A Class of Data Structures for Associative Searching

J. A. Orenstein
Department of Computer and Information Science
University of Massachusetts, Amherst

T. H. Merrett
School of Computer Science
McGill University

Abstract

By interleaving the bits of the binary representations of the attribute values in a tuple, an integer corresponding to the tuple is created. A set of these integers represents a relation. The usual ordering of these integers corresponds to an ordering of multidimensional data that allows the use of conventional file organizations, such as Btrees, in the efficient processing of multidimensional queries (e.g. range queries). The class of data structures generated by this scheme includes a type of kd tree whose balance can be efficiently maintained, a multidimensional Btree which is simpler than previously proposed generalizations, and some previously reported data structures for range searching. All of the data structures in this class also support the efficient implementation of the set operations.

1.0 INTRODUCTION

The most fundamental action of any information system is the retrieval of the records of a file that satisfy some condition. If the records are identified by a "primary key" and the search condition involves no other fields of the file, then a large number of data structures and search algorithms are available; this is one of the most thoroughly studied problems in computer science.

Efficient searching is more difficult if the search condition involves fields other than those of the primary key. In recent years, a number of solutions to this problem have appeared (see [Bent79b, Oren83a] for surveys). Some of these are *ad hoc* extensions of techniques for primary key searches. Some are "biased" so that searching on some fields is much less efficient than searching on others. All of them discuss the problem in terms of one particular data structure. For example, the kd tree [Bent75, Bent79a] is derived from the binary tree but cannot be based, instead on the AVL tree.

Our approach is more general. Any data structure that efficiently supports random and sequential accessing can be adapted for the efficient evaluation of a large and useful class of queries. A *random access* for x is a search for the record of a file whose key is the smallest not less than x, according to some ordering. A *sequential access* locates the successor of a record according to the ordering. Data structures with these two properties are "indexed-sequential data structures" (ISDSs). The class of ISDSs includes some of the most widely used file organizations, e.g. the Btree [Baye72] and its variants (see [Come79] for a survey) and ISAM [IBM66]. With this approach, several new data structures for range searching have been obtained and several known data structures have been re-discovered.

Our motivation in studying this problem was to find data structures for use in a relational database system. This is reflected in our terminology. Instead of "file", "field" and "record", we will use the terms *relation*, *attribute* and *tuple*. A relation, R, defined on k attributes, A_0, ... , A_{k-1}, is $R(A_0, \ldots, A_{k-1})$. k is the *degree* of the relation. Associated with each attribute A_i is a *domain*, $dom(A_i)$, the set of values which A_i may assume. A tuple of R is $[a_0, \ldots, a_{k-1}]$ where $a_i \in dom(A_i)$.

A *range query* is a Boolean function of one tuple which specifies a range of values for each attribute:

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

If $L_i = -\infty$ and $U_i = \infty$ then A_i is not of interest and the clause can be omitted from the query. The class of range queries is a large and useful one. A subset is the class of *partial match* queries in which $L_i = U_i$ or $L_i = -\infty$ and $U_i = \infty$, i=0, ... , k-1. That is, either a single value is specified for each A_i or A_i is not of interest. Even this smaller class of queries is important. For example, searching for the records with a given value of the primary key is a partial match query with k=1. We will be concerned with the more general class of range queries.

We will use the following notation. $\langle s_1{:}n_1 \mid s_2{:}n_2 \mid \ldots \mid s_m{:}n_m \rangle$ denotes the string

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

where each s_i is a string of bits. If n_i is 1 then $s_i{:}n_i$ may be written as s_i. For example, $\langle 011{:}2 \mid 0\rangle = 0110110_2$.

2.0 THE Z ORDERING

The elements of each $dom(A_i)$ are usually ordered. Furthermore, in practice, the domains are finite in size. For example, names can be ordered lexicographically and can be represented by fixed-length strings of characters. If the cardinality of a domain is D then a value from this domain can be represented by one of the integers in {0, ... , D-1}. The ordering of the integers reflects the ordering of the domain values.

It is sufficient, then, to assume that $dom(A_i)=\{0, 1, \ldots, D_i-1\}$ where $D_i=|dom(A_i)|$, i=0, ... , k-1. It will be convenient to assume that each D_i is a power of 2. Then $d_i=\log_2 D_i$ is the number of bits required to represent elements of $dom(A_i)$. The number of bits needed to represent a tuple is $h=d_0+\ldots+d_{k-1}$.

