ALGORITHM AND PERFORMANCE EVALUATION OF ADAPTIVE MULTIDIMENSIONAL CLUSTERING TECHNIQUE

Shinya FUSHIMI*, Masaru KITSUREGAWA**, Masaya NAKAYAMA*
Hidehiko TANAKA*, and Tohru MOTO-OKA*

*Department of Electrical Engineering
University of Tokyo

**Institute of Industrial Science
University of Tokyo

ABSTRACT

This paper proposes the multidimensional clustering technique called GKD-tree method which is fully adaptive to both of tuple and query distributions. The method is based on the theoretical analysis of the performance of multidimensional clustering. Its algorithm and performance are described. It is shown that GKD-tree method can largely reduce the average number of page accesses. For several distributions, its behavior is analyzed, and the modification to the algorithm is suggested to further improve the performance. GKD-tree method was first developed as the physical database organization of the relational database machine GRACE. The implementation issues of GKD-tree method in the database machine are also discussed.

1. Introduction

Multidimensional clustering technique has been widely recognized as the promising method of the physical database design[5]. Several algorithms have been proposed and evaluated[4-13]. But none of them were fully adaptive to both distributions of tuples and queries. In general, tuples are distributed with some biases in the space formed as the Cartesian product of domains of attributes. For example, the relation EMPLOYEE(NAME, AGE, DEPARTMENT, SALARY) may contain tuples most of which have values from 20 to 40 in the AGE attribute when they share the value "programming" in DEPARTMENT attribute. In addition, queries issued to that relation are distributed also with some biases in the space consisting of all possible queries. To see this, consider the typical situation in which a number of queries to EMPLOYEE relation refer to the NAME attribute with restriction predicate on DEPARTMENT. Another example is that the clerk of the bank runs the database application program many times which is applied to checking accounts of less than 1000 dollars balance to subtract the check service charge. Therefore, the ultimate objective the multidimensional clustering technique should achieve is summarized to minimize the average number of page accesses for given distributions of tuples and queries, which reveal such several biases in the real environment.

In this paper, the new multidimensional clustering technique called generalized KD-tree (abbreviated GKD-tree) method is proposed. GKD-tree method is essentially the extension of KD-tree method[4], and uses the space splitting technique, as opposed to the metric-based one[7]. The technique achieves the adaptive page partitioning to both of distributions based on the theoretical analysis of the average number of page accesses with multidimensional page partitioning assumed. Its performance is evaluated by comparing with that of KD-tree method. It is shown that GKD-tree method can largely reduce the average number of page accesses. This method, however, is not always superior to KD-tree method. Then, the algorithm is further improved by analyzing its behavior of partitioning process. GKD-tree technique is developed as the physical database organization of the parallel relational database machine GRACE[1,2,15]. Hence it accompanies some assumptions on its implementation environment. These implementation issues are also discussed. We will use the terms of relational model such as relation and attribute throughout the remaining part of this paper.

2. Basic Theorem and Queries

2.1. Basic Theorem for Analyzing Multidimensional Clustering

Let R be the relation having k attributes $A_1, \ldots, A_k$. And let T and N be the number of tuples of R and the number of pages storing R,

respectively. $\boldsymbol{D}$ denotes the Cartesian product of the domains of attributes referred to by the relation, i.e., $\boldsymbol{D}=\prod_{i=1}^{i=k} \mathrm{dom}(A_i)$. We call $\boldsymbol{D}$ the *base space of the relation*. To derive the formula which expresses the average number of page accesses, we use two distribution functions:

(1) Tuple Distribution Function $D(t)$
$D(t)$ is defined so that $D(t)=n$ iff $R(A_1,\cdots,A_k)$ contains n copies of the tuple $t=(v_1,\ldots,v_k)$. Unless $R(A_1,\cdots,A_k)$ is created by *transpose*, clearly $n=0$ or 1. Trivially,

$$\int_{t\in \boldsymbol{D}} D(t)\,dt = T.$$

$D(t)$ affects the possible partitionings of tuples into pages.

