be recommended as the successor of the Btree. This work falls in three areas:

1) Fine tuning: selecting values for parameters (e.g. threshhold for sparseness, number of sparse buckets permitted).

2) Studying various strategies for administrative details such as when to split and when to join, (i.e. the TimeToSplit and TimeToJoin algorithms).

3) Experiments comparing MLOPLH and the Btree.

3.7. B+tree algorithms

We now discuss the various B+tree operations required by MLOPLH. AddToBucket and RemoveFromBucket are the normal B+tree insertion and deletion algorithms. The others are discussed below.

We use the B+tree instead of the Btree because it simplifies the algorithms which follow. For simplicity of explanation we suppose that a discriminator used in an internal node is a complete record, (i.e. an integer of the file). So both leaves and internal nodes have the following format: $[p_0, r_1, p_1, r_2, p_2, \ldots, r_c, p_c]$ where p is a pointer and r is a record. c is the capacity of a page. The records in the subtree pointed to by p_i are greater than or equal to r_i, $i = 1, \ldots, a$, where a is the number of records actually present in the page. The records in p_0's page are strictly less than r_1. For a leaf page, p_i =

null, i = 0, ... , a.

To maintain a minimum load factor of 50%, each page (except possibly the root) must contain at least c/2 records. The algorithms below will preserve this property.

3.7.1. DistribOK

DistribOK examines the contents of a bucket, returning true iff the records can be distributed non-degenerately. If this is possible then Distribute will be called and two non-sparse buckets will be created.

The records in a bucket, b, are divided into two sets: S_i is the set of records whose Level(b) + 1st bit = i, i = 0, 1. $x \in S_0$ and $y \in S_1$ implies $x < y$ since all records in the bucket agree in the first level(b) bits.

A bucket is sparse if it contains fewer than a given number of records, s. The sparseness of S_0 can be decided in O(log n) time, where $n = |S_0 \cup S_1|$ is the number of records in the bucket:

- Find the first record on the leftmost leaf.
- Find the s-1st successor. This is record R(b,s).
- S_0 is sparse if $R(b,s) \in S_1$ since there were fewer than s records in S.

The sparseness of S_1 can be decided in a similar way.

3.7.2. SplitBucket

The S_0 records will form one bucket and the S_1 records will form another. These are buckets B_0 and B_1 respectively. On each

level of the B+tree there is one discriminator on one page which acts as a "boundary" between S_0 and S_1 on that level. Each page of this type will be split yielding one page for each bucket's B+tree. So splitting of the B+tree can be achieved in one pass of the set of boundary pages from the root to the leaf boundary page.

Splitting a page may yield 0, 1 or 2 pages with too few records. These pages will be called deficient. The "right fringe" of B_0 and the "left fringe" of B_1 have to be adjusted to eliminate such pages. For example, a deficient page on the right fringe of B would be combined with its left neighbour, (possibly a brother). If this causes overflow, it is dealt with in the normal way (and the net effect is a "rotation" of records into the deficient page).

We are assuming that the neighbour of a page can be found quickly. This is possible if the pages on each level are organized into a doubly linked list.

The algorithm consists of three parts.

1) Find the path through the B+tree containing the boundary pages.

2) Split the B+tree through the boundary pages.

3) Eliminate deficient pages from the right fringe of B_0 and from the left fringe of B_1.

SplitBucket(b)

```
FindBoundary(b,stack)
SplitOnBoundary(stack,A0,A1)
FixRightFringe(A0,R0)
FixLeftFringe(A1,R1)
root(b) := R0
root(brother(b)) := R1
return

end SplitBucket
```

3.7.2.1. FindBoundary

This algorithm is straightforward. On each level, the first S_1 record is located and its address is placed in stack.

FindBoundary(b,stack)

```
P := b
stack := EmptyStack()
repeat until p = null
   j := FindFirstS (P)
   push((P,j),stack)
   P := pj
end
return

end FindBoundary
```

3.7.2.2. SplitOnBoundary

Each page in stack is split at the indicated position. The resulting pages are entered into stacks A_0 and A_1 which will be used in the elimination of deficient pages.

SplitOnBoundary(stack,A_0,A_1)
Page P is split to yield pages P and P´. C and C´ are the previous values of P and P´, (i.e. from the lower level).

```
C := null
C´ := null
repeat until Empty(stack)
   (P,j) := pop(stack)
      (* P = [p0, r1, p1, ... , rj, pj] *)
   GetPage(P´)
   P := [p0, r1, p1, ... , rj-1 , pj-1 ]
      (* note: pj-1 = C *)
   P´ := [C´, rj , pj , ... , rd , pd ]
   push(P,A0)
   push(P´,A1)
   C := P
   C´ := P´
end
return

end SplitOnBoundary
```

3.7.2.3. FixRightFringe

We now show how to eliminate deficient pages from the right fringe of B_0. A similar algorithm is used for B_1. The fringe pages were stored in a stack. Popping the stack yields the fringe pages, starting with the root. Each page will be combined with its left neighbour. If overflow results, it is handled in the usual way. Finally, there may be some empty pages at the top of B_0. These are eliminated and the root is stored in R.

FixRightFringe(A,R)

```
R := top(A)
repeat until Empty(A)
   P := pop(A)
   if deficient(P)
   then
      if P has a left neighbour
      then
         P := Lcombine(P)
      end
   else
      (* P is OK *)
   end
end

(* Find the root *)
while R has no records
   P := R
   R := page(R).p0
   ReturnPage(P)
end
return

end FixRightFringe
```

Lcombine concatenates two pages and returns the resulting page. To do this, a discriminator must be located.

This involves O(log n) time where n is the number of records in B , (since the discriminator is the leftmost record in P's subtree). Since the search for a discriminator may occur O(log n) times, the running time of FixRightFringe (and therefore SplitBucket) is $O((\log(n))^2)$. (But recall that a B+tree exists only in case of overflow and that $n \leq NR$, the number of records in the entire file.)

3.7.3. JoinBuckets

Two buckets, b and b' are to be joined. The records in one bucket precede all those in the other. Suppose that b contains the smaller records. In general, the heights of b and b' are not the same. Let h be the height of b and let h' be the height of

b'. For purposes of explanation assume that $h \geq h'$.

An essential property of B+trees is that all leaves are at the same level. So the leaves of b and b' must be at the same level. Therefore, the root of b' will be at the same level as level $h - h'$ of b, (the root is at level 0).

Now consider the root of b', P', and its neighbour in b: P, the rightmost page of level $h - h'$. The layouts of P and P' are $[p_0, r_1, p_1, \ldots, r_a, p_a]$ and $[p'_0, r'_1, p'_1, \ldots, r'_{a'}, p'_{a'}]$ respectively. To merge the two B+trees, replace node P by $[p_0, r_1, p_1, \ldots, r_a, p_a, M', p'_0, r'_1, p'_1, \ldots, r'_{a'}, p'_{a'}]$ where M' is the minimum record in b'. This can be found in logarithmic time. Overflow resulting from this concatenation is handled in the usual way.

The main results of this chapter are summarized in chapter 8, sections 1 - 3.

Part Two

Transaction Processing

Chapter 4

A Survey of Recovery Techniques and Concurrency Control Systems

In this chapter we review the relevant literature on recovery and concurrency control. It may seem odd that these subjects are dealt with together: they have traditionally been considered to be separate problems. This point of view has contributed to the complexity of implementations.

Recently, the database community has noticed that the two problems are not independent: parallelism can be enhanced by using slightly out of date information which is kept for purposes of recovery.

In 1978, Reed considered the relationship between the two problems in a more general context [Reed78]: A relation can be seen as a sequence of versions. Each update generates a new version. Recovery can be achieved by bringing a recent version up to date. Concurrency can be enhanced by permitting a user to read a selected (old) version of the relation.

Some of Reed's ideas have been incorporated into the design of the Local Database Manager (LDM) [Chan82]. Their design resembles ours in some ways but there are large differences also.

In sections 1 and 2, recovery and concurrency control techniques are surveyed. In section 3, the designs of System R, INGRES and LDM are examined.

1. Recovery

A database system must be able to recover from a system failure or "crash". A soft crash leaves the contents of secondary storage intact. A hard crash damages the contents of secondary storage. Following recovery, the state of the relations must be correct and interrupted transactions must be backed out, (i.e. all the changes due to the incomplete execution are reversed), and re-executed.

In section 3 the recovery systems of three relational database systems will be described. What follows is a description of techniques from which recovery systems can be built. These techniques have been summarized in [Verh78].

An obvious technique is to take "checkpoints" (make backup copies) periodically. Following a crash a previous state of the system will be available. A problem with this technique is that it is expensive to copy large relations.

The audit trail [Bjor75] (or "log" or "journal") is a chronological list of operations carried out on the database. It serves a variety of purposes: it can be used to bring an old copy of the database up to date following a crash; it can be used to back out transactions (in case of deadlock or a failure); it can be examined to verify that policies regarding the use of the system are observed. The audit trail is, therefore, likely to be a component of any database system even if it is not used for recovery purposes.

The checkpoint is often used in conjunction with a log to restore the current state of the database. This has the undesirable property that transactions which committed after the checkpoint and before the crash have to be re-executed. We will

propose a soft crash recovery system that does not have this problem: checkpoints always include the results of all committed transactions.

When a log is used for recovery, the database contains three components: the current version of the database; an old version of the database; the log which allows the old version to be brought up to date. The differential file [Seve76] is a related idea. One version of the database is kept. This is the static "master file", (an old version of the database). The "differential file" stores the changes accumulated since the master file was created. The difference between the log and the differential file is the following: the log records transactions and is not searched in the evaluation of a query (since the current version of the database is available). The differential file stores the actual tuples being changed (old and new versions) in a form suitable for efficient searching. It is searched to obtain any tuples overriding tuples in the master file. Lorie has pointed out a number of problems concerning the performance of a database based on differential files [Lori77]. However, if implemented properly, all of these problems can be solved, (see chapter 5 section 1.2). The differential file will play a central role in our design for the PDB.

Another method of maintaining an old copy of a relation is to store both original and updated versions of the pages storing the relation. (This is the backup/current version method. It is used in System R [Astr76] and is described in detail in [Lori77]. See section 3.) Two directories to the pages are stored. One gives access to the current version; the other represents the old version. Unchanged pages will have matching entries in the two

directories. To create a new checkpoint, the current directory is copied to the old directory. This mechanism protects against failures that leave storage intact.

To ensure correctness of processing, multiple copies of the data can be stored. If all copies are not in agreement, (except during the actual updating of the copies), the majority is assumed to be correct. This method has not been used in any database systems that we know of but has been used in other applications. An obvious drawback of this method is the cost of keeping several copies current.

Careful replacement is a technique of rewriting updated data. The idea is to avoid updating in place: the new version is written in a different location from that of the original. When this has been done correctly the original can be returned to free storage. This method should not be used for files whose pages contain pointers directly to other pages since, when a logical page is updated, its physical address changes [Verh77]. (If one level of indirection is used, e.g. the page map of System R, this problem can be avoided.)

In chapter 5 we will propose the use of the differential file for soft crash recovery. The motivation is that the differential file simplifies both recovery and concurrency control and has other advantages concerning performance. The differential file can be seen as storing all previous states of a relation (since the creation of the MF).

2. Concurrency control

To improve the performance of a database system, several users should be able to access the database simultaneously. If, in such an environment, access to the data is not regulated, anomalies can arise [Eswa76]. Concurrency control methods provide the regulation that avoids these anomalies.

2.1. Concurrency control in centralized systems

Several basic notions were introduced in [Eswa76]:

A database is a collection of entities, each of which may be "locked" by no more than one transaction at a time. A lock guarantees exclusive access to the entity by the transaction until the entity is "unlocked". Some locks need not be exclusive: a transaction performing a read of an entity x must exclude writers of x but can allow other readers. A writer must have exclusive possession of an entity.

If the database system contains a single processor then the steps of the transactions will be interleaved. The sequence of steps is a "schedule". A correct (or "consistent" or "serializable") schedule must give the same result as some serial execution of the transactions (i.e. as if there were no interleaving) since transactions are atomic, (see chapter 1 section 3). The interleaving of steps of transactions is controlled by lock and unlock operations. A correct schedule is obtained if the locking protocol described above is observed and if, in addition, all transaction are "two phase". A transaction is two phase if all lock operations are performed before all unlock operations.

For recovery purposes locks should be held until transaction committal: Suppose that a transaction T writes x and then releases its lock on x. x is then read by transaction T'. If T is backed out, (e.g. the user who initiated T cancels the transaction), then T' is incorrect since it has read the value of x created by T. If T had not unlocked x until after backout was complete, T' would have the correct (original) value of x.

The entities that can be locked are relations, attributes, tuples or physical objects such as files and pages. There are some problems in using tuples as the entities to be locked:

- A tuple is a value, not a variable. A tuple is locked in order to update it but when the tuple (i.e. the value) is changed it becomes another tuple, (see chapter 1 section 4). One way around this is to identify tuples by keys. Then only non-key attributes can be modified. Another solution is to create a key known as the "tuple identifier".
- A lock can only control access to tuples that are present. As explained in [Eswa76], it is also necessary to lock "phantom" tuples: tuples that are not present in the relation. Such a tuple, t, must not be inserted by one transaction while another transaction holds a lock that would have included t had it been present.

Other logical entities (relations and attributes) are too "coarse" for locking: parallelism is severly restricted.

In response to some of these problems, the "predicate lock" was proposed [Eswa76]. Predicates can be used to describe the tuples being read and written by a transaction. The predicates of actions from different transactions are in conflict if at least one of them is a write and if there exists a tuple (which may or

may not be present in the relation) that satisfies both predicates. In case of a conflict one of the transactions must wait or be aborted and re-executed later.

Detecting conflict is not always possible unless the class of predicates is restricted (see [Eswa76]). For predicates corresponding to range queries, detecting conflict is trivial.

A variation of the predicate lock is the "precision lock" [Jord81]. The idea is to restrict the class of predicates describing writes to predicates specifying a value for each attribute, (i.e. a point). This is reasonable since, in practice, tuples are written one at a time and all attribute values are known. Now, testing conflict is simple.

Another scheme based on locking can be used when the recovery system maintains both the old and new values of updated objects [Baye80, Stea81] (as in the recovery system of System R [Lori77], or with differential files). Rather than have a transaction wait or abort when a conflict occurs, it is sometimes possible to give the transaction access to old values. This method has the useful property that read-only transactions never have to be aborted when deadlock occurs, (deadlock is discussed below). This is important since, in many applications, read-only transactions are by far the most common.

Little is known about the performance of the various concurrency control methods. Ries and Stonebraker [Ries77, Ries79] have studied the problem of lock granularity. That is, how large should be the entities controlled by locks? "Coarse" locks have lower maintenance costs than "fine" locks but reduce parallelism. They conclude that the size of the locked entity should be dependent on the size of the portion of the relation

accessed by the transaction. A "lock hierarchy" was proposed: Coarse locks for transactions that access a large portion of the relation and fine locks for more selective transactions. A threshhold of 1% is suggested by the simulation. A coarse lock protects a relation and a fine lock protects a tuple.

Although predicate locking was not tested in the simulations, several conclusions indicate that this method might be suitable: for example, the granularity of the lock automatically adjusts to the selectivity of the transaction. Furthermore, the number of read locks is proportional to the number of transactions, not to the number of entities being locked.

Any locking method can give rise to deadlock. A "waits for" graph shows which transactions are waiting for which other transactions to release locks. For example, T -> T' (T waits for T') says that T has requested a lock which T' is currently holding. There are four necessary conditions for the existence of deadlock [Coff71]:

1) Exclusive control of resources: this occurs when a writer sets a lock (which must be exclusive).

2) Waiting for a lock while holding another lock.

3) Transactions are not pre-empted.

4) There is a cycle in the waits for graph.

Deadlock, once detected, can be resolved by aborting one of the deadlocked transactions, (causing condition (3) not to hold). A cycle can be detected by searching the graph (in time proportional to the number of nodes in the graph). The algorithm given in [Aho74] applies to undirected graphs. It is simple to adapt it for directed graphs.

Deadlock can be avoided by causing conditions (2) or (4) not

to hold. By obtaining all locks at once (and not starting if any lock is denied) deadlock is avoided since (2) is violated [Have68].

If the lockable resources are ordered and if locks must be requested in that order then cycles cannot form in the waits for graph (violating (4)).

An entirely different approach to concurrency control is taken by Kung and Robinson [Kung81]. They make the "optimistic" assumption that transactions usually do not conflict. After a transaction completes its processing (but before it is committed) it is validated: conflicts with simultaneously running transactions are tested for. If a conflict is found, the transaction is backed out and re-executed. With this method the overhead of locking is avoided at the cost of running validation tests. Furthermore, deadlock does not occur.