Viewing each attribute value as an integer, a tuple $[a_0, \ldots, a_{k-1}]$ can be seen as the point $(a_0, \ldots, a_{k-1})$ in kd space. Correspondingly, a range query can be represented by a hyper-rectangle. The response to a query is the set of tuples whose points are inside the hyper-rectangle. This hyper-rectangle is the *query region* (QR). Figure 1 shows how tuples and range queries are represented in a 2d space.

In order to process queries efficiently, tuples that will be retrieved together should be stored together on as few pages as possible. For range queries, this means that tuples which are close in space should be stored near each other. Sorting on the primary key achieves this for the primary key only. What is needed is a way of ordering the points so that closeness in the kd space in *any* direction results in closeness in the ordering.

The operation of *shuffling* creates such an ordering - the *z-ordering* of tuples. This ordering is demonstrated in figure 2. The "z path", in two dimensions, starts in the lower left corner, visits every point in z order and terminates in the upper right corner.

A tuple is shuffled by interleaving the bits of the binary representations of the attribute values. Suppose that each $D_i=2^d$. Then the representation of $P = [P_0, \ldots P_{k-1}]$ in binary is $P = [P_{00}P_{01}\ldots P_{0,d-1}, \ldots, P_{k-1,0}P_{k-1,1}\ldots P_{k-1,d-1}]$ where P_{ij} is the jth bit of P_is binary representation. Z order, denoted by $\leq_z$, 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 indicated bits. Thus point A of figure 1 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 $A \leq_z B$.

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. 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 modifying the shuffling function. To describe a particular shuffling function, an array, attr, can be used to indicate the attribute from which each bit of the shuffle value was taken. For the cyclic shuffling function, attr[i]=i mod k, i = 0, ... , d-1. Attr can be used to describe arbitrary shuffle functions. 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, $\text{shuffle}([00_2, 111_2]) = 01101_2$ and $\text{shuffle}([10_2, 001_2]) = 10001_2$ so $[00_2, 111_2] \leq_z [10_2, 001_2]$.

Sorting can be achieved by not doing any "interesting" interleaving; instead, the attribute values are just concatenated:

$$\text{TrivialShuffle}(X) = X_{00}X_{01}\ldots X_{0,d-1}X_{10}X_{11}\ldots X_{1,d-1}\cdots X_{k-1,0}X_{k-1,1}\ldots X_{k-1,d-1}$$

The only requirement on the shuffling function is that it does not change the order of bits from any attribute value.

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 is applied to tuples retrieved in a search before they are returned to the caller.

3.0 EVALUATING RANGE QUERIES

In order to evaluate a query, a data structure storing the tuples and a search algorithm are required. The data structure we require is any ISDS storing shuffled tuples. The search algorithm to be described relies on the z-ordering of the tuples created by the storing of shuffled tuples in the ISDS.

The search algorithm generates k dimensional "search regions" (SRs). A subset of the generated SRs partition the QR. By retrieving all points in each partition, the query is evaluated.

The steps taken by the search algorithm depend on the relationship of the SRs with the QR. For each SR, three cases can occur:

1. The SR is outside the QR. The SR does not contain any points whose tuples satisfy the query. Nothing needs to be done in this case.
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 returned to the caller.

3. The SR overlaps but is not inside the QR. In this case, the SR is split into two smaller SRs which are handled recursively.

Testing the relationship between the SR and the QR can be performed easily and without reference to the data. Cases (1) and (3) can also be carried out without accessing the data. The ISDS is accessed only in case (2).

The key to this scheme is the construction of the SRs in case (3); i.e. the way in which an SR is split into two. If done in a certain way, the retrieval of tuples in case (2) will be simple and efficient. The method is the following. On the ith level of recursion, split the SR evenly into two SRs along *attr[i]*. Then the tuples retrieved in case (2) will be consecutive in z order. Since the ISDS stores the tuples in z order, the tuples in the SR can be retrieved in a nearly minimal number of disk accesses.