(2) Query Distribution Function $Q(q)$
$Q(q)$ is the normalized distribution function of the query, that is,

$$Q: \boldsymbol{Q} \rightarrow [0,1]$$

where

$$\boldsymbol{Q} = \{\ \textit{all possible queries}\ \},$$

and

$$\int_{q\in \boldsymbol{Q}} Q(q)\,dq = 1.$$

Let $\pi(q)$ denote the number of pages accessed by the query q. Suppose $\boldsymbol{D}$ is divided into N pages $p_1,\ldots,p_N$[1]. In general, $\bar{\pi}$, the average number of page accesses for given $Q(q)$, can be obtained first by multiplying $\pi(q)$ by $Q(q)$, then by integrating it on q. Then we have,

$$\bar{\pi} = \int_{q\in \boldsymbol{Q}} \pi(q)Q(q)\,dq. \qquad (1)$$

Here we introduce some useful functions and predicates. $\Omega(p_j,q)$ is defined to be true, if the page p_j is accessed by q, and false, otherwise. Then define the binary function $\delta(p_j,q)$ as follows:

$$\delta(p_j,q) = \begin{cases} 1 & \text{if } \Omega(p_j,q)=\textit{true} \\ 0 & \text{if } \Omega(p_j,q)=\textit{false}. \end{cases}$$

Now we have as the concrete form of $\pi(q)$:

$$\pi(q) = \sum_{j=1}^{N} \delta(p_j,q). \qquad (2)$$

Further define the following set and function:

$$\Gamma(p_j) = \{\ q\in \boldsymbol{Q} \mid \Omega(p_j,q)\ \}$$

and

$$H(p_j) = \int_{q\in\Gamma(p_j)} Q(q)\,dq.$$

We have the following theorem.

[Theorem 1] For given $Q(q)$ and the set of pages $\{p_1,\ldots,p_N\}$, the average number of page accesses is given by

$$\bar{\pi} = \sum_{i=1}^{N} H(p_j). \qquad (3)$$

[Proof]

$$\begin{aligned}
\bar{\pi} &= \int_{q\in \boldsymbol{Q}} \pi(q)Q(q)\,dq \\
&= \int_{q\in \boldsymbol{Q}} \Big(\sum_{j=1}^{N} \delta(p_j,q)\Big)Q(q)\,dq && \text{(by (2))} \\
&= \sum_{j=1}^{N} \int_{q\in \boldsymbol{Q}} \delta(p_j,q)Q(q)\,dq \\
&= \sum_{j=1}^{N} \int_{q\in \Gamma(p_j)} Q(q)\,dq && \text{(by definition)} \\
&= \sum_{j=1}^{N} H(p_j). && \text{(by definition)}
\end{aligned}$$

The theorem claims that the average number of page accesses can be given by summing up $H(p_j)$, the weight of p_j obtained by integrating the weights of queries accessing p_j. Note that $\bar{\pi}$ is determined only by $Q(q)$ and $\{p_1,\ldots,p_N\}$, and that the theorem holds independent of the clustering algorithms.

2.2. Queries

We will focus the selection operations and consider queries formed by conjunctions of ranges. That is, the domain of each attribute assumed to be the ordered set, and the query issued by the user can be put in the the general form:

$$x_1 \leqq v_1 \leqq y_1 \wedge, \ldots, \wedge x_k \leqq v_k \leqq y_k,$$

whose meaning is that "select tuples where v_i, the value of its i-th attribute, is greater than or equal to x_i and less than or equal to y_i". All the domains are further assumed to be upper and lower bounded sets, i.e., for