2.2. Concurrency control in distributed systems

Most of the recent work in the field of concurrency control has been concerned with distributed databases. Bernstein and Goodman have decomposed the problem into sub-problems [Bern81]. They claim that most concurrency control methods fall into this framework and differ only in their solutions to the sub-problems.

The database is distributed among several sites. There may be partial or total redundancy of data or none at all. A log at each site lists, in chronological order, the actions performed at that site, (a log is like the schedule of a centralized system). A transaction may be executed at several sites; the log at each site would then contain entries for the transaction.

As with a centralized system, the execution of the transactions in a distributed system must give the same results as the serial execution of some total ordering of the transactions. We will refer to this sequence of transactions as the serialization. Serializability is guaranteed by the following: in every log containing conflicting operations O and O´ from transactions T and T´ respectively, O precedes O´ if and only if T precedes T´ in the serialization (see [Bern81]). In other words, each log places certain constraints on the total ordering of the transactions. T -> T´ indicates that T must precede T´ in the total ordering due to a conflict in some log.

Conflicts may involve a reader and a writer, (T ->rwr T´), or two writers, (T ->ww T´). T -> T´ if T ->rwr T´ or T ->ww T´. The execution of the transactions is serializable if ->rwr and ->ww are acyclic and if there is a total ordering consistent with ->rwr and ->ww.

Now, a concurrency control method can be seen as a composition of two "synchronization techniques" [Bern81]:

1) A technique for read-write (rw) synchronization. That is, a technique for ensuring that ->rwr is acyclic.

2) A technique for write-write (ww) synchronization, (ensuring that ->ww is acyclic).

A complete concurrency control method must, in addition, guarantee that -> is acyclic.

Most of the techniques available for solving rw and ww synchronization fall into two categories: two-phase locking (2PL) and timestamp ordering (T/O).

2PL has been described in section 2.1. Various extensions have been proposed for dealing with the redundant data of a

distributed system.

Deadlock is a problem characteristic of locking (whether or not the system is distributed). As with a centralized system, deadlock can be avoided or detected and broken.

Deadlock can be prevented by assigning priorities to transactions. A transaction T can wait for T' if T has a lower priority. No cycle can result since the priorities create a total ordering. If T has a higher priority it might preempt T'.

To avoid the situation in which a low priority transaction is repeatedly restarted, the priority of a transaction could be related to its age (as in [Stea81]). If the oldest transaction in the system cannot be preempted then no transaction "starves".

Deadlock is detected by searching for cycles in a global "waits-for" graph. This presents problems in a distributed environment since communication with the processor responsible for the graph is necessary to update the graph [Gray78, Ston79].

Menasce and Muntz [Mena79] presented two methods for distributed deadlock detection. One method organizes the processors into a tree. Any change in the waits for graph is propagated up the tree. Deadlock, if it occurs, is detected at the node furthest from the root whose descendents are deadlocked. Their other algorithm does not place the processors in a hierarchy; all processors have the same status. This algorithm is incorrect [Glig80] and to fix it would render it impractical.

A general problem with detection methods is that deadlock can tie up the system between searches of the waits-for graph. This, and the difficulties of detecting deadlock in a distributed system suggest that deadlock free concurrency methods are more suitable. T/O methods are deadlock free.

T/O methods operate by assigning "timestamps" to transactions. A timestamp denotes the time of some significant event: e.g. time of initiation of the transaction or time of committal. The sites of the network are synchronized so that a timestamp at one site is meaningful at another. Timestamps are unique. At any site, conflicting operations are processed in order of increasing timestamp; the serial execution of transactions in order of increasing timestamp would result in the same database state. Deadlock does not occur using T/O but transactions may be aborted in some cases.

Rw synchronization is achieved in the following way: if a transaction with timestamp T attempts to read an object most recently written at time W > T then the read is invalid and the reading transaction is aborted. A similar strategy handles a writer in conflict with the most recent read of an object. Ww synchronization is also based on this scheme. Our design for the PDB permits the use of a simpler ww synchronization technique.

To ensure that all copies of redundant data are updated (or none of them are), writing consists of two phases: a "prewrite" which is a command to each site to make secure copies of the data to be written. The actual write is triggered by a separate command after all prewrites are performed. Since there is some time between the prewrite and the actual write, conflicting reads and writes received in timestamp order are buffered, (so that they can be processed in timestamp order). Conflicting operations received out of order cause restarts as described above.

Complete concurrency control methods use two synchronization techniques: one for rw synchronization and another for ww synchronization. There are 48 combinations involving the

variants of 2PL and T/O summarized in [Bern81] (except that one of them is incorrect, see [Bern81]).

The two techniques must combine to produce a total ordering of transactions. For methods where both techniques are 2PL or both are T/O, this is automatically guaranteed. For hybrid methods, it is possible to generate timestamps for transactions based on the locking scheme. The timestamp represents the lock point: a time between the last lock and the first unlock (recall that locking is two phase). (These generated timestamps also determine the serialization for pure 2PL methods.)

Performance issues are outside the scope of [Bern81] but they are important: T/O methods have a serious drawback. When used for ww synchronization, the timestamp of the last transaction to write each tuple must be known. When used for rw synchronization, the timestamp of the most recent reader of each tuple must also be known. This implies a write (of the timestamp) corresponding to each read. In chapter 5 section 3.1.2 we will discuss these issues in more detail. The differential file will be useful in solving one of these problems. In chapter 5 we will also show how several other synchronization techniques can be implemented using the differential file.

3. Recovery and concurrency control in practice

In this section the recovery and concurrency control systems of System R and INGRES are described. These are "traditional" systems in that they treat the two problems separately. We also discuss LDM which is a "multiversion database" (MVDB). In an MVDB, a state of a relation is derived from its predecessor by the application of all of the updates of some transaction. The availability in an MVDB of past versions of the database simplifies the recovery and concurrency control systems.

3.1. System R

3.1.1. Recovery

The "shadow" mechanism of System R's recovery system has been described in detail in [Lori77]. The recovery of a segment following a crash is considered. Thus, the objects in the segment need not be considered separately. Recovery from both hard and soft crashes is considered.

A page map is used to translate logical page addresses into physical page addresses. The shadow mechanism depends on the use of two page maps: the current version and the shadow (a backup). When a logical page is updated for the first time, the physical page is located using the current page map. Instead of updating that physical page, a new one is allocated and is written with the new version of the page. (I.e. updates are not done "in place".) The current page map is updated to point to the new physical page. The shadow page map still points to the original physical page which has not been changed or discarded. Subsequent

changes affect the new physical page. Since a backup is present, it is safe to update the new physical page in place.

If a soft crash occurs at any point, the shadow version of the page map stores an old state of the database. (The current page map is unreliable following the failure). To recover, then, the shadow page map is copied to the current page map. (If this process is interrupted by another failure, it is simply restarted.)

It is preferable to restore a recent state. To allow this, the shadow page map should be updated periodically. That is, the current page map must be backed up. When this is done, the original, (now out of date), versions of updated pages can be freed. The system then has shadow and current versions that match.

The mechanism described is inadequate for restoring the current state of the segment. For this purpose, a checkpoint and log are used. (For details see [Gray81].) This method makes the shadow mechanism unnecessary, (see chapter 5 section 2.4.1).

To recover from a hard crash, a checkpoint must be available. This checkpoint should be more secure than those used for soft crashes. System R uses a tape. This "long" checkpoint coincides with the last of a fixed number of backup page map saves. The copying to the tape is run as a separate process. The basic scheme for reclaiming pages has been modified to keep old pages (i.e. they are not freed) until they have been copied to tape.

3.1.2. Concurrency control

Transactions may be run at any of three consistency levels. The highest level is the one normally considered in the

literature on concurrency control.

Concurrency control is implemented using locks. Logical objects (segments, relations, TIDs, ranges of index values) can be locked. Physical locks on pages are also necessary: transactions which do not conflict logically may try to update the same page simultaneously.

A dynamic lock hierarchy protocol is used that adapts the granularity of the lock to the selectivity of the transaction. This idea is supported by a simulation study of locking granularity [Ries77, Ries79].

The duration of a writelock is to the end of a transaction (for recovery purposes). The physical lock on a page may be released as soon as the update of the page (in primary memory) is completed.

When a transaction is locked out, deadlock is tested for. If deadlock is found then one of the deadlocked transactions is backed out (using the disk log). The preferred victim is a "young" transaction holding locks of short duration (i.e. physical locks on pages). Such a transaction would be the cheapest to re-execute.

3.2. INGRES

The recovery and concurrency control systems of INGRES have been discussed in [Ston76].

3.2.1. Recovery

The pages of an INGRES file are updated in place. So the soft crash recovery mechanism is necessarily different from that of System R.

Relations and inversions are not updated until after the transaction which generated the updates is committed. Updates are stored in a file. If the transaction is backed out for any reason then the file is discarded.

The use of deferred updates is also motivated by semantic problems connected with updates. Stated briefly, two of the problems have to do with repeated updates to the same tuple (as identified by its TID): a tuple satisfying a search predicate which is updated once might be found again and updated again. This problem is avoided in INGRES since the updates are not registered in the relation until after all updates have been stored in the deferred update file.

Recovery from hard failures is accomplished by using a backup tape to reload a checkpoint and then re-executing the transactions stored in a log.

3.2.2. Concurrency control

The only entities available for locking are the attributes of relations. This coarse granularity was selected to avoid the need for a large lock table in primary memory, (INGRES was originally implemented on a minicomputer with a 64K memory limitation). A

"crude" form of predicate locking is being considered as a replacement. Predicate locks also have small space requirements.

Deadlock is avoided by requiring transactions to acquire all necessary locks before execution starts. If the transaction is blocked then it is postponed for a fixed amount of time. This "predeclaration" strategy is reasonable given the granularity of the locks. It would be less suitable in a predicate locking scheme.

3.3. LDM

The Local Database Manager (LDM) [Chan82] is a multiversion database (MVDB). That is, all of the previous states of each relation (since some previous time) are available. Several LDMs can be connected to form a distributed database system.

The current version of a relation is stored in a segment: a set of pages. Each page of the segment is the head of a list of previous version of the (logical) page. A traversal of such a list encounters the version in reverse chronological order. The previous versions are stored in the version pool.

Eventually the version pool will become full of old versions. Garbage collection is necessary to reclaim "dead pages". A dead page is a version of a page which will never again be referenced. All transactions which could have accessed a given dead page have completed. In [Chan82] an efficient garbage collection scheme is described. A small number of pointers to the version pool, (a circular buffer), are maintained. Pages behind such a pointer are definitely garbage, those in front may or may not be. As transactions finish processing the pointers are advanced.

3.3.1. Recovery

The maintenance of versions makes recovery simple. Following a crash, a scan of the version pool brings the system to a state in which only uncommitted transactions have to be re-executed.

To ensure that a committed transaction is stored securely, all buffers containing updates of the transaction are forced to disk before the transaction ends.

LDMs recovery scheme is very similar to our own. This is due to the fact that both systems are multiversion databases. Our recovery scheme is described in detail in chapter 5.

3.3.2. Concurrency control

Transactions which perform updates use two-phase locking on the set of pages being accessed. Deadlock is searched for periodically. A novel feature (for real systems) is the way in which read-only transactions are handled: they read old versions, (available in the version pool), and are not even considered by the concurrency control system. (This method is described in detail in chapter 5 section 3.2.) An old version is retrieved by searching the current version. For each accessed page, the list of versions is searched for the appropriate version of the page.

Again this facility closely resembles our own design since both systems are multiversion databases. In our design though, the "time" attribute will be treated as another attribute of the relation. The query will be modified to include a restriction on the time attribute. Thus searching for an old version can be achieved by query modification, (a technique proposed for other purposes by Stonebraker [Ston75]).

Chapter 5

The Differential File and its Use in the Physical Database

In this chapter, a new design for the PDB is proposed. This design has two interesting features:

1) It is a multiversion database, (MVDB). This allows the use of simple recovery and concurrency control systems which do not have high overhead costs. Furthermore, greater parallelism is possible.

2) The multiversion capabilities derive from the presence of a "differential file". This is where our design differs from that of LDM, another MVDB: LDB uses a "version pool" instead of a differential file.

All previous database systems suffer from a fundamental design flaw: a relation is stored in a single dynamic file. It is the need to keep the data in this file current that causes many problems:

1) It restricts the choice of file organizations which can be considered; a dynamic organization is required.

2) Since each inversion (to the relation) must be maintained, the number of inversions which can be used is limited.

3) Physical sequentiality of logically sequential pages is difficult to maintain. This can cause performance to suffer. This is true even if updates are done "in place".

4) Either transaction backout or transaction committal is slow: the current version has to be changed during these operations. In System R, backout is slow. Changes already incorporated must be backed out. This takes time and requires that locks be held until backout is complete. INGRES, on the other hand, achieves backout relatively easily: a "deferred update" file is discarded. But committal is slow [Ston76].

With a differential file based system, there is a large static

master file and a smaller dynamic file. Since the master file is static, problems (1) - (3) are avoided. They still affect the differential file but most of the work in processing a query involves the master file because it is much larger. Problem (4) is avoided since the differential file allows the co-existence of "before" and "after" versions of relations.

1. Differential files

The idea of a differential file is not new. It was proposed for use in a general purpose database system in [Seve76]. That paper also discusses some specialized applications in which the idea has already been used.

1.1. Outline of the system

The basic idea is to represent a relation using two files: a master file (MF) and a differential file (DF). The MF is a snapshot of the database at some time. The DF is a much smaller file which stores updates issued since the MF was built. We will assume that the MF is consistent: the MF represents a state in which all transactions have either completed or have not been started. (This is easy to guarantee.)

To process a query both the MF and the DF are searched. The results from the MF are corrected by the relevant updates from the DF.

Our design for the differential file system follows from the model of dynamic relations presented in chapter 1 section 4.

Recall that an update is either an insertion or a deletion. Each DF record will therefore contain a "status" flag to indicate whether the tuple is being inserted (status = present) or deleted (status = absent).

Since the same tuple can be updated several times, (e.g. it is inserted, deleted and re-inserted), the updates must be chronologically ordered. So a DF record also stores the timestamp of the transaction that generated it. The timestamp can be used to identify the transaction which generated the update since timestamps are unique.

To summarize, if a tuple of the relation is $(a_0, \ldots, a_{k-1})$ then the format of a DF record is $(a_0, \ldots, a_{k-1}, t, s)$, where t is the timestamp and s is the status. The meaning of such a record is that tuple $[a_0, \ldots, a_{k-1}]$ changed its status to s at time t.

The life cycle of the MF and DF is as follows, (see figure 1). Initially, the MF is up to date and the MF is empty. Updates are placed in the DF. All queries on the relation combine results from the MF and DF. When the DF reaches a certain size, (or after a certain period of time), the MF and DF are merged to create a new MF. The DF is then cleared and the cycle begins again. We will call this process "reorganization".

1.2. Advantages and disadvantages of the differential file

The advantages of using the differential file have been discussed in the introduction of this chapter. Lorie has raised a number of objections to the use of the DF [Lori77]. Each of these objections can be overcome.

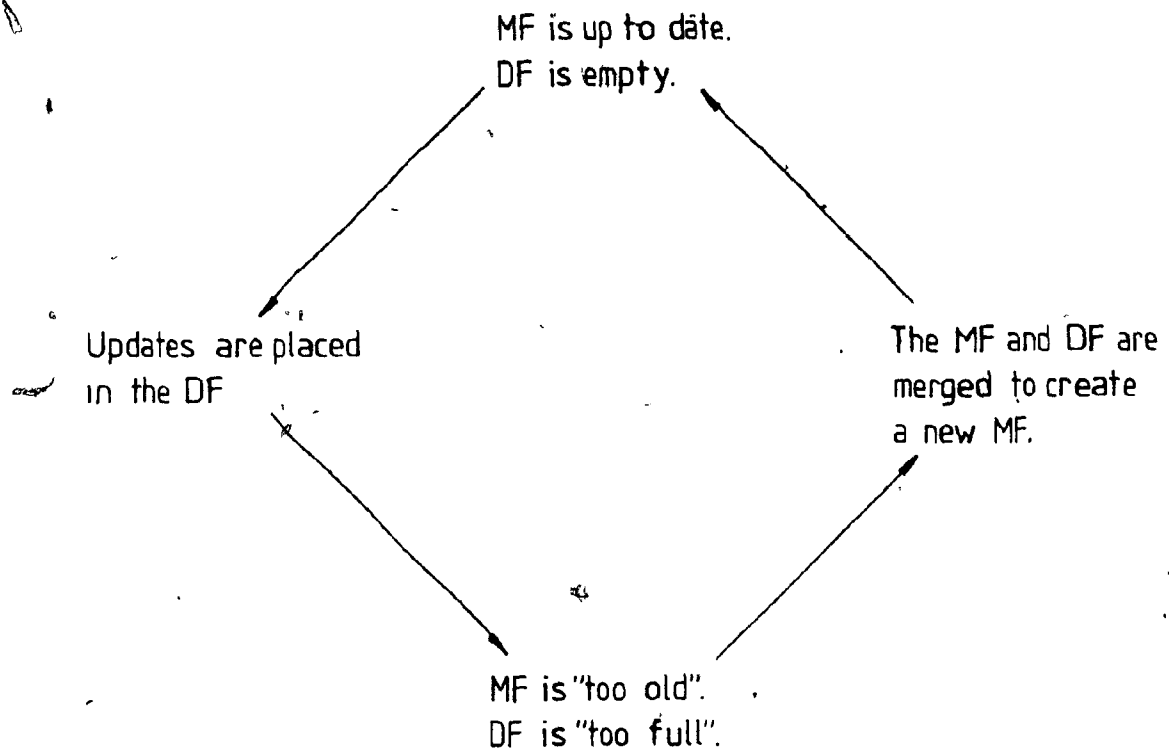


Figure 1. Life cycle of the master file (MF) and the differential file (DF).