To see why this works, first consider 1d data from the domain $\{0, \ldots, 2^d\text{-}1\} = \{\langle 0{:}d\rangle, \ldots, \langle 1{:}d\rangle\}$ (recall that the cardinality of each domain is a power of 2). One tuple of a relation defined on this domain is a d bit number $X = \langle X_0 \mid \ldots \mid X_{d-1}\rangle$. Each bit of X corresponds to a split of a region of the space into two sub-regions of equal size; the bit is 0 in one sub-region, 1 in the other. For example, X_0 partitions the region $\langle 0 \mid 0{:}d\text{-}1\rangle{:}\langle 1 \mid 1{:}d\text{-}1\rangle$ into $\langle 0 \mid 0{:}d\text{-}1\rangle{:}\langle 0 \mid 1{:}d\text{-}1\rangle$ and $\langle 1 \mid 0{:}d\text{-}1\rangle{:}\langle 1 \mid 1{:}d\text{-}1\rangle$. X_1 partitions each of these into two sub-sub-regions and so on. In general, the ith bit partitions $\langle X_0 \mid \ldots \mid X_{i-1} \mid 0 \mid 0{:}d\text{-}i\text{-}1\rangle{:}\langle X_0 \mid \ldots \mid X_{i-1} \mid 1 \mid 1{:}d\text{-}i\text{-}1\rangle$ into $\langle X_0 \mid \ldots \mid X_{i-1} \mid 0 \mid 0{:}d\text{-}i\text{-}1\rangle{:}\langle X_0 \mid \ldots \mid X_{i-1} \mid 0 \mid 1{:}d\text{-}i\text{-}1\rangle$ and $\langle X_0 \mid \ldots \mid X_{i-1} \mid 1 \mid 0{:}d\text{-}i\text{-}1\rangle{:}\langle X_0 \mid \ldots \mid X_{i-1} \mid 1 \mid 1{:}d\text{-}i\text{-}1\rangle$. Note also that a prefix corresponds to a region of the space; e.g. $\langle X_0 \mid \ldots \mid X_{i-1}\rangle$ is a prefix of all values (which represent points) in the range $\langle X_0 \mid \ldots \mid X_{i-1} \mid 0{:}d\text{-}i\rangle{:}\langle X_0 \mid \ldots \mid X_{i-1} \mid 1{:}d\text{-}i\rangle$.

Next, consider a kd space. Each bit of the shuffle value represents a split of a kd region. In one sub-region the value of the bit is 0, in the other it is 1. The direction of the split is determined by the attribute from which the bit was taken, attr[i]. The SRs generated in case (3) are kd sub-regions which can be described by prefixes of the shuffle value. (Each recursive call in case (3) fixes the value of one more bit of the prefix.) Let us denote the range of shuffle values corresponding to such a prefix by $SR_{lo}{:}SR_{hi}$. In the kd space, this range of shuffle values corresponds to a region containing an *uninterrupted* segment of the z path.

We will denote SRs by $[l_0{:}u_0, \ldots, l_{k-1}{:}u_{k-1}]$. Then $SR_{lo}=\text{shuffle}([l_0, \ldots, l_{k-1}])$ and $SR_{hi}=\text{shuffle}([u_0, \ldots, u_{k-1}])$.

Now the handling of case (2) can be explained. To retrieve all of the tuples in an SR, retrieve all of the tuples, t, such that $SR_{lo} \le \text{shuffle}(t) \le SR_{hi}$. This can be accomplished in two steps:

1. Perform a random access on the ISDS using SR_{lo} as the search argument. The first tuple in the SR, according to z ordering, will be retrieved.
2. Perform sequential accesses on the ISDS until the shuffle value of the retrieved tuple exceeds SR_{hi}.

Figure 2 demonstrates the partitioning of the QR from figure 1. Note that each partition contains an uninterrupted segment of the z path.

The search algorithm is given below in greater detail. The algorithm is invoked by calling RangeSearch(QR, $[0{:}D_0\text{-}1, \ldots, 0{:}D_{k-1}\text{-}1]$, 0). The first argument is the query region; the second is the search region, initially the entire space; the third argument is the level of recursion, initially 0.

```
RangeSearch(QR, SR, level)
if SR∩QR = ∅    (* case 1 *)
then    (* do nothing *)
else if SR⊆QR    (* case 2 *)
then    (* retrieve all tuples in the SR *)
   t := randac(SR_lo)
   while tuple(t)≤SR_hi
     report unshuffle(tuple(t))
     t := seqac(t)
   end
else    (* SR∩QR≠∅    case 3 *)
   (* This case cannot occur if the SR is *)
   (* a single point. *)
   RangeSearch(QR, left(SR, attr[level]), level+1)
   RangeSearch(QR, right(SR, attr[level]), level+1)
end if

return
end RangeSearch
```

Tuple(t) returns the shuffled tuple located at address t. Randac(x) locates the first tuple of the relation in z order whose shuffle value is at least x and returns the address of that tuple. Seqac(t) returns the address of 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 using a bit of attribute a and returns the "left" sub-region; the sub-region for which the selected bit is 0. Right(SR,a) returns the corresponding "right" sub-region for which the selected bit is 1. 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.