[1] *Page* means here the subspace of the base space of the relation. Tuples in the *page* are subject to be stored in one *actual page*. This rule gives the one-to-one mapping between *pages* and *actual pages*

any $v_i \in dom(A_i)$, $MIN_i \leqq v_i \leqq MAX_i$. Then, the predicate bounded by MIN_i and MAX_i can be interpreted as no selection predicate for attribute A_i is involved in the query. Hence we can assume without loss of generality that every query refers to *all* of attributes. Considering $x_i \leqq y_i$, we can regard the query as the point $\prod_{i=1}^{k}[x_i, y_i]$ in the $2k$-dimensional space. That is,

$$Q = \{ \prod_{i=1}^{k}[x_i, y_i] \mid x_i, y_i \in dom(A_i) \text{ for } all\ i \}.$$

This space is formed as the Cartesian product of k right-angled triangles. The i-th component of the query corresponds to the point in the i-th right-angled triangle Q_i as shown in Fig.1. Three points in Fig.1 show three kinds of possible cases. The point A in Fig.1 represents the normal range predicate for attribute A_i, whereas the point B and C correspond to the situations that no predicate and exact match predicate are given to A_i respectively. Therefore, this class is general and large enough to include not only the range queries, but exact match and partial match queries.

Since we consider the range queries and multidimensional clustering, the predicate $\Omega(p_j, q)$ defined above becomes definite. By *multidimensional clustering*, we mean the clustering procedure which produces pages by splitting the base space recursively along more than one axes. Therefore, all of pages are of hyperrectangle form in the base space: $p_j = \prod_{i=1}^{k}[\alpha_{ij}, \beta_{ij}]$. Hence, we can express the page as the point in Q, as well as the query (Fig.1). Then, $\Omega(p_j, q)$ is reduced to the following form:

$$\begin{aligned} \Omega(p_j, q) &\equiv p_j \cap q \neq \phi \\ &\equiv x_i \leqq \beta_{ij} \wedge y_i \geqq \alpha_{ij} \text{ for all } i \end{aligned} \quad (4)$$

Now, for given page $p_j = \prod_{i=1}^{k}[\alpha_{ij}, \beta_{ij}]$, the queries which satisfy $\Omega(p_j, q)$ can be represented as the shaded area in Fig.1 for each attribute. This area gives the exact representation of $\Gamma(p_j)$.

The formula (3) can be computed for some simple distributions of queries and shows interesting results[15]. For example, if we assume that range queries are uniformly selected on Q, and that the base space is partitioned into the lattice where the domain of A_i is divided into n_i intervals, we have:

$$\begin{aligned} \bar{\pi} &= \prod_{i=1}^{k}\left(\frac{n_i}{3} + 1 - \frac{1}{3 \cdot n_i} \right) \\ &\approx \prod_{i=1}^{k}\left(\frac{n_i}{3} \right) \\ &= O\left(\frac{N}{3^k} \right). \end{aligned}$$

That is, the average number of page accesses is reduced inversely proportional to the 3's power to the k-th. On the other hand, we have the performance of the traditional one-dimensional access method by letting $n_1 = N$, $n_2 = \cdots = n_k = 1$:

$$\begin{aligned} \bar{\pi}' &= \frac{N}{3} + 1 - \frac{1}{3 \cdot N} \\ &= O\left(\frac{N}{3} \right). \end{aligned}$$

This result clearly shows the potential superiority of the multidimensional clustering to the classical one-dimensional access method.