1.2.1. Double accesses

One disadvantage of using an MF and a DF is the need to search both files. (It is puzzling that Lorie raises this point since his reference for differential files, [Seve76], discusses this point in detail.)

As discussed in [Seve76] this cost can be reduced by using a filter [Bloo70]. The filter is a device for answering the question "does the DF contain relevant updates?" with one of two answers: "no" or "possibly". The DF has to be searched only if the latter answer is received. Of course, consulting the filter should cost less than accessing the DF. It is therefore kept in primary memory at all times. We discuss the filter in more detail in chapter 6 section 2.3.

1.2.2. Cost of merging

Periodically, the MF and DF must be merged. This is a time consuming operation. There are two ways to lessen the impact of this operation:

1) The merge can be done by a concurrent process. That is, it is not necessary to shut down the system. A method for performing an "on-line" merge of this kind was briefly described in [Seve76], (see section 4.1.2). Another method will be given in section 4.1.3. The merge can be run during periods of low activity further reducing the effect of reorganization on the system's performance.

2) In order to recover from a hard crash, dumps must be taken periodically (whether or not a DF is used). If the dump is done at the same time as the merge, the apparent cost of the merge is lowered.

1.2.3. Multiple updates to a tuple

The DF must be able to deal with an update to a tuple already updated in the DF. In our view of updates this means that the tuple appears and disappears, (or disappears and re-appears). This will not be any problem in our proposed implementation of the DF.

2. A recovery system based on the DF

In this section we describe a recovery system for a PDB based on the differential file. The system will be able to recover from soft crashes and hard crashes that damage the MF, DF or both. We assume that the contents of primary memory are lost following either type of crash.

Soft crashes are more frequent than hard crashes. It is easier to recover from a soft crash: clearly, the soft crash recovery system is not concerned with the static MF; it must deal only with the DF and associated dynamic objects, (e.g. a list of active transactions).

Transaction processing can be modelled by the graph of figure 2. A transaction spends some time in each state along a path from Start to End. (The Backout state is entered when the user aborts the transaction.) A soft or hard crash may occur in any state. Transition from one state to another is instantaneous.

We will discuss the normal processing of each state, soft crash recovery and hard crash recovery.

There are seven dynamic objects which are modified in the processing of updates:

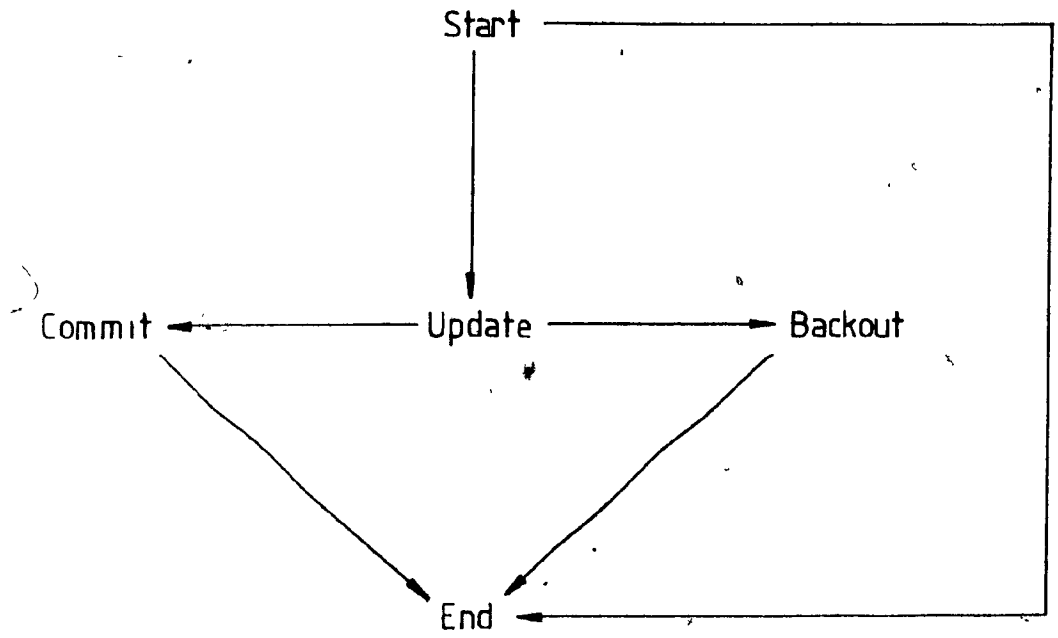


Figure 2. Model of transaction processing.

1) The DF.

2) The filter.

3) The page map: a table showing which disk pages are in use.

4) The progress report: A small disk file which records the state most recently entered by each active transaction.

5) The active list: a list of identifiers of all executing transactions, (those that have entered Start but not End). The starting time of each transaction is stored in this list.

6) The lock table: A list of locks held by each transaction, (required only if locking is used for concurrency control).

7) The update lists: Associated with each transaction is a list of pointers to the updates it has generated.

The DF, page map, progress report and update lists are stored on disk and updated in primary memory. The soft crash recovery mechanism operates correctly even when updates of these objects, written into a page buffer, have not reached the disk when a crash occurs. The active list and filter are in primary memory only. The lock table may or may not be small enough to fit in primary memory.

For hard crash recovery, a copy of the MF and a log are stored on tape.

2.1. Normal processing

The processing of the Start, Update, Commit, Backout and End states is described below. We assume that locking is used for concurrency control but the algorithms are easily modified for other concurrency control methods.

Writing into the log consists of sending a message to the tape log. This write goes to tape immediately; it does not wait in a buffer. Writing to the progress report (on disk) proceeds in the same way.

There is a problem concerning the timestamps of DF records due to the use of locking. If transactions are not assigned timestamps when they begin execution (as is the case with locking and optimistic concurrency control) then the timestamp is unknown when the update is placed in the DF. Following committal, the timestamps should correspond to the serialization enforced by locking. That is, if T_i appears to precede T_j because of locking, then the timestamp of T_i should be smaller than that of T_j.

Bernstein points out that the "lock point" of a transaction can be used as the timestamp (if locking is being used) [Bern81]. The lock point is a time after the last lock has been claimed and before the first unlock. (Recall that locking is two-phased; see chapter 4 section 2.1.) For recovery purposes, no locks are released until committal is complete, so a logical choice for the timestamp would be the time at which committal begins.

This suggests the following strategy: When transaction T_i places an update, use timestamp ∞_i, a very large timestamp unique to T_i, which means "this update from T_i has not been committed" (since the current time does not exceed ∞_i). Thus, concurrent transactions will ignore the update. During committal, ∞_i will be

replaced with TC_i, the time of committal of T_i, (see section 2.1.3). The processing of the various states is described below.

Our method does not have some overhead costs of other methods: e.g. the cost of creating checkpoints periodically.

2.1.1. Start

1) Write the Start message in the log and the progress report.

2) Enter the transaction's identifier in the active list and allocate an update list for the transaction.

A starting transaction must be allocated space in the progress report. The space used by a transaction which has reached End may be re-used, (see section 2.1.5).

2.1.2. Update

This state is entered when the first update is generated by the transaction.

1) Write the Update message in the progress report.

2) For each update:

2.1) Record the update in the DF, (assigning a timestamp of ∞, as discussed above).

2.2) Place a pointer to the update in the update list.

2.3) Write the update in the log.

2.1.3. Commit

1) Force to disk all pages in the buffers containing updates generated by the transaction.

2) Write the Commit message to the log and the progress report. The message includes TC, the time at which Commit was entered. This will be the timestamp of the transaction.

3) Traverse the update list of the transaction. For each update, set the corresponding bit in the filter and reset the timestamp to TC.

2.1.4. Backout

1) Write the Backout message to the log and the progress report.

2) Traverse the update list of the transaction, deleting each encountered update.

2.1.5. End

1) Write the End message to the log and the progress report.

2) Release all locks.

3) Remove the transaction from the active list and discard the update list.

The appearance of End in the progress report indicates that another transaction can use the space in the progress report occupied by the transaction that wrote the End.

2.2. Soft crash recovery

Soft crash recovery proceeds in two phases. First, the DF is scanned to restore data structures, finish committals and perform backouts. Then, interrupted transactions are restarted.

A scan of the DF is feasible since it is a small file. The scan could be avoided but then normal processing would be more expensive.

2.2.1. Phase 1

The filter, active list, page map, lock table and update lists were partially or totally resident in primary memory when the crash occurred.

These could all be updated securely (using careful replacement for example) but extra I/O would be generated. It is preferable to "penalize" the recovery system. That is, all of these data structures can be recovered from the progress report (which is reliable following a soft crash) and a scan of the DF.

DF records are handled in the following way during the scan:

An update from a transaction in Commit causes a filter bit to be set and the timestamp is reset to the TC of the transaction (available from the progress report).

An update from a transaction in Backout or Update is deleted.

All other DF records are from transactions that have ended. Each such record is used to set a bit in the filter.

When phase 1 is complete, all committing transactions interrupted by the crash will have finished committal and all transactions that were in Update or Backout have been backed out.

Below, we discuss the recovery of the dynamic components of the differential file system.

2.2.1.1. The active list

The active list can be reconstructed from the progress report: Transactions which are not in the End state are active. A transaction in the End state has finished step 1 of processing End (see section 2.1.5). The crash makes steps 2 and 3 unnecessary (in case they have not been executed yet).

2.2.1.2. The filter

The filter can be reconstructed during a scan of the DF. Any update from a transaction that has completed (i.e. reached End) or is in the Commit state sets a bit in the filter.

2.2.1.3. The update lists

These lists are no longer necessary (so they are not reconstructed). Any transaction in the Update state will be backed out and resubmitted: DF records from transactions in Update are deleted. Thus the backout will be achieved during the scan of the DF. Transactions in Backout will also be backed out during the scan. Committing transactions relied on the the update lists. However, committal is completed during the scan.

2.2.1.4. The page map

When transactions start running again, the page map will be needed. During the scan of the DF, all pages in use will be located; they are entered in the page map. All other pages are assumed to be free, (so pages that were allocated to update lists are automatically reclaimed).

Note that the page map is not needed during the scan since the update lists are not being reconstructed. By storing the page map on fixed pages, outside the page allocation scheme, the problem of finding pages for storing the page map is avoided.

2.2.1.5. The lock table

The lock table does not have to be reconstructed. During the DF scan, only committals and backouts are being processed and these processes cannot conflict. Resubmitted transactions must re-acquire their locks in the normal way but resubmittal does not begin until the DF scan is complete.

To summarize, phase 1 of soft crash recovery proceeds as follows: (transaction(u) is the transaction which generated update u. state(T) is the state that transaction T was in when the crash occurred.)

1) Reconstruct the active list from the progress report.
2) Scan the DF pages. For each page:
2.1) Enter the page in the page map.
2.2) For each update, u, on the page:

```
case state(transaction(u)) of
Start: (* cannot occur *)
Update: Remove the update.
Commit: Set the corresponding filter bit and reset
        timestamp to TC(transaction(u)).
Backout: Remove the update.
End: (* do nothing *)
default: (* the transaction ended before the crash *)
   Set the corresponding filter bit.
```

2.2.2. Phase 2

After phase 1 the page map has been reconstructed. All transactions that were in Update, Backout or Commit (in the

reconstructed active list) have been processed. In phase 2, End is written to the log and progress report for all active transactions. All that remains is to resubmit interrupted transactions that are still not complete; namely, those transaction that were in Update or Start when the crash occurred.

The execution of resubmitted transactions and new transactions can proceed concurrently.

Soft crash recovery is complete following phase 2.

2.2.3. Correctness of the soft crash recovery mechanism

We will now re-examine the algorithms of section 2.1. The possibility of a soft crash during each step of each algorithm will be considered and it will be shown that the soft crash recovery mechanism operates correctly in each case.

2.2.3.1. Start

A transaction in Start has not generated any updates and there is nothing to back out. Phase 2 of recovery ends and resubmits the transaction. It does not matter whether step 2 has been executed or not since the active list is reconstructed following the crash.

2.2.3.2. Update

If the crash occurred after step 1 then no updates have been generated. The transaction is ended and restarted in phase 2.

If the crash occurred during step 2 then all updates will be backed out in phase 1 of recovery.

Note that step 2.1 may involve the allocation of a new page for the DF. The allocation is reflected in the page map. It is

possible for a crash to occur after the allocation is noted in the page map but before the new page is linked into the DF. The page is not lost since it will be reclaimed in phase 1 of recovery. The update on the page is lost but would have been backed out anyway (since the transaction has not committed).

2.2.3.3. Commit

Since the progress report has the Commit message, all the updates generated by the transaction must have reached the disk, (due to the ordering of steps 1 and 2). Thus the DF contains all that is needed for committal, (which will occur during phase 1).

If step 3 is interrupted by a crash then committal is completed during phase 1 of recovery. (The filter is lost in the crash and is reconstructed during the scan.)

2.2.3.4. Backout

Any records missed due to the interruption of step 2 will be backed out during phase 1 of recovery.

2.2.3.5. End

Steps 2 and 3 are "achieved" by the crash itself.

2.2.4. Soft crash during recovery

If a crash occurs during recovery, the recovery algorithm is simply restarted. The DF and progress report are unchanged following the second crash and nothing else is needed for recovery.

2.3. Hard crash recovery

Any damaged dynamic object can be reconstructed using the log.

2.3.1. Rebuilding the MF

The tape copy of the (old) MF is copied to disk.

2.3.2. Rebuilding the DF

To rebuild the DF, the updates and backouts stored in the tape log are re-executed. The execution of each log entry proceeds as in section 2.1. Each update must be preceded by a writelock to ensure correctness. The readlocks held when the transactions originally executed are not necessary.

Update lists are maintained as the log is re-executed so that the filter can be set properly on transaction committal.

As for soft crash recovery, the page map is not available during hard crash recovery. However since the DF was damaged and is being discarded, the page map can be cleared and reconstructed during the reconstruction of the DF.

When the DF has been recovered, soft crash recovery can be started at phase 2.

2.3.3. Rebuilding the progress report

The progress report can also be rebuilt in a scan of the log. The states recorded in the log are written onto the progress report. When a change of state is encountered, it is reflected in the progress report in the usual way. By the time the log has been scanned, the progress report is up to date.

3. Concurrency control

The final major component of the PDB is the concurrency control system. System R and INGRES have considered these to be independent problems. In fact, concurrency control was an afterthought in INGRES [Ston76].

Recently, concurrency control has been seen as a problem related to recovery: both old and new versions of updated objects are kept for recovery purposes. The availability of both versions enhances parallelism [Baye80a, Stea81].

MVDBs have been shown to be of use in recovery and concurrency control [Chan82], (see also [Reed78] for a more general discussion). The old versions can be used for recovery purposes and to enhance parallelism. It is a generalization of the idea of using old and new values: all old values since a given time are maintained. The advantages of storing a history instead of just the most recent version are explained in section 3.2.2.

3.1. The role of the DF in concurrency control

Virtually every concurrency control method requires some information that is present in the DF. In this section, we will show how the DF can be used by several concurrency control methods. These techniques are applicable in a centralized system and at the sites of a distributed system.

3.1.1. Transaction backout

With the DF, transaction backout is very simple and does not require locking. Furthermore, backout and resubmittal can occur simultaneously. These are important considerations because most

concurrency control methods require some transactions to be backed out. Backout is slower and incompatible with resubmission in other systems.