Left and right can be calculated easily. If a is the attribute being split and the range of attribute a in the SR is l_a:u_a then the corresponding ranges in left(SR) and right(SR) are, respectively, l_a:u_a' and l_a':u_a, where $u_a'=(l_a+u_a-1)/2$ and $l_a'=(l_a+u_a+1)/2$.

4.0 REFINEMENTS TO RANGESEARCH

The RangeSearch algorithm of section 3 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 2 illustrates the problem. Near the query boundaries, many very small SRs (i.e. one or two bit regions) may be generated. (In a much larger space with a correspondingly larger QR, the problem would be more serious.) The number of these SRs is determined by the location and dimensions of the query and is independent of the data stored. 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 is h, the number of bits required to represent a tuple. Each SR occupies 2h bits so the stack of SRs occupies $2h^2$ bits. Since, in practice, h may be in the thousands, the size of the stack can be a problem.

4.1 Eliminating the stack

The stack of SRs does not have to be stored explicitly. 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, ... , l_a:u_a, ... , l_{k-1}:u_{k-1}] where a=attr[i], then LSR(i)=[l_0:u_0, ... , l_a:u_a', ... , l_{k-1}:u_{k-1}] and RSR(i)=[l_0:u_0, .., l_a':u_a, ... , 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 one SR and a stack of bits. The ith bit indicates whether SR(i+1)=LSR(i) or SR(i+1)=RSR(i).

Even this small stack of bits can be eliminated. The binary representation of l_a:u_a' (or l_a':u_a) is <*:d-c-1 | Y | 0:c>:<*:d-c-1 | Y | 1:c> where c≥0, d 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=<*:d-c-1 | 0 | 0:c>:<*:d-c-1 | 1 | 1:c>. 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.

4.2 Reducing the number of search regions generated

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

A leaf page of such an ISDS contains tuples whose shuffle values are s_1, ... , s_p where p is the number of tuples on the page and $s_1 < ... < s_p$. Many small SRs are likely to fall between adjacent tuples. Along a query boundary of size x there can be up to O(x) small SRs. 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, ... , a_{k-1}] then the corresponding region is [a_0:a_0, ... , a_{k-1}:a_{k-1}]. To begin reconstruction of the state of the search algorithm we set SR = [a_0:a_0, ... , 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 4.1, ancestral SRs can be reconstructed. 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 space that should not be explored. If there is overlap and the child

was a right child then both children of the SR have already been explored.) With SR and level reset, the search resumes with RSR(level). The modified RangeSearch algorithm is given below.

Figure 3 illustrates the search using the data of figure 1. The tuples have been stored on pages 1-4 in z order and have been placed at postions in the diagram representing their shuffle values. The query is broken up into SRs (displayed beneath the tuples); each SR corresponds to one of the partitions of figure 2. In the original version of RangeSearch, each SR would be searched in turn yielding tuples 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. 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. *)

SR := [0:D0-1, ... , 0:Dk-1-1]
level := 0
search-done := false

repeat until search-done
      if SR⊆QR then
         Process(SR, t)
            (* get all the tuples in the SR and *)
            (* scan the remainder of the last *)
            (* page read *)
         level := h
         SR := [a0:a0 , ... , ak-1:ak-1]
            (* where [a0, ... , ak-1] is the *)
            (* last tuple on the page *)
         Reconstruct(SR, level, search-done)
         SR := right(SR, attr[level])

      else if SR∩QR≠∅
         (* explore the left SR first *)
         SR := left(SR, attr[level])

      else
         (* SR∩QR=∅ *)
         Reconstruct(SR, level, search-done)
         SR := right(SR, attr[level])

      end if

      level := level+1
end
return
end RangeSearch
```

```
Process(SR, t)
(* Retrieve all tuples in the SR, starting on the *)
(* page accessed by SRlo. (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(SRlo)
while tuple(t)≤SRhi
   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 tuple(t)≤tuple(end-of-page)
   if tuple(t)∈QR then report tuple(t)
   t := seqac(t)
end