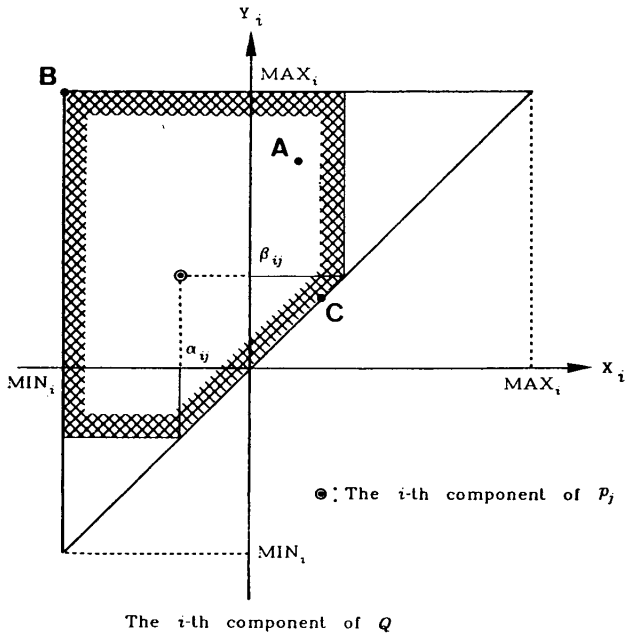


Fig.1 Representation of Queries and Page

3. Algorithm of Generalized KD-tree

3.1. Generalized KD-tree

Generalized KD-tree (GKD-tree) is essentially the extension of KD-tree[4,5]. For comparison, we first briefly review the KD-tree method. For given relation whose tuples fit in N pages, *KD-tree* method first selects the value of attribute A_1 of some tuple (say γ_1) such that it divides the whole set of tuples evenly into two subspaces D_l and D_r. That is, D_l and D_r are created by cutting the base space along the line (value of A_1) $= \gamma_1$ so that D_l and D_r contains N/2 tuples each, and that D_l contains all the tuples whose A_1 values are

less or equal to γ_1, while D_r contains others. Here the value thus selected and the attribute A_1 are respectively called *discriminator value* and *discriminator attribute*. We simply call this pair *discriminator*. For discriminator value γ_1 and discriminator attribute A_1, we have

$$D_l = [MIN_1,\gamma_1]\times[MIN_2,MAX_2]\times\cdots\times[MIN_k,MAX_k],$$

and

$$D_r = [\gamma_1,MAX_1]\times[MIN_2,MAX_2]\times\cdots\times[MIN_k,MAX_k].$$

Then this step is recursively applied to D_l and D_r with page number N/2 and with the discriminator attributes changed in the cyclic order as $A_2, A_3, \ldots, A_k, A_1, \ldots$ until the produced subspace contains tuples which fit in one page. The *KD-tree* is created at the same time whose node stores the pair of discriminator attribute and value produced at each recursion step along with pointers to its left and right sons. The discriminator attribute of the node at level i is thus i modulo $k+1$. Since KD-tree method divides the base space recursively into hyperrectangle pages, range queries are elegantly processed and page overflow/underflow problems are resolved by local and dynamic page split/merge technique[4-6]. In addition, this method is adaptive to the tuple distribution in the sense that pages created by the algorithm are always fully loaded with tuples, in contrast to the multidimensional trie methods[11,12]. On the other hand, it is obvious that the KD-tree algorithm only respects the distribution of tuples and ignores how and what queries are issued. The tuples frequently accessed in a bunch may be divided into rather many pages. As a result, relatively many pages would be accessed on the average.

The algorithm can be made adaptive also to the query distribution by first introducing the degree of freedom on selecting the discriminators, and then by supplying the appropriate procedure to select the optimal discriminator at each partitioning step: provided that the space concerned contains tuples of n pages, we can use any of k attributes and, for each of them, any of n-1 values as discriminator candidates[2]. When the procedure to select discriminators is given, page partitioning will proceed as follows. First we select one of $k\cdot(N-1)$ discriminator candidates by this procedure, and split the base space by them. Suppose the candidate (A_i,γ_i) is selected and the resultant subspaces contain tuples of n and $N-n$ pages respectively. Then this step is recursively applied to the subspaces with the numbers of pages n and $N-n$ until the resulted space contains tuples of one page or less. For such page partitioning, the index structure is created as in the KD-tree method. The node stores the the pair of discriminator attribute and value along with pointers to left and right sons. The tree thus constructed is called the *generalized KD-tree* (abbreviated *GKD-tree*).