3.1.2. The DF stores write sets

Some concurrency control methods depend on the write set of a transaction: the set of objects modified by the transaction. Given our view of updates, (see chapter 1 section 4), the write set of a transaction is the set of tuples whose status was changed by the transaction. These methods also require other information but the write set is already available in the DF: it is the set of DF entries pointed to by the transaction's update list.

3.1.2.1. Precision locking

Precision locking [Jord81] is a variant of predicate locking. Reads and writes are described by predicates which are stored in a lock table. The write predicates are restricted to be exact match queries. When a write is performed, it is checked against the lock table for conflicting reads, (the simplicity of the write predicate makes this test easy to perform). If no conflict occurs, the write predicate is entered in the lock table.

The lock table contains both read and write predicates. The write predicates are those DF entries pointed to from an update list. To test a read predicate for conflict, the updates on all update lists could be examined. It may be more efficient to process the query (corresponding to the read predicate) on the DF in the usual way, ignoring all committed updates. This yields the set of conflicting updates.

Testing a write predicate for conflict involves scanning the read predicates in the lock table. To expedite this process, it may be feasible to store the read predicates in a kd trie-like data structure. We have found such a data structure whose space requirements seem to be linear in the number of predicates stored (if the predicates are restricted to be range queries).

3.1.2.2. Optimistic concurrency control

The main difference between precision locking and optimistic concurrency control is that the latter postpones testing for conflict until just before committal. At this point, it is too late to lock so if a conflict is detected, one of the involved transactions must be backed out and re-executed.

The mechanisms of testing for conflict are the same as for precision locking.

3.1.3. The DF stores timestamps

A T/O method orders transactions by assigning timestamps to transactions in increasing order. Transactions with smaller timestamps have higher priority. Due to this ordering, deadlock cannot occur but transactions will be backed out if some sequence of operations does not respect the chosen ordering.

Using Bernstein's terminology [Bern81], T/O methods can be used for rw synchronization, ww synchronization or both. When used for ww synchronization, associated with each object, (tuple in our case), is the timestamp of the most recent writer of the object. When used for rw synchronization the timestamp of the most recent reader is also required. This is a very expensive

requirement. Let R-TS(x) be the timestamp of the youngest transaction that has read x. (Its timestamp is larger than that of all other readers of x.) Then every time a read of x takes place, R-TS(x) may have to be updated. This can generate a lot of I/O. Also, R-TS(x) must be kept for all tuples whether present or absent (in the current version). For these reasons, T/O methods should not be used for rw synchronization.

For ww synchronization, only the W-TSs are needed: the youngest writers of each tuple. The same objections regarding I/O and space requirements can be raised but, using the DF, these problems can be solved since only the W-TSs of tuples in the DF are needed:

The MF represents a "consistent" state. That is, it represents the state of a relation after one transaction has executed and before its successor (in the ordering guaranteed by serializability) has executed. In other words, no transaction that commits after the creation of the MF has a timestamp less than M-TS: the time at which the MF appears to have been created.

Now consider a write of x from a transaction T with timestamp t. According to the ww synchronization protocol, T must abort if t < W-TS(x). Since T is executing, t > M-TS. If t > W-TS(x) then t does not have to abort (whether or not W-TS(x) < M-TS). If t < W-TS(x), (T has to abort), then W-TS(x) > M-TS (since t > M-TS).

Thus, if T has to abort it is because of an update stored in the DF. If x has not been updated in the current DF then T does not have to abort. In other words, if W-TS(x) < M-TS then W-TS(x) is not needed.

In fact this synchronization technique does not use the DF to full advantage. This point is discussed in the following

section.

3.1.4. The DF is a multiversion database

3.1.4.1. Using old and new versions

Bayer et al. [Baye80a] have proposed a concurrency control method that makes use of the fact that, for recovery purposes, two versions of updated objects exist. Readers try to read the new value unless this would violate serializability; then the old value is read. A writer can create a new value for an object if only one version of it exists and no other writer is preparing a new vaue. There are a number of variations of the basic method; two of them are particularly interesting:

1) A read-only transaction never has to be backed out. When backout is required there is always at least one writer that can be backed out. (But it may be cheaper to back out a read-only transaction.)

2) If writers are serialized then no backouts will ever be necessary and all reads will obtain the after values. This strategy is only feasible in a centralized system.

This concurrency control method causes transactions to abort when deadlock occurs or when serializability is violated. Both occurrences are indicated by a cycle in a dependency graph. For this reason, the method may not be suitable in a distributed environment [Bern81]. (A distributed version of this method has been given [Baye80b]. Stearns and Rosenkrantz have also proposed a distributed concurrency control method that uses old and new versions [Stea81].)

All of these methods can use the DF as a source of old versions. But these methods can be improved upon since the DF stores all old versions since the creation of the MF.

3.1.4.2. Using multiple versions

An MVDB provides an environment suitable for the use of the concurrency control methods of Bayer et al. and Stearns and Rosenkrantz. These authors take advantage of old values present for reasons of recovery.

Other authors take a different approach: they note that multiple versions are useful in both recovery and concurrency control [Reed78, Chan82]. Even greater concurrency is possible using an MVDB.

The use of an MVDB makes ww synchronization unnecessary. Even if writes to the same object are received "out of order", the database will be in a correct state after processing them. Processing the older update after the younger one does not affect the current version; another old version is created. The insertion of this old version, however, may conflict with a reader. This would be detected by the rw synchronization technique.

3.2. Read-only transactions

In a static database, concurrency control would be unnecessary since any number of readers can share an object. In a dynamic database, concurrency control is necessary but, under certain conditions, the execution of read-only transactions, (i.e. transactions which do not place updates), can be expedited.

If updates are performed in place then nothing can be done for read-only transactions: writers cannot share with readers (or other writers). But if old and new versions of updated objects are available then a read-only transaction can avoid waiting or abortion if slightly out of date values are acceptable. Such values are often available as a by-product of a recovery system Also, multiversion databases are more general; they store all previous versions.

3.2.1. Using old and new versions

As noted in [Baye80a], the old version which is kept for recovery purposes can be given to a read-only transaction to avoid waiting. Permitting this read implies a certain ordering: R is read-only, W is a writer. Suppose that R reads the old version of x after W has prepared a new version. Then R precedes W in serialization. Subsequent actions must be consistent with this ordering.

If deadlock or an inconsistency is later discovered it is necessary to abort a transaction. It is shown in [Baye80a] that it is never necessary to abort and restart a read-only transaction (although it may be cheaper to do this than to restart some writer).

3.2.2. Using multiple versions

In a multiversion database, a read-only transaction can select any previous version to read or it can wait for an up to date version. In a system based on the differential file, any version since the creation of the MF can be read.

When a read-only transaction, R, is submitted, the time of the

version to be read, r, must be selected. r will determine the position of R in the serialization of all transactions. Clearly, if a transaction T ends before R begins then T precedes R. Now consider a writing transaction T which started before R and is active when R begins. It may commit after R begins, (all of its updates will have timestamp TC > r). These updates should be ignored by R. Finally, if T starts committal before R begins, (committal has not finished when R begins), then R cannot read any of T's updates since they do not all have the correct timestamp until T is finished committing. Thus r should be set just below the smallest TC of all transactions which are committing when R begins. LDM handles read-only transactions in this way.

This treatment of read-only transactions is more generous than that of Bayer et al. Since all past versions are stored, there is no constraint involving the writer of a new version and readers of the old version; there is no "pressure" for the old version to disappear due to replacement by the new version. In fact, read-only transactions can be completely ignored by the concurrency control system.

4. Operations on the physical database

In this section the implementations of basic PDB operations are considered. These are: reorganization, querying, the other operators of the relational algebra, update, backout and committal.

4.1. Reorganization

4.1.1. Basic algorithm

As more and more updates are stored in the DF, the performance of the system will deteriorate. This is because the DF will almost always have to be searched (as the filter becomes filled). Also, the DF is growing: searches are more expensive for larger files than for smaller ones. To prevent the performance from deteriorating too much, a new MF is created periodically and the DF is then cleared. The obvious method of doing this is to merge the two files. This requires that the MF and DF be ordered in the same way, i.e. they must be merge compatible. We will discuss merge compatability later. Assuming that the files are merge compatible and that updates to the same tuple are stored in reverse chronological order (i.e. the most recent update appears first), the algorithm given below performs the merge.

Reorganize
m and d are records from the MF and DF respectively. d.tuple is the DF record with the timestamp and status fields deleted. Read returns the next record in a file or ∞ if no more records are present. Next locates the next record in the DF which updates a different tuple; that is, all further (older) updates to the current d.tuple are skipped.

```
m := read(MF)
d := read(DF)
repeat until m = d = ∞

   case m < d:   (* m has not been deleted *)
      write(m)
      m := read(MF)

   case m = d:   (* m has been deleted. However, the most *)
                 (* recent update could be either an insertion *)
                 (* or a deletion. In any case, the current *)
                 (* update is the most recent and older updates *)
                 (* can be ignored. *)
      if d.status = present
      then write(m)  (* the tuple was re-inserted *)
      next(d)   (* skip over out of date updates *)

   case m > d:   (* A tuple has been inserted. As above, the *)
                 (* most recent update is the only one of *)
                 (* interest and could be an insertion or *)
                 (* a deletion. *)
      if d is an insertion then write(d.tuple)
      next(d)
end
clear(DF)
clear(filter)
release old MF pages
return

end Reorganize

read(file)
get next record in file
if end-of-file then return(∞) else return record
end
return

end read


next(d)
d' := d
repeat until d.tuple > d'.tuple
   d := read(DF)
end
return

end next
```

The system must be closed to users when Reorganize is executed. The only reason for this is that the DF must not change while the merge occurs. Otherwise, some updates that occur during reorganization would be included and some would not. Also, very recent updates (from uncommitted transactions) may be intermediate or they may be backed out.

4.1.2. On-line reorganization

To avoid shutting down the system while reorganization occurs, the use of the "differential differential file" (DDF) has been proposed [Seve76]. (The DDF would have its own filter.) When reorganization is about to begin, the DF is closed (i.e. it will not receive more updates). Subsequent updates are placed in the DDF. Searches must now refer to as many as three files: the MF, the DF and the DDF.

Reorganization begins when the DF is closed and proceeds as in section 4.1.1 (except that the system is not shut down). When the reorganization is finished, the new MF replaces the old one and the DDF replaces the DF.

There are some subtleties involved in using the DDF. These concern transactions that are executing at C, the time at which reorganization starts. Clearly, transactions that end before or start after C are not a problem. Updates from these transactions are all in the DF or all in the DDF respectively. However, a transaction that starts before and ends after C must be treated carefully:

Consider a transaction that has not reached Commit at time C, (suppose it is in Update). Updates from this transaction that are in the DF will not be included in the new MF. When the DF is

discarded, the updates will be lost. Two possible solutions are:

1) Abort the transaction and restart it so that all updates will appear in the DDF.

2) During Reorganize, move such updates into the DDF as they are encountered during the merge.

Some transactions may be in the Commit state at time C. The timestamps of these transactions are less than C. If reorganization starts as scheduled then some updates that have not yet been committed may be missed. An easy solution is to wait until all committing transactions end and then start reorganization. Of course, more transactions may enter Commit during the wait. This is not a problem since their timestamps exceed C. Their updates will be included in the next merge.

4.1.3. Another on-line reorganization method

The on-line reorganization method described penalizes query processing since as many as three files may be searched. On-line reorganization can also be accomplished without using the DDF.

The DDF allows the DF to remain fixed even though updates are being processed. Suppose that reorganization is initiated at time C. Instead of putting updates in the DDF, they are still placed in the DF (i.e. the DF is not closed). The reorganizing algorithm is modified to ignore updates whose timestamps exceed C. After the merge is complete, the old DF records, those with timestamps not exceeding C, are deleted from the DF, (for purposes of recovery they are not deleted during the merge). Then, the DF contains only new updates.

This algorithm has a number of problems:

- The reorganizing process deletes records from the DF. These

updates must not interfere with other users of the DF. (Short term locks would be placed by Reorganize.)

- Every record inserted into the DF will eventually be deleted (during reorganization). The data structure representing the DF must be able to cope with all these deletions efficiently and without degeneration.
- Two passes of the DF are required since deletion of a DF record cannot immediately follow its inclusion in the merge: Suppose a crash occurs just after the deletion appears in a buffer. Since the new MF record had not reached disk when the crash occurred, the update is lost.

The DDF avoids all of these problems. In practice, using the DDF would not seriously affect performance. It is very sparsely filled and is only required during reorganization. We therefore recommend the use of the DDF.

4.1.4. When to reorganize

The process which reorganizes does not have exclusive control of the system, (if, for example, the DDF mechanism is used). It can be run on a time-sharing basis during periods of low activity.

The question of how often to reorganize, (i.e. the time between initiations of the process), is open. The problem of when to reorganize (or take a checkpoint etc.) has been considered but it was always assumed that reorganization required shutting down the system to users.

Under this assumption (and several others) Shneiderman shows that the optimal time between reorganizations is

$$t = \sqrt{2R / \theta}$$

where R is the cost of reorganization and θ is the rate of deterioration of the cost of searching [Shne73]. Lohman and Muckstadt obtained a similar but more general result in [Lohm77].

In some sense, there really is not much of a problem here: reorganization is not a serious burden since it is performed during periods of low activity. As soon as one reorganization is complete, the next one could be started.

To formulate an optimization problem would require an understanding of how the performance of the system is affected by the reorganization process.

4.1.5. Recovery of the reorganization process

Since reorganization is a lengthy process, a crash should not cause the reorganization to abort and restart. Instead, it should be able to resume with little or no loss of progress.

The essential information is L, the value of the last tuple written to the new MF. The merge can be resumed after locating the smallest value greater than L in the old MF and in the DF.

L can be located easily: it is the largest value stored in the new MF (according to the order used for merging).

4.2. Querying

In the context of a DF based PDB, querying involves these four steps:

1) Search the filter.

2) Search the DF corresponding to positive responses from the filter.

3) Search the MF.

4) Combine the results of (2) and (3).

The details of the algorithm depend on the design of the MF, DF and filter. We therefore postpone giving more detail until the design of these components have been discussed. The algorithm is described in chapter 6 section 2.4.

4.3. Other relational algebra operators

In this section we consider the implementation of more complex operators: set operators, project, θ-join, $\mathcal{U}$-join and $\mathcal{T}$-join. These may be executed directly on the base relation represented by the MF and DF but it is more likely that these operators would be applied after selection. Moving selection "down" the parse tree of the query has been suggested to reduce the size of intermediate results [Smit75, Hall76]. The selection may be quite selective and produce output which is much smaller than the relation which was searched. There are two consequences of this point:

1) This is a very important optimization in practice.

2) As far as performance is concerned, selection is the most important operator since it will usually have to deal with the largest volume of data. The operations carried out following selection will be expedited since they will receive smaller operands than they would have otherwise.

This strategy has been used by optimizers in System R [Seli79], INGRES [Ston76] and PRTV [Todd76].

We will discuss the implementation of operators in both contexts, (i.e. executed on the base relation directly and following other operators). When an operand is a base relation the algorithm implementing the operator must see the current

version of the relation. This will be accomplished by performing a merge (as in the Reorganize algorithm) passing the results to the algorithm via a pipe.

4.3.1. Set operators

These operators are, in most cases, most efficiently implemented using a merge. Sorting may be necessary to ensure merge compatability.

Z ordering is useful here. Z ordered base relations are merge compatible. Even if selection is first applied to the base relations, z ordering is preserved. The Rangesearch algorithm returns tuples in z order. Of course, relations clustered on some access set also have this property as long as selection preserves this order.

4.3.2. Projection

The usual method of implementing projection is to project the tuples and sort them. One more pass is used to eliminate duplicates. This method applies to base relations, (tuples piped from the merge would be projected), and to results returned from other operators.

Clustering on some access set is more useful than (low bias) z ordering for projection: projection on a prefix of the access set can be done in one pass.

4.3.3. Θ-join

It has been suggested by simulation studies that using one of two methods for computing the natural join following selection is

nearly always optimal [Blas77]:

1) This method assumes that inversions on the join attribute exist for both operands. The inversions are merged to find matching index values. When a match is found, a tuple from one operand is retrieved and the predicate is applied to it. If the tuple satisfies the predicate then matching tuples from the other relation (which satisfy the predicate) are retrieved. Tuples of the join are then constructed.

2) This method does not require any inversions. Each operand is sorted on the join attribute. During the sort, tuples not satisfying the predicate can be discarded. The join is carried out during a merge of the sorted tuples.

Method (1) is not always applicable. Method (2) can always be used but (1) is usually preferable.