return
end Process
```

```
Reconstruct(t, SR, level, search-done)
repeat until level=0
      or (SR∩QR≠∅ and childside=left)
   Restore(SR, level, childside)
end
search-done := (level=0 and childside=right)

return
end Reconstruct
```

```
Restore(SR, level, childside)
level := level - 1
a := attr[level]
(* the range of attribute a of SR, la:ua, *)
(* expressed in binary, is *)
(* <*:d-c-1 | Y | 0:x> : <*:d-c-1 | Y | 1:x>. *)
if Y=0
then
   childside := left
   ua := 2 ua - la + 1
else
   childside := right
   la := 2 la - ua - 1
end

return
end Restore
```

If the ISDS is tree-based but does not store all data in the leaves (e.g. the Btree) then a simple modification of this scheme can be used. The idea is to make sure that reconstruction is always done from a leaf page. For

details, see [Oren83a].

4.3 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 4). 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 from [Oren83a]. The QR considered was $[0{:}X_0\text{-}1,\ 0{:}X_1\text{-}1]$. 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 $X=X_0=X_1$, $R(QR)=O(2^q)$ where q is the distance between the leftmost and rightmost 1 in the binary representation of X [Oren83a]. (The case in which $X_0 \neq X_1$ was also discussed in [Oren83a].) This result says that R(QR) is oscillatory; R([0:X-1, 0:X-1])=R([0:2X-1, 0:2X-1]). This is so 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⊂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 $X'=0011100_2$ satisfies our requirements.

There is a relationship between the bits of X and the partitioning of the QR. Increasing X to X′ corresponds to moving the boundaries of the query outward so that larger (but 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 not(t∈QR) then delete t from report
end

return
end RangeSearchQE
```

5.0 MULTIDIMENSIONAL DATA STRUCTURES BASED ON Z ORDER

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 (required by RangeSearch) must be supplied for each 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 5). We now discuss a number of ZMDSs.

5.1 Zkd binary search

The simplest ISDS is the (1d) array of ordered data. By storing shuffled tuples in z order, 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 r. Seqac(t) increments t (an index to the array) by 1. Of course, this data structure can only be used efficiently for static files.

5.2 Zkd tree

By using the binary tree as the ISDS, a more dynamic ZMDS is obtained. It is not the same as Bentley's kd tree [Bent75, Bent79a] since 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 7, maintenance of 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 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.

5.3 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 than Robinsons's K-D-B tree [Robi81] and Scheuermann and Ouksel's MDB tree [Sche81]. 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.

5.4 Kd Trie

Orenstein's kd trie [Oren82] is the ZMDS obtained from Fredkin's trie [Fred60].

5.5 EXCELL

Tamminen's EXCELL [Tamm81] is an MDS based on 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.

5.6 Multiple attribute trees

The MDB tree of Scheuermann and Ouksel [Sche82] is related to the doubly-chained tree [Suss63], the multiple attribute tree [Kash77] and the modified multiple attribute tree [Gopa80]. These are all ZMDSs which use TrivialShuffle but differ in the particulars of the underlying data structure. Customized search algorithms, different from RangeSearch, were proposed for these data structures.

5.7 Zkd order-preserving linear hashing

Linear hashing is a data structure by Litwin which is dynamic and can usually locate a record in one disk access [Litw80]. It can be made order-preserving - it is then an ISDS and yields a ZMDS. The resulting data structure has been discovered twice previously. Burkhard uses a completely different search algorithm from those presented here. It uses k nested loops, one for each attribute. Each loop explores the range of values of the corresponding attribute which is covered in the query [Burk83]. Ouksel and Scheuermann found the same structure but did not give an algorithm for range searching [Ouks83].

Both of these data structures are, essentially, order-preserving linear hashing (OPLH) transformed as described here, (used with different search algorithms). Some deficiencies of OPLH are explored by Orenstein in [Oren83b]. To avoid these problems, "multi-level" OPLH was proposed. This ISDS may have superior performance to that of the Btree. It can, of course, also be transformed into a ZMDS.

6.0 OTHER USES OF Z ORDERING

Data in files are often ordered to allow the 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 such as the kd tree and the KDB tree.

7.0 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. The expected cost of partial match and range queries given a certain distribution of tuples was analyzed in [Oren83a]. The results are the same as for the kd tree, the kd trie 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). The advantage of our approach is that it is more general since it builds upon a large class of data structures, the ISDSs. Also, ZMDSs guarantee z order, permitting the efficient implementation of algorithms that require merging.

8.0 SUMMARY AND CONCLUSIONS

We have presented a method for converting any ISDS into a data structure for range searching. Our method is to store in an ISDS integers derived from the tuples. To evaluate a query, the RangeSearch algorithm is used to generate random and sequential accesses to the ISDS.

This class of data structures is ideal for representing relations in a relational database system because it provides efficient implementation of range searching and the set operations.

References

[Baye72] R. Bayer, E. McCreight.
Organization and maintenance of large ordered indexes.
Acta Informatica 1, 3 (1972), 173-189.

[Bent75a] J.L. Bentley.
Multidimensional binary search trees used for associative searching.
Comm. ACM 18, 9 (1975), 509-517.

[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 Comp. Surv. 11, 4 (1979), 397-410.

[Burk83] W.A. Burkhard.
Interpolation-based index maintenance.
Proc. 2nd ACM SIGACT-SIGMOD Symposium on Principles of Database Systems,
(1983), 76-89.

[Come79] D. Comer.
The ubiquitous B-tree.
ACM Comp. Surv. 11, 2 (1979), 121-138.

[Fagi79] R. Fagin, J. Nievergelt, N. Pippenger, H.R. Strong.
Extendible hashing - a fast access method for dynamic files.
ACM Trans. on Database Systems 4, 3 (1979), 315-344.

[Fred60] E.H. Fredkin.
Trie memory.
Comm. ACM 3, 9 (1960), 490-499.

[Gopa80] V. Gopalakrishna, C.E.V. Madhavan.
Performance evaluation of an attribute based tree organization.
ACM Trans. on Database Systems 5, 1 (1980), 69-87.

[IBM66] IBM66 Corp.
OS ISAM logic.
GY28-6618, IBM66 Corp. White Plains, N.Y. (1966).

[Kash77] R.L Kashyap, S.K.C. Subas, S.B. Yao.
Analysis of the multiple-attribute-tree based organization.
IEEE Trans. on Soft. Eng. SE-3, 6 (1977), 451-456.

[Knut73] D.E. Knuth.
The Art of Computer Programming, vol. 3: Sorting and Searching,
Addison-Wesley, Reading Mass., (1973).

[Litw80] W. Litwin.
Linear hashing: a new tool for file and table addressing.
Proc. VLDB6, (1980), 212-223.

[Oren82] J.A. Orenstein.
Multidimensional tries used for associative searching.
Information Processing Letters 14, 4 (1982), 150-157.

[Oren83a] J.A. Orenstein.
Data Structures and Algorithms for the Implementation of a Relational Database System, Ph.D. thesis,
McGill University (1983).
Also available as Technical Report SOCS-82-17.

[Oren83b] J.A. Orenstein.
A dynamic hash file for random and sequential accessing.
Proc. VLDB9, (1983), 132-141.

[Ouks83] M. Ouksel, P. Scheuermann.
Storage mappings for multidimensional linear dynamic hashing.
Proc. 2nd ACM SIGACT-SIGMOD Symposium on Principles of Database Systems,
(1983), 90-105.

[Robi81] J.T. Robinson.
The K-D-B tree: a search structure for large multidimensional
dynamic indexes. Proc. ACM SIGMOD, (1981), 10-18.

[Sche82] P. Scheuermann, M. Ouksel.
Multidimensional B-trees for associative searching in database systems.
Information Systems 7, 2 (1982), 123-137.

[Suss63] E.H. Sussenguth.
The use of tree structures for processing files.
Comm. ACM 6, 5 (1963), 272-279.

[Tamm81] M. Tamminen.
Order preserving extendible hashing and bucket tries.
BIT 21, 4 (1981), 419-435.

[Yao78] A.C.C. Yao.
On random 2-3 trees.
Acta Informatica 9, (1978), 159-170.

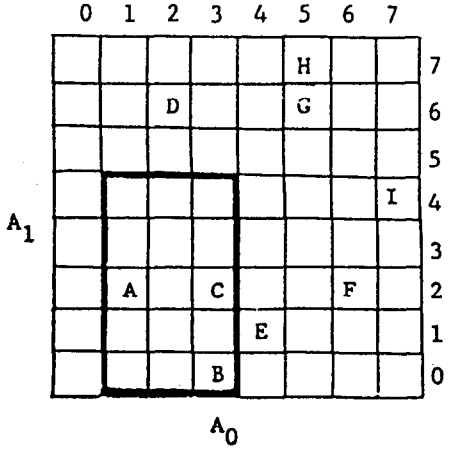

Figure 1.