For other joins (e.g. <-join) the inversions of (1) are not as useful as for the natural join. Also, (1) cannot be used if the join follows operators other than selection, (since inversions would be unavailable).

If ZMDSs are used then another algorithm becomes feasible: Suppose the join is $R[X \ \theta \ Y]S$ where $\theta \in \{=, \neq, <, \leq, >, \geq\}$. Perform a selection on R using the predicate. For each tuple returned, r, perform a selection on S, appending to the predicate the condition $r[X] \ \theta \ Y$. Then combine r with the tuples retrieved from S to create the tuples of the join.

This algorithm has not been analyzed as those of [Blas77]. It is essentially the "tuple substitution" method used in INGRES [Ston76].

4.3.4. μ-join

A μ-join can be implemented using a merge. The details depend on which μ-join is being performed. For example, consider $R[Y_R \mu Y_S]S$ (as in section 1 section 2.1.4). If $r[Y_R] \not\subseteq S[Y_S]$ then the $\cap$-join would not create a tuple but the $\cup$-join would.

Z ordering does not seem to be useful for the implementation of μ-joins.

4.3.5. σ-join

Consider $R(X, Y)[Y \sigma Y]S(Y, Z)$. To implement this operation efficiently, S should be sorted on Y and R should be sorted on (X, Y). Then a merge yields the X values of the result. The relations may already be sorted as indicated, (but they definitely will not be if ZMDSs are used).

Other algorithms which make use of inversions and ZMDSs could be used. We are not aware of a study comparing the costs of these methods.

4.4. Updating, transaction backout and transaction committal

The process of updating the DF has already been described briefly. Here we consider the process in more detail and the related issues of transaction backout and transaction committal.

4.4.1. Updating

The basic idea is to place in the DF a record describing the update. $(a_0, \ldots, a_{k-1}, t, s)$ indicates that tuple $(a_0, \ldots, a_{k-1})$ takes on status s at time t, the timestamp of the transaction generating the update.

Two successive updates to a tuple should have opposite stati, (e.g. it does not make sense to insert a tuple that is already present). In other words, whenever a tuple is updated, its status changes.

Consider a transaction that updates the same tuple more than once, (e.g. a tuple is inserted, deleted and re-inserted). If each update of the same tuple generated a new DF record then the DF will have several records with the same values for (a_0, ... , a_{k-1}, t). They will differ only in status. Thus it will not be known which status is correct (i.e. most recent). To correct this, subsequent updates could operate by just negating the status flag. But then it may appear as if there are two successive insertions or deletions. The correct procedure is to check for the existence of the DF record identified by (a_0, ... , a_{k-1}, t). If none is found, the update is processed by inserting the record. If the record is found then it is deleted. (Intuitively, the transaction has "changed its mind").

Each transaction should maintain a list of pointers to DF entries it has generated (to facilitate backout; see section 2). If a DF entry is deleted as described above, the list of pointers must be updated to reflect the change.

4.4.2. Transaction backout

Transactions may be backed out for a variety of reasons. When deadlock occurs (in a locking scheme) one transaction must be aborted. In other concurrency control methods backout is also required (e.g. a transaction which fails validation in optimistic concurrency control [Kung81]). A transaction interrupted by a system failure has to be backed out. Finally, the transaction may

be cancelled (if it has not been committed) by the user who initiated it.

Recall that each transaction keeps a list of pointers to the DF entries it has created. To back out a transaction, the indicated DF records are merely deleted. This does not require any locking at the logical level: the DF entry was not visible to other transactions. (But physical locking of DF pages being updated is required during the actual deletion of the record. See chapter 6 section 2.2.2.)

This method is simpler than that of System R [Astr76] which has to run the log "backwards" on the current version of the relation. Logical locking is required. The method used in INGRES is simpler than ours: a file containing the updates of the transaction is discarded [Ston76].

4.4.3. Transaction committal

The existence of a list of active transactions has been assumed, (see section 2). A transaction not on this list is assumed to be committed.

To commit a transaction, each DF record generated by the transaction is used to set the corresponding filter bit, the time of committal is placed in each such record and the transaction is removed from the active list. Updating the active list must be done last. Otherwise, another transaction may not find all of the updates generated by the committing transaction.

A summary of this chapter can be found in chapter 8 section 4.

Part Three

Integration

Chapter 6

Detailed Design of the Physical Database

In sections 1 and 2, three problems in the design of the PDB were considered:

1) Data structures for storing and searching relations.

2) Recovery from soft and hard crashes.

3) Concurrency control.

We proposed a design for a multiversion database based on the differential file. In this chapter, the implementation of the master file, differential file and filter are considered. The results of chapter 3 will be very useful.

1. Design flaws in existing systems

The designers of System R and INGRES have written retrospections of their systems [Cham81, Ston80]. These observations and others will help in the design of new PDBs.

1.1. Lessons learned from System R

An early version of System R was built on top of a relational memory system called XRM (extended relational memory) [Lori74]. XRM stores tuples consisting of pointers to attribute values stored in representations of domains. The time needed to construct the actual tuples makes this organization undesirable, (although it saves storage space). The pages of System R segments store values instead of pointers to values since efficiency is of greater concern than space requirements.

Another lesson learned from the preliminary version was that tuple identifiers (TIDs) returned from searches of inversions,

are expensive to manipulate. Thus the System R optimizer will never consider the use of more than one inversion [Astr76, Seli79]. (The most selective inversion will be searched. Tuples retrieved will then be tested individually to see if they satisfy the restrictions on other attributes.) We regard this as evidence that MDSs are more suitable for processing anything other than very simple queries which do not require the manipulation of TIDs.

In the retrospection, it was emphasized that clustering is an important property. If inversions are used then clustering is possible on one access set only, (recall that inversions are highly biased). This further supports the use of low bias MDSs which cluster on several attributes simultaneously.

We agree with the designers of INGRES who did not implement links [Ston76]. Their inclusion complicates the PDB since they cause the space required by tuples to vary dynamically. If, (as in most systems), tuples are fixed in width, then only the inclusion of links causes the space requirement of a tuple to change. Links also have maintenance costs which limit their use.

Most importantly, clustering on a binary link is of questionable value: the tuples of two different relations (in the same segment) may be stored on one page due to a clustered binary link. It is claimed that this expedites the processing of certain joins [Astr76]. But a join is usually preceded by selection on one or both operands. This selection reduces the sizes of the operands and speeds up execution of the join. The selection is often highly selective, (see chapter 5 section 4.3). When discussing joins, then, it is important to consider the effect of clustered binary links on the performance of selection: the

performance is worse than would be the case if each page stored tuples from one relation only.

1.2. Lessons learned from INGRES

INGRES makes use of the UNIX I/O facilities. Much better performance could have been obtained by a customized facility, (for example, physical clustering would have been possible). This alternative approach was taken by the designers of System R.

INGRES stores some relations in an "ISAM-like" file [Ston76]. ISAM [IBM66] is a static organization. The designers of INGRES regret not having used a dynamic organization, (e.g. the Btree). It avoids the need for periodic reorganization (which is scheduled by the database administrator) and degradation of performance due to the existence of overflow records.

2. Data structures for the physical database

In this section we consider the problem of selecting data structures for the major components of the PDB: the MF, the DF and the filter.

Our primary concern is to minimize the amount of I/O generated in processing transactions. Storage utilization is a secondary consideration. These priorities are justified by the rapidly falling cost of secondary storage.

2.1. The MF

The data structure selected to represent the MF should support the efficient evaluation of range queries. Another important consideration is that the MF should be built as tuples are supplied (in a certain order) from the merge of the old MF and the DF. Hopefully, it will not be necessary to store the tuples in a temporary file for preprocessing.

2.1.1. Use an MDS

The two main contenders for the MF are:

1) An indexed-sequential file, clustered on one access set with inversions on one or more access sets.

2) An MDS.

As noted in chapter 2, inversions are suitable for processing simple queries but not complex queries. (System R never uses more than one inversion to process a query. See section 1.1.)

MDSs, on the other hand, process complex queries efficiently but do not perform as well as inversions for simple queries.

These two choices are not mutually exclusive. An MDS can be used. This will dictate a certain clustering which will not correspond to clustering on any access set (unless a high bias MDS is used). Then, inversions on selected access sets can be constructed. Simple queries will use one of the inversions if possible. Other queries will use the searching capabilities of the MDS.

The advantages of using an MDS outweigh the loss of physical clustering on some access set (which is incompatible with the use of a low bias MDS). Physical clustering permits efficient sequential processing on one attribute but not on others. An MDS

provides some clustering on all attributes simultaneously and also supports the efficient processing of range and partial match queries.

Note that any number of inversions can be set up for the MF and there is no maintenance involved. Since the MF is static, so are the associated inversions. The only penalty paid for the use of an inversion is the cost of setting it up, but this is done during reorganization which does not shut down the system. (And we are not concerned with storage costs.)

2.1.2. Which MDS?

Now the problem is to select an MDS. An MDS requiring a sort of the old MF to ensure merge compatability with the DF should not be used: the MF may be quite large. For this reason, the kd tree, the K-D-B tree, multidimensional clustering and multipaging are ruled out. These MDSs do not order the tuples in any consistent way. For example, an inorder traversal of the kd tree does not yield the tuples in a sequence determined by the tuples themselves. The order of insertions and deletions also plays a role in determining the sequence, (see chapter 3 section 2.1).

All of the other MDSs described in chapter 2 are ZMDSs. We reject the use of high bias ZMDSs. They offer no advantages over low bias ZMDSs which can be made more biased if so desired.

An important property of the ZMDSs is that they store tuples in a specific order: z order. Thus, if the MF and DF are not merge compatible, only the DF has to be sorted.

Some candidates for the MF data structure are discussed in the next section.

2.1.3. Practical ZMDSs for the MF

2.1.3.1. EXCELL

EXCELL has severe worst case problems. They can be alleviated but this would require some preprocessing, so EXCELL cannot be built with tuples piped directly from the merge.

2.1.3.2. HCELL

HCELL partitions the space into a grid as does EXCELL but allows overflow to occur, (thus HCELL may use a coarser grid). When a cell overflows, it is dealt with by imposing another grid on it and using another EXCELL-type structure. Thus a tree is set up; in most cases a very shallow one. (The kd trie is a special case of HCELL: A region of the space is split into two sub-regions.)

For any cell which does not overflow, a random access costs usually one but no more than two disk accesses: one access is sometimes required to read the HCELL directory page and another to retrieve the page representing the cell. Most random accesses will require one disk access since the directory is much smaller than the set of pages storing the tuples and since the Rangesearch algorithm would never require a directory page to be read more than once. (Also, a sequential access never references the directory if the pages are linked.) Note that the pages representing the cells can be allocated sequentially on disk. This allows for efficient processing of sequential accesses across cell boundaries (which would be generated by the Rangesearch algorithm).

If a cell overflows then the corresponding primary page stores

all or part of another HCELL directory. In either case, the pages of this second level can also be allocated sequentially. These pages interrupt the sequence of the pages of the first level of HCELL (as they should to maintain z order).

HCELL avoids the worst case of EXCELL since overflow affects individual cells, not the entire directory. However, a worst case resembling that of the kd trie does exist. It is possible to have a deep HCELL resulting in low storage utilization and slower random access.

2.1.3.3. The zkd Btree

The zkd Btree cannot degenerate as HCELL can but it gives poorer performance than HCELL for processing random accesses, (if the HCELL is not degenerate).

A zkd Btree node has a smaller branching factor than an HCELL node: An HCELL directory entry consists of a pointer to a page. A zkd Btree entry must, in addition, store a discriminator (possibly a complete tuple). The discriminator occupies several bytes. Thus a zkd Btree is usually less "bushy" than HCELL and is deeper resulting in slower random accessing.

2.1.3.4. zkd MLOPLH

Zkd MLOPLH is the MLOPLH data structure (see chapter 3 section 3) transformed into a ZMDS (as in chapter 3 section 2). This data structure combines the best features of HCELL and the zkd Btree. Random accessing is at least as fast as for HCELL (since a directory is not needed), and definitely faster than for the zkd Btree. The cost of sequential accessing is comparable to the cost for a zkd Btree.

The performance of zkd MLOPLH can be improved due to some characteristics of the MF and the way in which it is built.

- Tuples are piped from the merge in a specific order.
- After the merge is finished, the MF is static. This allows the creation of a zkd MLOPLH without any sparse buckets since the cost of update operations is not a concern. (Recall that there is a tradeoff involving the worst case times of updating and sequential accessing. See chapter 3 section 3.5.2.1.)
- The size of the new MF is known in advance. Therefore, the zkd MLOPLH should not be grown from a single bucket. In this way a lot of splits are avoided.

Based on these observations, the zkd MLOPLH can be grown as follows:

- Allocate enough primary pages to store the entire file without overflow (assuming a perfectly smooth distribution).
- Place records in buckets according to the prefix of the shuffle value. Note that all insertions to a bucket will occur consecutively.
- When both buckets of a pair of brother buckets have been created, join them if at least one of them is sparse, (and keep doing this recursively until the sparseness is eliminated).

The algorithm MFLoad, given below, creates the zkd MLOPLH.

This data structure appears to be the most suitable one for representing the MF. Random accessing costs one disk access if the accessed bucket did not overflow. Otherwise, the cost is no worse than for a zkd Btree. Sequential accessing is as fast as for a zkd Btree since no sparse pages exist.

MFLoad()
The file is initialized with 2^m empty buckets. b is the number of the bucket being filled. The coroutine GetNextTuple supplies t, a tuple from Reorganize, and sends a value of ∞ when Reorganize is finished. Define $h_m(\infty) = 2^m+1$. CloseBucket is responsible for eliminating sparse buckets.

```
b := 0
GetNextTuple(t)
while t < ∞
   if h_m(t) > b
   then   (* t is the first tuple of a new bucket *)
      for i := b to h_m(t)-1
         CloseBucket(i,m)
      end
      b := h_m(t)
   end
   AddToBucket(t,b)
   GetNextTuple(t)
end
return

end MFLoad
```

CloseBucket(b,m)
If b is the "right" brother then both brothers have been set up. Combine them if either is sparse (and call CloseBucket recursively for the resulting bucket).

```
level(b) := m
b' := b - 2^(m-1)
if b' ≥ 0
then   (* b' is the left brother, b is the right brother. *)
   if sparse(b) or sparse(b')
   then
      JoinBuckets(b',b)   (* See chapter 3 section 3.7.3 *)
      CloseBucket(b',m-1)
   end
else   (* b is the left brother *)
end
return

end CloseBucket
```

2.2. The DF

The requirements of the DF are quite different from those of the MF. The DF is highly dynamic: there are many insertions and some deletions. The DF is also used in query processing and in the merge with the MF.

2.2.1. Use a zkd Btree or zkd MLOPLH

The use of many inversions is not feasible since each would have a maintenance cost, and the DF is highly dynamic. (Also, the presence of inversions would complicate the recovery system.) So all accessing will have to be done through an MDS. There are several advantages obtained by using a DF which is merge compatible with the MF:

- The DF does not have to be sorted before reorganization.
- The processing of range queries will be simplified. It will also be faster since sorting will not be necessary to ensure merge compatability.

We therefore restrict our attention to ZMDSs.

A DF record has two more fields than an MF record: timestamp and status (which we will refer to as t and s respectively). The bits of these values must be included during shuffling, (a ZMDS is being used). Clearly, to maintain merge compatability with the MF, all of these bits should appear after the bits of the attribute values. Also, it will be useful to have all of the updates to a tuple stored in reverse chronological order. So after the bits of the attribute values, the complemented bits of the timestamp are appended. Finally, the status bit is appended. In other words, the shuffle function for the DF is

$$\text{shuffle}_{DF}([a_0, \ldots, a_{k-1}, t, s]) =$$

<shuffle([a_0, ... , a_{k-1}]) | complement(t) | s>

Since the DF is dynamic, static organizations are rejected. HCELL has some problems with storage utilization and therefore, sequential accessing, (since it is not multi-level in the same way as zkd MLOPLH).

We therefore recommend the use of a zkd Btree or zkd MLOPLH for the DF. We cannot recommend the use of zkd MLOPLH without reservation. Due to the static nature of the MF, certain improvements to zkd MLOPLH were possible, (e.g. the elimination of sparse pages). But with the dynamic DF, these improvements are not possible.

2.2.2. Concurrent operations on the DF

Two transactions which do not conflict logically may conflict physically. Suppose that transaction T locks the predicate $X \leq 10$ and that transaction T′ locks the predicate $X > 10$ (where X is an attribute of the relation). If T writes a tuple with $X \leq 10$ and T′ writes a tuple with $X > 10$ then there is no logical conflict. But if both updates are directed to the same page of the DF there is a physical conflict.