In a 2d space, tuples are represented by rectangles. The query shown is $1 \leq a_0 < 3$ & $0 \leq a_1 \leq 4$. The response is {A, B, C}.

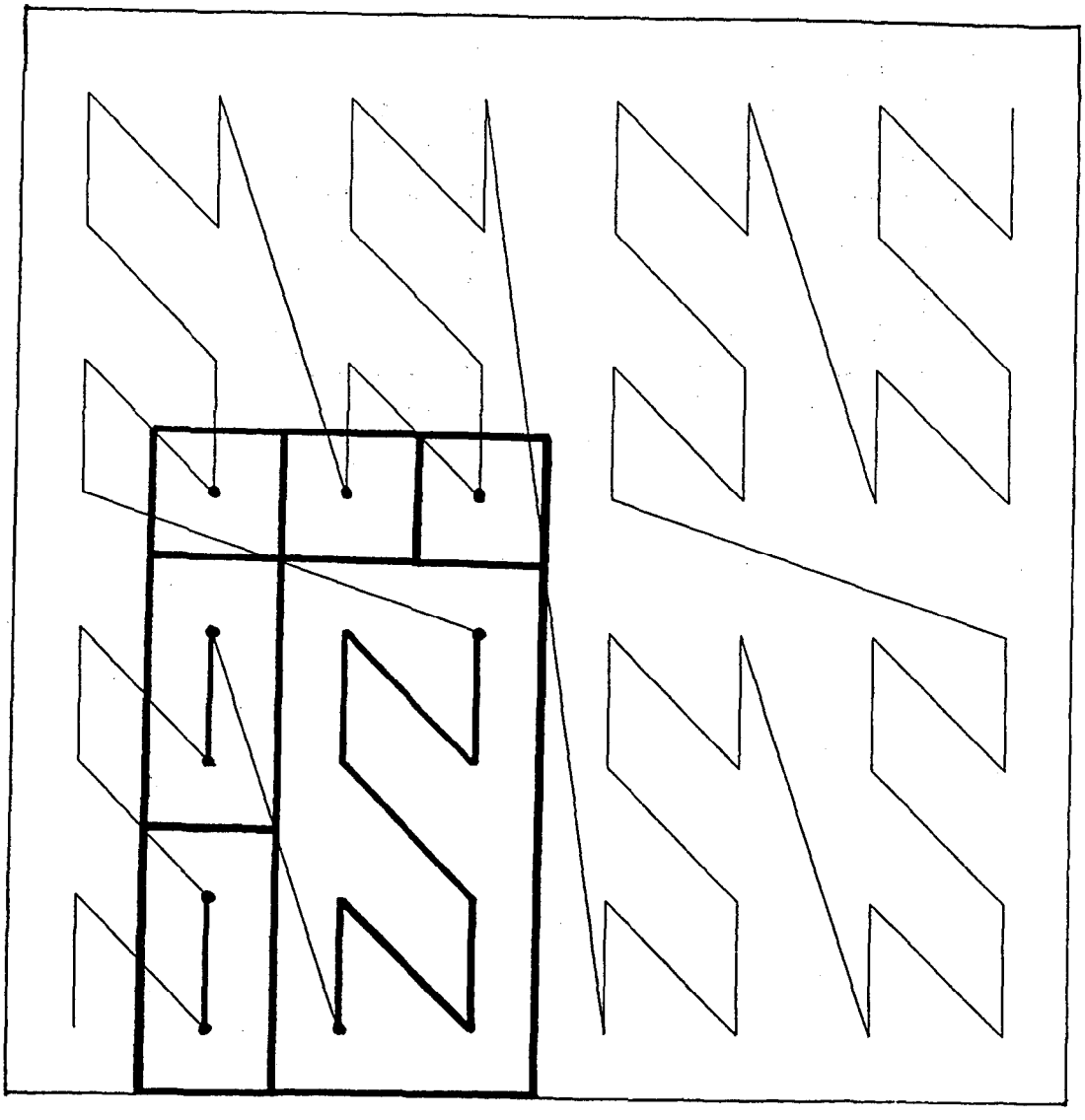

Figure 2. The points of each SR are consecutive in z order.