This is a much simpler concurrency control problem than the problem of guaranteeing serializability. For example, if zkd MLOPLH is used, then a very simple locking scheme is possible: a transaction merely locks the bucket being updated for the duration of the update.

The problem of concurrent Btree operations has been studied. A slightly modified Btree requires a writer to lock a small number of pages [Lehm81]. This method does not handle deletions well, but the DF does not have to process many of them; the entry

can be flagged as deleted.

Optimistic concurrency control can also be used for both data structures. Instead of locking out a conflicting writer, the younger transaction involved in the conflict is forced to re-execute. This method requires that each writer make a private copy of the pages or buckets being updated.

2.3. The filter

The function of the filter is to indicate when a DF may have an update relevant to a query [Seve76]. We will design a filter for the PDB. Analysis of the filter will show that it is virtually worthless in the context of a relational database. We then discuss filters that are helpful in this context.

2.3.1. The original proposal

The filter for a relational database, as originally proposed, is an array of M bits [Seve76]. Initially all the bits are set to 0. When a record with key k is updated the hash values $h_1 = f_1(k)$, ... , $h_X = f_X(k)$ are computed. (Each f_j returns an integer in [0, M-1].) Bits h_1, ... , h_X are set to 1.

To use the filter, the hash functions are applied to the search argument. If all of the accessed bits have values of 1 then it is possible (but not certain) that the DF contains a relevant update of the record corresponding to the search argument. If any of the bits are 0 then the DF does not contain a relevant update.

The parameters M and X control the accuracy of the filter. Obviously, M should be as large as is feasible, (given that the

filter should be kept in primary memory). Given M, X can be selected to optimize performance according to a variety of criteria (see [Seve76]).

The filter as described is not suitable for our purposes. It can handle match queries on 1d data. In the next section we discuss a more general filter.

2.3.2. A filter for a relational database

The filter described above is not satisfactory for two reasons:

1) The queries to be processed are range queries, not just match queries.

2) The data is multidimensional.

The first point suggests that the hash function(s) should be order preserving. Without any knowledge of the distribution of the points (corresponding to the updates) the only logical choice for the hash function is

$$h(k) = \lfloor k / s \rfloor$$

where s is some scaling factor.

The second point is easily dealt with: multidimensional data can be treated as one dimensional following shuffling.

The filter, then, is designed as follows: tuples consist of d bits; shuffling yields an integer in $[0, 2^d-1]$. The filter contains M bits and, in practice, $M << 2^d$. The hash function to be used is

$$h(t) = \lfloor (M / 2^d)\ \text{shuffle}(t) \rfloor$$

where t is the tuple being updated. Each bit in the filter represents $2^d/M$ bits of the space consecutive in z order. Thus each bit corresponds to one "cell" of the space.

2.3.3. The filter is useless

It can be argued that the filter proposed is almost useless in a relational database. Suppose that the filter has M bits and that n updates have been placed in the DF. Then, assuming that the updates are uniformly distributed, the probability that a given update falls in the cell of a given filter bit is 1/M. The probability of a miss is 1 - 1/M. The probability of all n updates missing the cell is $(1 - 1/M)^n$. So the probability that at least one of the n updates falls in the cell is $p = 1 - (1 - 1/M)^n$. This is the probability that a given filter bit is on. A partial match query on t of the k attributes covers $M^{1-t/k}$ of the filter's bits.

Now, the probability that all of these bits are off is

$$p_o = (1-p)^{M^{1-t/k}}$$

p_0 is usually very close to zero.

$$\log_e(p_o) = M^{1-t/k} n \log_e(1-1/M)$$

$$\simeq M^{1-t/k} n(-1/M)$$

$$= -n/M^{t/k}$$

$$p_o \simeq e^{-n/M^{t/k}}$$

p_0 is strongly dependent on t/k. Figure 1 plots p_0 vs. t/k for $n = 2^{10}$ and $M = 2^{16}$. The graph shows that the filter is virtually certain to refer a query to the DF unless over half of the attributes are specified (in a partial match query). The results hold for a wide range of M and n.

The use of the filter in the original proposal [Seve76] was

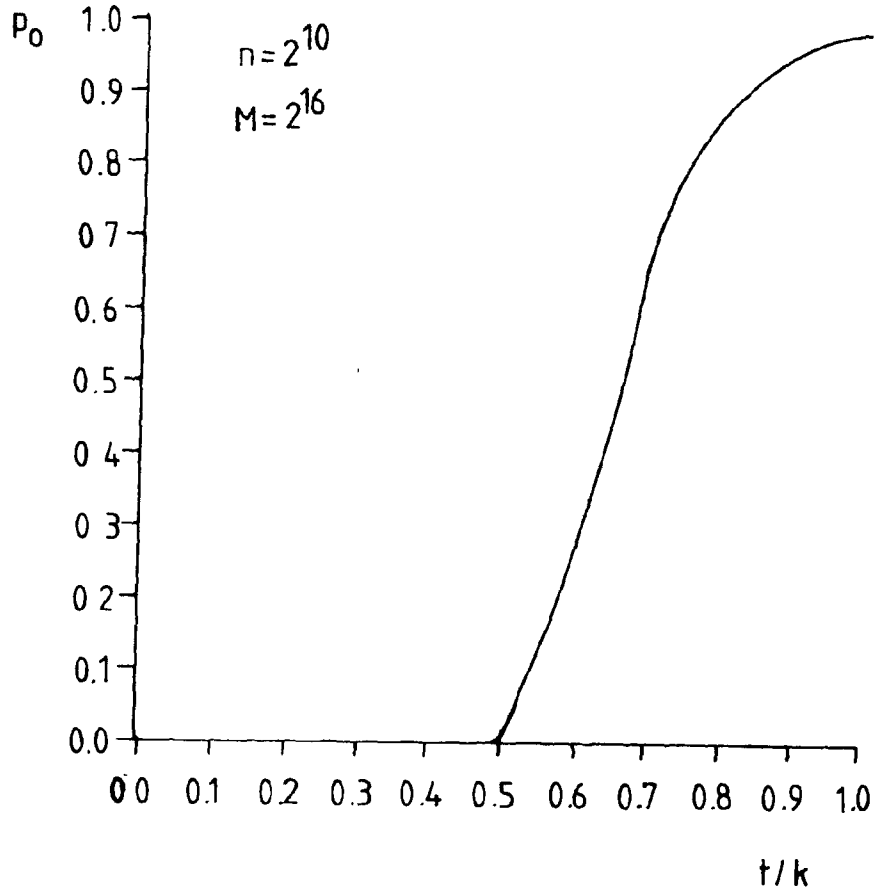


Figure 1. p_o vs. t/k

justified: t = k = 1 and t/k = 1. But for multidimensional data, the benefit of the filter is not so clear.

Now consider how many of the accessed filter bits are on. A partial match query covers $M^{1-t/k}$ filter bits. Each has probability

$$p = 1 - (1-1/M)^n$$

$$1 - e^{-n/M}$$

of being on. The expected number of filter bits that are on is approximately

$$N = M^{1-t/k}(1-e^{-n/M})$$

Figure 2 shows how N grows with n.

To summarize these results, it is very unlikely that the DF will not be referred to.

Now recall how the DF is accessed. The Rangesearch algorithm examines a search region, SR, by generating a random access and one or more sequential accesses. The random access uses the filter, and, as shown, is almost certain to be referred to the DF. From this point on, the filter can be ignored: the filter will indicate which regions of the space (inside the SR) have been updated but this is helpful only if a random access is being processed.

The filter, as designed, is useless.

2.3.4. Multiple filters

Recall from section 2.1 that the MF is represented by an MDS augmented by inversions. Consider a simple query: X = x, and suppose that an inversion on X exists. From section 2.3.3 it is almost certain that the DF will have to be accessed, (t = 1). Since the MF is accessed quickly using an inversion, it is

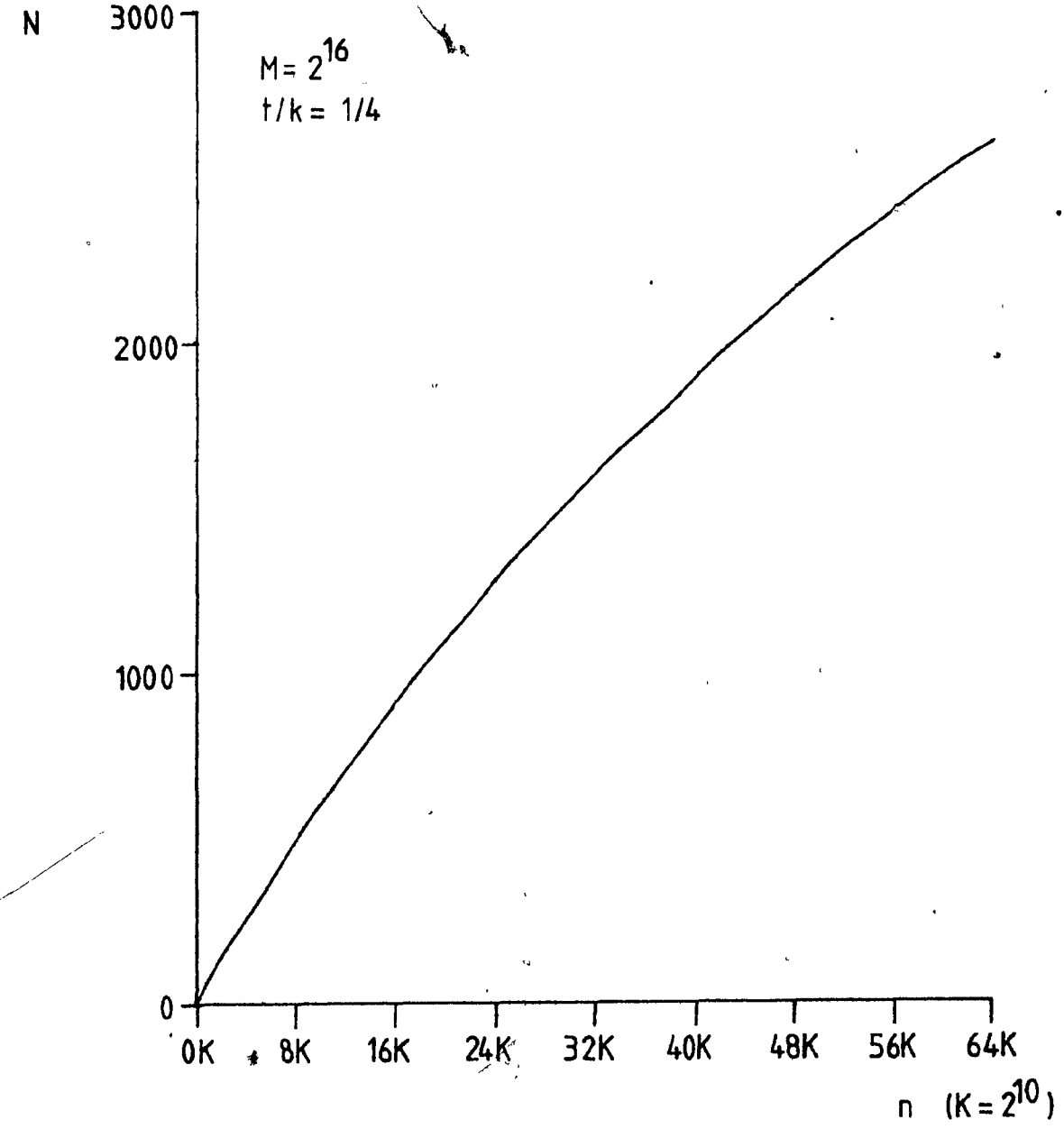


Figure 2. N vs. n

possible that it will be more expensive to access the DF (which does not have an inversion on X) than the MF.

But consider another filter, designed to deal exclusively with accesses on X. Then k = 1 and for the above query, t/k = 1. From the results of section 2.3.3 it can be seen that this filter would be valuable in processing queries on X.

We therefore propose the following: for each inversion, I, set up for the MF, a filter should be set up for dealing with queries on the access set of I. These filters should be kept in primary memory. The modification of the algorithms given in this chapter to deal with multiple filters is straightforward.

2.4. Searching the MF and DF

Now that the design of the MF, DF and filter have been specified, the algorithm for processing a range query can be specified. The outline of chapter 5 section 4.2 can now be simplified since the (multidimensional) filter has been eliminated. All that is required to process each SR is a simple merge of the MF and DF. The basic algorithm and optimizations are discussed below.

2.4.1. The basic algorithm

The tuples of the MF and DF which lie inside a given SR are to be merged. The Reorganize algorithm, (see chapter 5 section 4.1), can be used with two modifications:

1) The initial values of m and d are determined by random accesses based on the SR:

m := randac(MF, loval(SR))

d := randac(DF, loval(SR))

2) The merge terminates when m > hival(SR) and d > hival(SR).

2.4.2. Optimizations

The application of the optimizations of Rangesearch, (see chapter 3 section 2), to the current stiuation is not straightforward. For example, there are now two pages (one from the MF and one from the DF) to "scan to the end" of.

Recall that the original motivation for the optimization was to avoid the processing of a huge number of small SRs. Let us define a "small SR" as an SR completely contained within an MF cell. To process a small SR it is not incorrect to process instead the entire MF cell containing the SR, filtering out tuples which do not actually satisfy the query. This strategy will "skip over" nearby small SRs which fall in the same cell. After the MF cell is processed, the search can be reconstructed to resume at the tuple whose shuffle value is the smallest exceeding the hival of the cell.

2.4.3. Using MF inversions

To process a simple query on an access set for which an inversion exists, a different search algorithm is required. We can take advantage of the DF filter corresponding to the MFs inversion (see section 2.3.4).

The inversion stores the index values (and corresponding pointers) in an indexed-sequential data structure, ordered by index value. Each filter bit represents a range of index values. To process a range query on the access set of an inversion we begin with the filter. If a given filter bit is off, the MF is accessed through the inversion. Otherwise, the retrieved MF tuples are merged with the entries from the DF corresponding to the range of index values represented by the filter bit.

Chapter 7

Archives

It is sometimes useful to be able to evaluate a query on a previous state of the database. For example, if some statistic, (e.g. average salary), is to be plotted against time, the necessary data can be extracted by accessing certain past versions.

It is not feasible to anticipate all such requests and then retain the required information. A more general approach is to maintain archives: all previous states of the database.

Current relational database systems do not support archives. The idea is not even present in the relational algebra: a relation represents the state of a set of objects at some instant; it is a "snapshot" so the relational algebra does not contain operators for querying past states.

The differential file provides the ability to access the recent past: it stores all versions of a relation since the creation of the MF. (This capability is used to allow read-only transactions to avoid concurrency control.) In this chapter, the idea of the differential file is extended to provide archives. The implementation will be discussed. Archives will occupy a very large amount of memory and new mass storage technology will be useful in dealing with this volume. This issue will also be addressed.

1. Incorporating archives into a relational database system

The inclusion of archives affects all levels of a relational database system. As indicated above, the notion of "relation" needs to be reconsidered and the relational algebra must be extended. These problems are beyond the scope of this thesis. We will discuss the implementation of archives.

1.1. Querying recent versions

Regardless of how the relational algebra is extended, one point is clear: it will be necessary to refer to the relation as it existed at a specific moment in the past. The extraction of this version has been called the surface operator by Merrett [Merr81]. If each tuple is seen as being either present or absent in relation R at all times then surface(R,t) is the set of tuples that was present in R at time t, (see figure 1). Recall that all the updates of a transaction are marked with the timestamp of the transaction. That is, all of the updates take effect at the same time. In this way, surface sees either all of the updates from a given transaction or none of them.

For now, consider versions no older than the MF, (i.e. "recent" versions). Surface(R,t) can be retrieved as follows: perform a merge of the MF and DF, ignoring DF entries whose timestamps exceed t. The Reorganize algorithm of chapter 5 section 4.1 is easily modified to ignore the DF entries described.

To retrieve a very recent state it may be necessary to wait until committing transactions finish committing, (i.e. transactions which started committing but did not finish when the

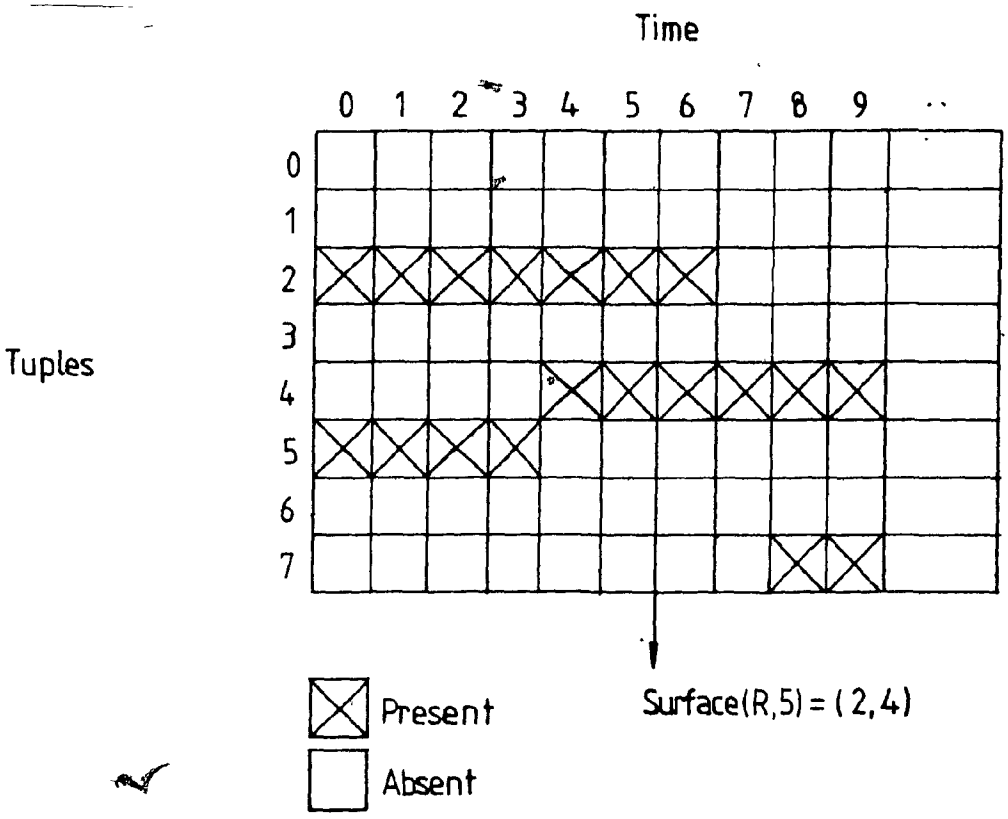


Figure 1. The surface operator.

search was initiated.)

The algorithm for querying a recent version (at time t) is simple. Normally, the DF is searched with a range query Q. To ignore all updates in Q's region dated later than t, search the DF using

$$Q_t = Q \text{ and } (\text{timestamp} \leq t)$$

In the DF, the timestamp was treated like any other attribute so Q_t is a range query and the usual search algorithm can be used. (This is essentially the "query modification" technique [Ston75].)

1.2. Querying any previous version

We now consider the problem of accessing versions older than the MF. There are a number of ways to modify the basic differential file scheme to support such accesses.

1.2.1. Never reorganize

If the entire relation is stored in the DF from the time it is created, (the MF is then empty and unnecessary), then all versions of the relation are present in the DF and the techniques of section 1.1 can be used.

This method is clearly impractical. Since the DF grows without limit it will become very expensive to search and keep on line. It is therefore necessary to update the MF and clear the DF periodically.

1.2.2. Keep old DFs

Following reorganization, the old DF is discarded. If it is kept then states that existed before reorganization can be accessed. If all old DFs are kept then any previous state can be accessed. These old DFs will be used as "anti-differential files" (ADFs) to construct earlier states from the MF, (see figure 2). The use of ADFs to implement archives was proposed by Merrett [Merr81]. Note that a DF can be used as an ADF by just reversing the meaning of the status flag. That is, $[a_0, \ldots, a_{k-1}, t, s]$ in the DF means that the status of $[a_0, \ldots, a_{k-1}]$ was changed to s at time t. In the ADF it means that before time t, the status was the opposite of s.

To query a version at time $t < t_M$ (where t_M is the time at which the MF was created) a new search predicate is defined:

$$Q_t = Q \text{ and } (\text{timestamp} \geq t)$$

Each ADF_i such that $t \leq t_i$ is searched using Q_t and the MF is searched using Q. The results from these files are merged as in Reorganize. (Actually, only the oldest accessed ADF, ADF_j where $t_{j-1} < t \leq t_j$, needs to be searched with Q_t. ADF_i, $i > j$ can be searched using Q since $t \leq t_i$ and $t_i < t_j$.)

Note that the DF stores tuples in reverse chronological order but the ADF must be accessed in (forward) chronological order. That is, to undo all of the updates of a tuple stored in an ADF, only the earliest update is of interest. The required modification to the merge algorithm is trivial.

The time required to evaluate a query on a version is directly related to the age of the version: older version will access more ADFs than younger versions.

1.2.3. Keep old MFs and DFs

If old MFs and DFs are kept then the time to access a previous version will be independent of the version's age. The price for this improved performance is that the MFs will occupy a very large amount of storage. This issue is discussed in section 3.

This organization is shown in figure 3.

To access a previous version at time t, find i such that $t_{i-1} \leq t < t_i$. Then search MF and DF_i (using Q_t = Q and (timestamp $\leq$ t) instead of Q for DF_i) and merge the results. This method is simpler than the use of ADFs since only two files are being merged and because there is no distinction made between recent versions ($t \geq t_M$) and older versions ($t < t_M$).

1.2.4. Keep some old MFs and all old DFs

Neither of the methods proposed in sections 1.2.2 and 1.2.3 is entirely satisfactory. For example, the use of ADFs may be unacceptably slow but the retention of all MFs may be impractical. Those methods are special cases of the following method: Store the r most recent MFs and DFs. All older versions can be reconstructed using ADFs (see figure 4).

Given a fixed amount of storage for the MFs, the value of r can be determined.

2. Implementing archives

We now discuss the data structures suitable for MFs, DFs and ADFs in the archives. The related issue of storage media is also addressed. We begin with the following two assumptions:

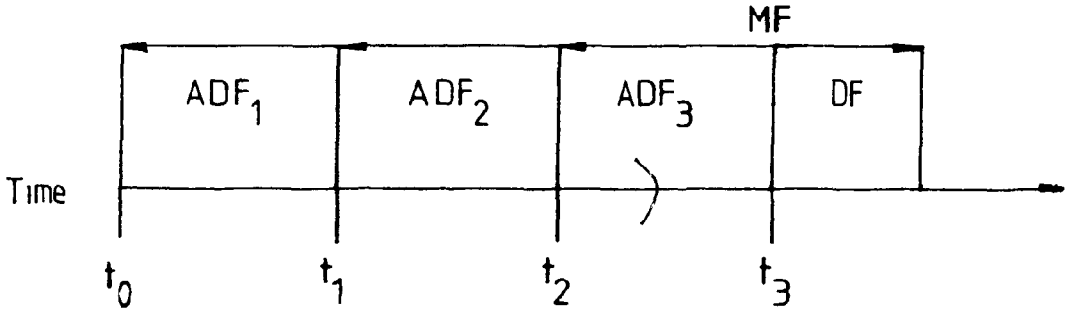


Figure 2. Archives using anti-differential files (ADFs).

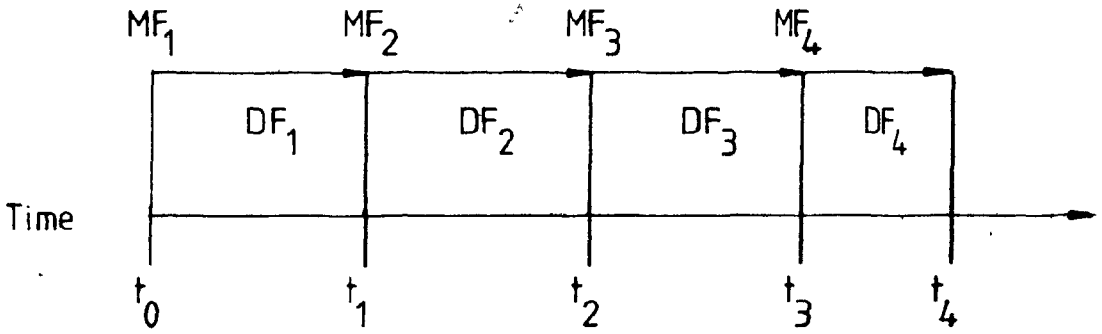


Figure 3. Archives using ADFs and old MFs and DFs.

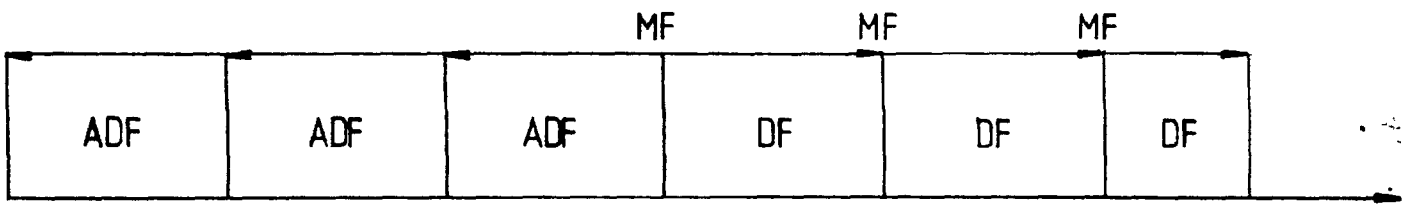


Figure 4. Archives using ADFs and recent MFs and DFs.

1) The archives are accessed much less frequently than is the current version.

2) Since the archives will be large and grow indefinitely, the issues of cost of storage and storage utilization are of primary importance.

The emphasis then, is on conserving storage, not on speed.

Eventually the archives will become so large that they cannot be managed at a reasonable cost. The obvious way to deal with this situation is to reclaim space occupied by the oldest ADFs, (assuming archives are stored as in section 1.2.4).

2.1. The MF

Two versions of the latest MF exist. One is stored on disk and is used for query processing. The other is a backup version stored on tape. Either version could be placed in the archives. Tape is cheaper but slower to use.

The data structure used for the MF, the zkd MLOPLH improved as in chapter 6 section 1.3.4, is suitable for use in the archives. The storage utilization is likely to be high (since it can be controlled). Furthermore, it is easy to show that storage utilization cannot fall below 25%. (More importantly, it cannot get arbitrarily close to zero.)

Based on our assumptions given above, inversions should not be stored: they occupy a significant amount of storage and speed is not a primary consideration.

2.2. The DF

The DF data structure, (zkd Btree or zkd MLOPLH), is also suitable for use in the archives. Storage utilization is kept in a fixed range.

No backup for the DF exists so it would have to be copied to a storage device explicitly for the sake of archiving: either to a disk for old DFs (and ADFs) or to the tape containing the old MFs.

2.3. Hard crash recovery reconsidered

The purpose of the hard crash recovery system is to restore damaged data structures. The log is used to recover the DF, filter and other dynamic objects. The MF is restored from a backup copy.

This system is as secure as its least secure component. Arbitrarily high degrees of security can be achieved by duplication of logs and backups but 100% protection is not possible. It is commonly assumed however, that the log is "stable", i.e. completely reliable.

The need for a backup of the MF can be reconsidered now that there are several old MFs present. Given that the log is stable and available as far back as the oldest MF, a backup of the current MF is unnecessary.

Suppose that the current MF, MF_r is damaged. (MF_1 is the oldest MF.) It can be constructed from the previous MF, MF_{r-1} and the log. If MF_{r-1} is damaged during reconstruction then MF_{r-2} can be used to restore MF_{r-1} which in turn can be used to restore MF_r. That is, a system with the r most recent MFs provides

security equivalent to r-1 backup MFs.

Generally, any MF_i can be restored from MF_j, $j < i$, so the archives are also protected, but the older an MF the less secure it is. The only irreplaceable component is MF_1, the oldest MF. If this is unacceptable then it could be protected by a backup copy created periodically. This backup would be created only during idle moments: its creation is not urgent. The most recent backup of this kind can be used with the log to restore MF_1.

3. Write-once memory

"Video disks" are new mass storage devices that are expected to be used in information systems in the near future [vand80]. Functionally, a video disk is a "write-once memory" (WOM). They are written and then used as read-only memory. These disks store data very densely and cheaply. They are, therefore, ideal for storing archives.

WOM is also ideal for backup purposes. After being written, the stored information is difficult to corrupt: the disk will never be put into a writing device again so only physical damage could affect the information.

If WOM were used, the life cycle of an MF would be as follows:

- The MF is created on WOM during reorganization and is used for processing queries.
- When the next MF is created the old MF is stored as part of the archives. This is also a suitable backup since WOM is difficult to corrupt.

WOMs cannot be erased and reused. Thus, the only limit to the

number of MFs that can be kept is warehouse space.

Chapter 8

Summary and Conclusion

In this thesis, a new design for the physical database was proposed. The distinctive features of this design are the data structures used, (the ZMDSs), and the use of the differential file scheme instead of a single dynamic file. In this chapter, the main results are summarized.

1. The kd trie

The kd trie (chapter 3 section 1) is a data structure which supports the efficient evaluation of partial match and range queries. It resembles the kd tree but differs in the way that discriminators are selected: with the kd trie, the discriminator that splits a region is completely determined by the region itself. The choice of a kd tree discriminator, on the other hand, is influenced by the points in the region. This difference has consequences in performance: The kd trie for a given set of points is unique. So updating cannot cause degradation of performance unless a "bad" distribution is achieved. The kd tree may degenerate. Furthermore, the balance of a kd tree cannot be maintained.

Analysis and experiments on the kd trie show that it is a practical data structure to use, (although it does have bad worst case performance).

The kd trie provides a consistent ordering of data. If tuple t precedes tuple t' in the inorder traversal of a kd trie containing both, then this ordering of t and t' occurs in all such kd tries. The kd tree does not have this property. In fact, two different kd trees storing the same data may yield different

inorder traversals.

The ordering of tuples provided by the kd trie is "z order". The kd trie made obvious the discovery of z order but is not necessary: precedence in z order can be decided without using a kd trie. This is so because the kd trie can be seen as a trie storing "shuffled" tuples, (i.e. the bits of the binary representations of the attribute values are interleaved). Z order corresponds to the usual numeric ordering of "shuffle values".

2. Z ordered multidimensional data structures, (ZMDSs)

The essential property of the trie, (as far as range searching is concerned), is that it is an indexed-sequential data structure (ISDS). That is, it supports random and sequential accessing. By storing shuffled tuples in any data structure with this property a ZMDS is obtained, (see chapter 3 section 2); the tuples are stored in z order. A ZMDS can be used to evaluate range queries efficiently.

The procedure for using a ZMDS is as follows:

- Shuffle all the tuples and store them in an ISDS.
- Evaluate range queries using the Rangesearch algorithm.
- Unshuffle the tuples returned by Rangesearch.

The input to the Rangesearch algorithm is a range query. It generates a sequence of "search regions" each of which generates one random access (to the ISDS) and at least one sequential access.

The advantages of the ZMDSs are due to their reliance on ISDSs. Existing ISDSs (e.g. ISAM, Btrees, etc.) can be used. The

analyses of the ISDSs also applies to the corresponding ZMDSs. Furthermore, any new results on ISDSs are immediately applicable to the ZMDSs.

3. Multi-level order preserving linear hashing, (MLOPLH)

We have modified linear hashing so that it can support sequential accessing, (see chapter 3 section 3). It is then an ISDS and therefore yields a ZMDS. This modification of linear hashing, order preserving linear hashing (OPLH), was achieved by using a monotonic hash function. The hash function used may not distribute the records to the buckets uniformly. If this happens, sequential processing may be slow. The problem is due to the appearance of buckets which are sparsely filled. "Multi-level" OPLH (MLOPLH) avoids this problem by combining sparse buckets with other buckets.

In its final form, MLOPLH has the following properties, (n is the number of tuples):

- A bit map containing one bit per bucket is used.
- Random accessing usually costs O(1) but is never worse than for a B+tree.
- Sequential accessing usually costs the same as for a B+tree; it may occasionally cost slightly more.
- Updates are usually no slower than O(log(n)). The worst case is $O(n/2^L)$, for some integer L. L can be set arbitrarily but there is a tradeoff: the worst case cost of a sequential access is $O(2^L)$.

To summarize our work on data structures: we have found a class of data structures, (ZMDSs), for evaluating range queries which is based on the class of ISDSs. The kd trie is the "founding member" of the ZMDS class. We have also found a new ISDS whose performance may be better than that of the B+tree. This therefore yields another ZMDS which has very good performance.

4. Transaction processing

In addition to supporting the basic operations on relations, a complete relational database system must be able to support concurrent access and be able to recover from soft and hard crashes.

The proposed design for the physical database is based on the use of a "differential file" system, (see chapter 5). A relation is stored in two parts:

1) The static master file (MF) represents a "snapshot" of the relation.

2) The dynamic differential file (DF) stores updates generated since the creation of the master file.