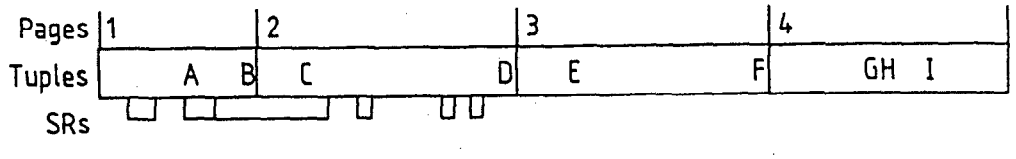

Figure 3. The relationship between tuples, pages and search regions.

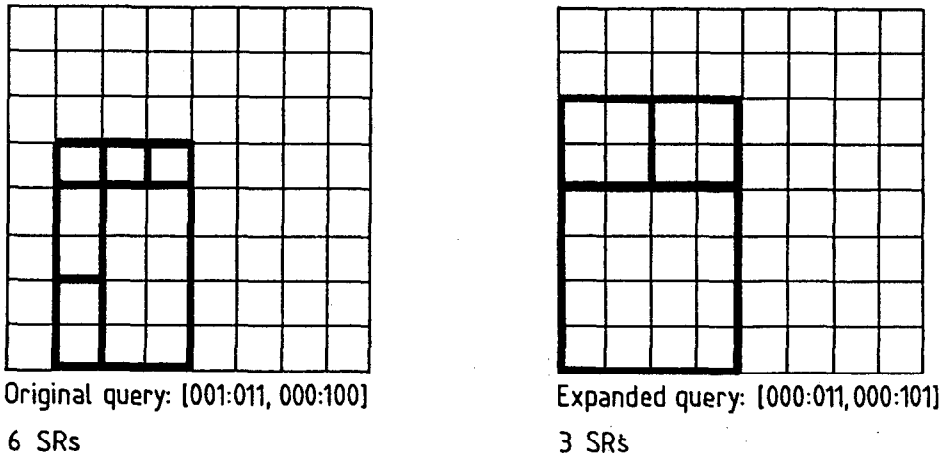

Figure 4. Query expansion

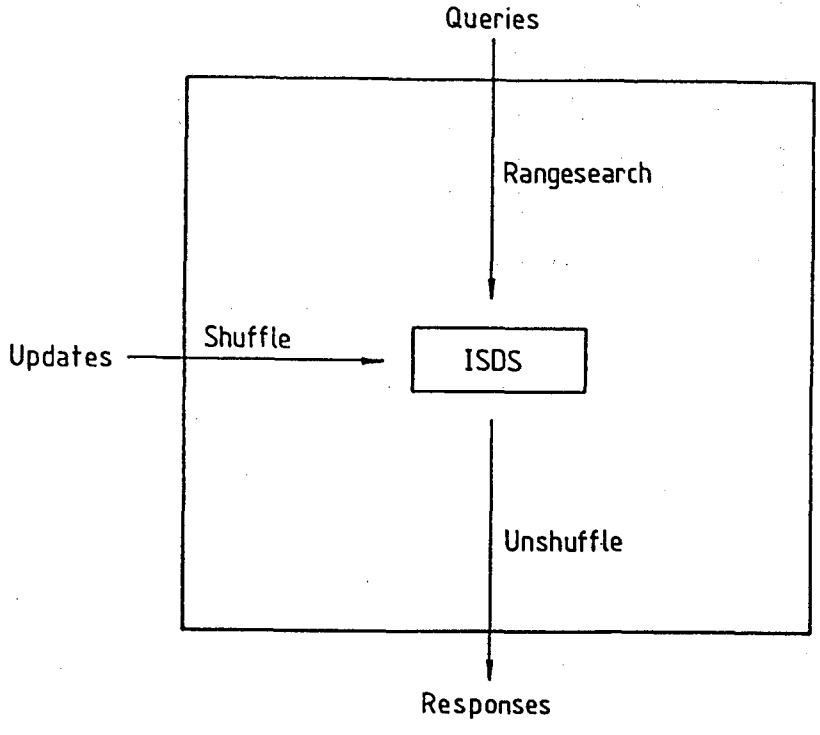

Figure 5. Converting an ISDS into a ZMDS.