Both the MF and DF are searched in the evaluation of a query. Periodically, the MF and DF are merged to create a more up-to-date MF; the DF is then cleared.

This organization is, in some ways, more complicated than using a single dynamic file. For example, searching is more complicated. But other aspects of transaction processing are simplified. These are discussed below. The use of the

differential file organization also has advantages in performance due to the static nature of the MF.

The use of the differential file results in a "multiversion" database: all versions of a relation (since the creation of the MF) are available in the DF. This allows the use of (soft crash) recovery and concurrency control techniques that would not otherwise be possible.

The recovery system is simplified because the DF stores the most recent state of the relation: it is not necessary to recover from an out of date checkpoint, (which is slower and more complicated). Also, the cost of creating the checkpoint is avoided. Concurrency is enhanced because of the availability of old versions. Read-only transactions can be completely ignored by the concurrency control system.

---

5. Data structures for the system

The class of ZMDSs is a good source of data structures for the representation of the MF and DF, (see chapter 6):

- A large number of ZMDSs are known (due to the large number of known ISDSs).
- They can be searched efficiently.
- Tuples are stored in z order in all ZMDSs. Thus algorithms requiring merging can be used without sorting either operand, (even if the operands are stored in ZMDSs based on different ISDSs). Therefore the merge of the MF and DF does not require a sort of the large MF. Other multidimensional data structures do not have this property.

The MF should be represented by a zkd MLOPLH, (the ZMDS derived from MLOPLH). Since the MF is static, the tradeoff between the worst case costs of updating and sequential accessing is not a concern. Inversions on "important" access sets should be set up for dealing with simple queries. A large number of such inversions can be used because the MF is static.

The DF should be represented by the zkd Btree or zkd MLOPLH. (The latter cannot be recommended without reservation for use in a dynamic situation until more is known about its performance relative to that of the zkd Btree.)

A filter is worthless in processing complex partial match and range queries. But for processing simple queries on a single access set, a filter for the access set is quite valuable. One filter should, therefore, be set up for each MF inversion.

6. Archives

The design proposed for the physical database can be extended to provide archives, (see chapter 7), the ability to access old (but recent) states, (states created after the MF). By storing old DFs and/or old MFs it is possible to access any previous state.

If old MFs are stored they provide increased protection from hard crashes. Recovery is possible from the most recent undamaged MF and a log of transactions.

7. Conclusion

Existing implementations of relational databases are unnecessarily complicated and slow. Our design for the physical database is based on the differential file organization. This creates a "multiversion" database. The availability of old versions results in relatively simple recovery and concurrency control systems. There are also advantages in performance.

The components of the differential file organization should be represented by ZMDSs. This simplifies reorganization and permits the efficient evaluation of partial match and range queries. The MF should be augmented by inversions for processing simple queries. For each MF inversion, there is a corresponding filter for the DF.

By storing old MFs and DFs, the design can be extended to provide archives and improved protection from hard crashes.

Aho74 A.V. Aho, J.E. Hopcroft, J.D. Ullman.
The design and analysis of computer algorithms.
Addison-Wesley, Reading Mass., (1974).

Astr76 M.M. Astrahan et al.
System R: relational approach to database management.
ACM TODS 1, 2 (1976), 97-137.

Baye72 R. Bayer, E. McCreight.
Organization and maintenance of large ordered indexes.
Acta Informatica 1, 3 (1972), 173-189.

Baye80a R. Bayer, H. Heller, A. Reiser
Parallelism and recovery in database systems.
ACM TODS 5, 2 (1980), 139-156.

Baye80b R. Bayer et al.
Distributed concurrency control in database systems.
Proc. VLDB6, (1980), 275-284.

Bern81 P.A. Bernstein, N. Goodman.
Concurrency control in distributed database systems.
ACM Comp. Surv. 13, 2 (1981), 185-222.

Bent75a J.L. Bentley.
Multidimensional binary search trees used for associative searching.
CACM 18, 9 (1975), 509-517.

Bent75b J.L. Bentley, D.F. Stanat.
Analysis of range searches in quad trees.
Information Processing Letters 3, 6 (1975), 170-173.

Bent76 J.L. Bentley, W.A. Burkhard.
Heuristics for partial-match retrieval in database design.
Information Processing Letters 4, 5 (1976), 132-135.

Bent79a J.L. Bentley.
Multidimensional binary search trees in database applications.
IEEE Transactions on Software Engineering, SE-5, 4 (1979), 333-340.

Bent79b J.L. Bentley, J.H. Friedman.
Data structures for range searching.
ACM Computing Surveys 11, 4 (1979), 397-410.

Bjor75 L.A. Bjork.
Generalized audit trail requirements and concepts for database applications.
IBM Sys. J. 14, 3 (1975), 229-245.

Blas77 M.W. Blasgen, K.P. Eswaran.
Storage and access in relational databases.
IBM Sys. J. 4, (1977), 363-377.

Blas81 M.W. Blasgen et al.
System R: an architectural overview.
IBM Sys. J. 20, 1 (1981), 41-62.

Burk82 W.A. Burkhard.
Advances in interpolation-based index maintenance.
Proc. CISC (1982) Princeton Conference.

Bloo70 B.H. Bloom.
Space/time trade-offs in hash coding with allowable errors.
CACM 13, 7 (1970), 422-426.

Cham81 D.D. Chamberlin et al.
A history and evaluation of System R.
CACM 24, 10 (1981), 632-646.

Chan82 A. Chan et al.
The implementation of an integrated concurrency control and recovery scheme.
Proc. ACM SIGMOD, (1982)

Chiu82 G. Chiu.
MRDSA user's manual.
Technical Report SOCS 82.9, (1982), McGill University.

CODA71 CODASYL Data Base Task Group.
April 1971 Report.

Codd70 E.F. Codd.
A relational model of data for large shared data banks.
CACM 13, 6 (1970), 377-387.

Coff71 E.G. Coffman Jr., M.J. Elphick, A. Shoshani.
System deadlocks.
ACM Comp. Surv. 3, 2 (1971), 67-78.

Come79 D. Comer.
The ubiquitous B-tree.
ACM Computer Surveys 11, 2 (1979), 121-138.

Date77 C.J. Date.
An Introduction to Database Systems, 2nd Ed.,
Addison-Wesley, Reading Mass., (1977).

Devr82 L. Devroye.
A note on the average depth of tries.
Computing 28, (1982), 367-3nl.

Dodd69 G.G. Dodd.
Elements of data management systems.
ACM Comp. Surv. 1, 2 (1969), 117-132.

Eswa76 K.P. Eswaran, J.N. Gray, R.A. Lorie, I.L. Traiger.
The notions of consistency and predicate locks in a database system.
CACM 19, 11 (1976), 624-633.

Fagi79 R. Fagin, J. Nievergelt, N. Pippenger and H.R. Strong.
Extendible hashing - a fast access method for dynamic files.
ACM TODS 4, 3 (1979), 315-344.

Fink74 R.A. Finkel, J.L. Bentley.
Quad trees - a data structure for retrieval on composite keys.
Acta Informatica 4, 1 (1979), 1-9.

Fred60 E.H. Fredkin.
Trie memory.
CACM 3, 9 (1960), 490-499.

Glig80 V.D. Gligor, S.H. Shattuck.
On deadlock detection in distributed systems.
IEEE Trans. on Soft. Eng. SE-6, 5 (1980), 435-450.

Gopa80 V. Gopalakrishna, C.E.V. Madhavan.
Performance evaluation of an attribute based tree organization.
ACM TODS 5, 1 (1980) 69-87.

Gray78 J.N. Gray.
Notes on database operating systems.
In Operating systems: an advanced course,
Lecture notes in computer science, vol. 60,
Springer-Verlag, N.Y. (1978), 393-481.

Gray81 J. Gray et al.
The recovery manager of the System R database manager.
ACM Comp. Surv. 13, 2 (1981), 223-242.

Hall76 P.A.V. Hall.
Optimization of single expressions in a relational database system.
IBM J. of Res. and Dev. 20, 3 (1976), 244-257.

Hamm76 M. Hammer, A. Chan.
Index selection in a self-adaptive database management system.
Proc. ACM SIGMOD, (1976), 1-8.

Hard76 W.T. Hardgrave.
A technique for implementing a set processor.
Proc. Conf. on data: abstraction, definition and structure.
ACM N.Y. (1976), 86-94.

Have68 J.W. Havender.
Avoiding deadlock in multitasking systems.
IBM Sys. J. 7, (1968), 74-84.

IBM66 IBM Corp.
OS ISAM logic.
GY28-6618, IBM Corp. White Plains, N.Y. (1966).

Jord81 J.R. Jordan, J. Banerjee, R.B. Batman.
Precision Locks.
Proc. SIGMOD (1981), 143-147.

Kash77 R.L. Kashyap, S.K.C. Subas, S.B. Yao.
Analysis of the multiple-attribute-based tree organization.
IEEE Transactions on Software Engineering, SE-3, 6 (1977), 451-456.

Kim79 W. Kim.
Relational database systems.
ACM Comp. Surv. 11, 3 (1979), 182-212.

Knut73 D.E. Knuth.
The Art of Computer Programming, vol. 3: Sorting and Searching,
Addison-Wesley, Reading Mass., (1973).

Kung81 H.T. Kung, J.T. Robinson.
On optimistic methods for concurrency control.
ACM TODS 6, 2 (1981), 213-226.

Lars80 P.A. Larson.
Linear hashing with partial expansions.
Proc. VLDB6, (1980), 224-232.

Lee77 D.T. Lee, C.K. Wong.
Worst-case analysis for region and partial region searches in multidimensional binary search trees and balanced quad trees.
Acta Informatica 9, (1977), 23-29.

Lehm81 P.L. Lehman, S.B. Yao.
Efficient locking for concurrent operations on B-trees.
ACM TODS 6, 4 (1981), 650-670.

Liou77 J.H. Liou, S.B. Yao.
Multi-dimensional clustering for database organization.
Information Systems 2, 4 (1977), 187-198.

Litw80 W. Litwin.
Linear hashing: A new tool for file and table addressing.
Proc. VLDB6, (1980), 212-223.

Litw81 W. Litwin.
Trie hashing.
Proc, 19-29. ACM SIGMOD, (1981).

Liu76 J.W.S. Liu.
Algorithms for parsing search queries in systems with inverted file organizations.
ACM TODS 1, 4 (1976), 299-316.

Lohm77 G.M. Lohman, J.A. Muckstadt.
Optimal policy for batch operations: backup, checkpointing, reorganization and updating.
ACM TODS 2, 3 (1977), 209-222.

Lori74 R.A. Lorie.
XRM - An extended (n-ary) relational memory.
IBM Scientific Center Report G320-2096, Cambridge, Mass. (1974).

Lori77 R.A. Lorie.
Physical integrity in a large segmented database.
ACM TODS 2, 1 (1977), 91-104.

Lum70 V.Y. Lum.
Multi-attribute retrieval with combined indices.
CACM 13, 11 (1970), 660-665.

Mena79 D.A. Menasce, R.R. Muntz.
Locking and deadlock detection in distributed databases.
IEEE Trans. on Soft. Eng. SE-5, 3 (1979), 195-202.

Merr76 T.H. Merrett.
MRDS - an algebraic relational database system.
Proc. Canadian Computer Conf., Montreal (1976), 102-124.

Merr77 T.H. Merrett.
Relations as programming language elements.
Information Processing Letters 6, 1 (1977), 29-33.

Merr78 T.H. Merrett.
Multidimensional paging for efficient database querying.
Proc. International Conference of Management of Data, Milan 1978, 277-290.

Merr81 T.H. Merrett
Private communication.

Merr82 T.H. Merrett, E.J. Otoo.
Dynamic multipaging: a storage structure for large shared data banks.
Proc. 2nd Int'l Conf. on Databases: Improving Usability and Responsiveness, Jerusalem (1982).

Ober82 R. Obermarck.
Distributed deadlock detection algorithm.
ACM TODS 7, 2 (1982), 187-208.

Oren81 J.A. Orenstein.
Multidimensional tries used for associative searching.
Technical Report SOCS-81-23 (1981), McGill University.

Oren82a J.A. Orenstein.
Multidimensional tries used for associative searching.
Information Processing Letters 14, 4 (1982), 150-157.

Oren82b J.A. Orenstein, T.H. Merrett.
A class of data structures for associative searching.
Technical Report SOCS-82-10, (1982), McGill University.

Ouks81 M. Ouksel, P. Scheuerman.
Multidimensional B-trees: analysis of dynamic behaviour.
BIT 21, 4 (1981), 401-418.

Reed78 D.P. Reed.
Naming and synchronization in a decentralized computer system.
Technical Report MIT/LCS/TR-205, M.I.T., (1978).

Ries77 D.R. Ries, M.R. Stonebraker.
Effects of locking granularity in a database management system.
ACM TODS 2, 3 (1977), 233-246.

Ries79 D.R. Ries, M.R. Stonebraker.
Locking granularity revisited.
ACM TODS 4, 2 (1979), 210-227.

Ritc74 D.M. Ritchie, K.Thompson.
The UNIX time-sharing system.
CACM 17, 7 (1974), 365-375.

Rive74 R.L. Rivest.
Analysis of associative retrieval algorithms.
Ph.D. thesis, Stanford University (1974).

Robi81 J.T. Robinson.
The K-D-B tree: a search structure for large multidimensional dynamic indexes.
Proc. ACM-SIGMOD, Ann Arbor, Michigan 1981, 10-18.

Roth74 J.B. Rothnie, T. Lozano.
Attribute based file organization in a paged memory environment.
CACM 17, 2 (1974), 63-69.

Roth80 J.B. Rothnie et al.
Introduction to a system for distributed databases (SDD-1).
ACM TODS 5, 1 (1980), 1-17.

Seli79 P.G. Selinger et al.
Access path selection in a relational database management system.
Proc. ACM SIGMOD, (1979), 23-34.

Sche82 P. Scheuerman, M. Ouksel.
Multidimensional B-trees for associative searching in database systems.
Information Systems 7, 2 (1982), 123-137.

Schk75 M. Schkolnick.
The optimal selection of secondary indices for files.
Information Systems 1, 4 (1975), 141-146.

Seve76 D.G. Severance, G.M. Lohman.
Differential files: their application to the maintenance of large databases.
ACM TODS 1, 3 (1976), 256-267.

Shne73 B. Shneiderman.
Optimum data base reorganization points.
CACM 16, 6 (1973), 362-365.

Smit75 J.M. Smith, P.Y.T. Chang.
Optimizing the performance of a relational algebra database interface.
CACM 18, 10 (1975), 568-579.

Stan80 T.A. Standish.
Data Structure Techniques.
Addison-Wesley, Reading Mass., (1980).

Stea81 R. Stearns, D. Rosenkrantz.
Distributed database concurrency controls using before values.
Proc. ACM SIGMOD (1981), 74-83.

Ston75 M. Stonebraker.
Implementation of integrity constraints and views by query modification.
Proc. ACM SIGMOD (1975), 65-78.

Ston76 M. Stonebraker et al.
The design and implementation of INGRES.
ACM TODS 1, 3 (1976), 189-222.

Ston79 M. Stonebraker.
Concurrency control and consistency of multiple copies of data in distributed INGRES.
IEEE Trans. on Soft. Eng. SE-5, 3 (1979), 188-194.

Ston80 M. Stonebraker.
Retrospection on a database system.
ACM TODS 5, 2 (1980), 225-240.

Suss63 E.H. Sussenguth.
The use of tree structures for processing files.
CACM 6, 5 (1963), 272-279.

Tamm80 M. Tamminen.
The extendible cell method for fast geometric access.
Report HTKK-TKO-A20, (1980), Helsinkii University of Technology.

Tamm81 M. Tamminen.
Expected performance of some cell based organization schemes.
Report HTKK-TKO-B28, (1981), Helsinkii University of Technology.

Todd76 S.J.P. Todd.
The Peterlee relational test vehicle - a system overview.
IBM Sys. J., (1976), 285-308.

vand80 J.A. van de Vos.
Megadoc, a modular system for electronic document handling.
Philips Technical Review 39, 12 (1980), 329-343.

Verh77 J.S.M. Verhofstad.
Recovery and crash resistance in a filing system.
Proc. ACM SIGMOD, (1977), 158-167.

Verh78 J.S.M. Verhofstad.
Recovery techniques for database systems.
ACM Comp. Surv. 10, 2 (1978), 167-195.

Will82 R. Williams et al.
R*: an overview of the architecture.
Proc. Improving Database Usability and Responsiveness, (1982), 1-27.

Yao78 A.C.C. Yao.
On random 2-3 trees.
Acta Informatica 9, (1978), 159-170.