[2]Although KD-tree is generalized in similar form in[6], we take the original definition in [4] as the KD-tree.

The only difference from the KD-tree method is that discriminator attributes and values are selected, not by the circular shifting and evenly partitioning basis, but by some procedure which should give optimal discriminator pairs.

The flexibility of the algorithm can be identified by the number of partitionings it can generate (denoted by C(N), where N is the number of pages into which tuples are partitioned). For GKD-tree, C(N) is computed by the recursion:

$$C(N) = \sum_{i=1}^{N-1} k\cdot C(i)\cdot C(N-i)$$

Then we have

$$C(N) = \frac{1}{N}\cdot\begin{pmatrix}2N-2\\N-1\end{pmatrix}\cdot k^{N-1} \qquad (N\geqq 1)$$

$$\approx (4k)^{N-1}/N\cdot\sqrt{\pi\cdot(N-1)}.$$

For KD-tree, this number is always one, since the sequence of discriminator values and attributes are determined a priori for given tuple distribution. It is clear that the GKD-tree gives much more flexibility to the query distribution as well as tuple distribution than KD-tree. This suggests that if such a flexibility can be fully utilized by the appropriate algorithm for selecting discriminators, the average number of page accesses will be largely reduced. Note that GKD-tree still enjoys advantages of KD-tree such as good clustering property even for range queries and ability to split/merge pages dynamically in case of page overflow/underflow. Note also that GKD-tree achieves the largest number of partitionings under the condition that pages are always fully loaded.

3.2. Algorithm of Selecting Discriminator Attributes and Values

Recall the clustering principle that the set of tuples frequently accessed in the lump should be stored in as small number of pages as possible. The same effect would be achieved if we follow the strategy that the set of tuples frequently accessed should *not* be partitioned and the set of tuples rarely accessed should be first partitioned as long as possible. We can identify such tuples by the following theorem.

Suppose the page $p_j = \prod_{i=1}^{k}[\alpha_{ij},\beta_{ij}]$ is to be divided into two on the discriminator attribute A_i by the discriminator value γ_{ij} $(\alpha_{ij}\leqq\gamma_{ij}\leqq\beta_{ij})$. Then there produced are two subspaces $p_j{}^l(\gamma_{ij})$ and $p_j{}^r(\gamma_{ij})$, where

$$p_j{}^l(\gamma_{ij}) = [\alpha_{1j},\beta_{1j}]\times\cdots\times[\alpha_{ij},\gamma_{ij}]\times\cdots\times[\alpha_{kj},\beta_{kj}],$$

and

$$p_j^r(\gamma_{ij}) = [\alpha_{1j},\beta_{1j}]\times\cdots\times[\gamma_{ij},\beta_{ij}]\times\cdots\times[\alpha_{kj},\beta_{kj}].$$

Let $p_j^{\Delta}(\gamma_{ij})$ be $p_j^l(\gamma_{ij}) \cap p_j^r(\gamma_{ij})$, which is the hyperplane in the base space.

[Theorem 2]

$$H(p_j)+H(p_j^{\Delta}(\gamma_{ij})) = H(p_j^l(\gamma_{ij}))+H(p_j^r(\gamma_{ij})), \quad (5)$$

or if we write $H(\alpha_{1j},\beta_{1j},\ldots,\alpha_{kj},\beta_{kj})$ for $H(p_j)$,

$$\begin{aligned} & H(\alpha_{1j},\beta_{1j},\ldots,\alpha_{ij},\beta_{ij},\ldots,\alpha_{kj},\beta_{kj}) \\ + & H(\alpha_{1j},\beta_{1j},\ldots,\gamma_{ij},\gamma_{ij},\ldots,\alpha_{kj},\beta_{kj}) \\ = & H(\alpha_{1j},\beta_{1j},\ldots,\alpha_{ij},\gamma_{ij},\ldots,\alpha_{kj},\beta_{kj}) \\ + & H(\alpha_{1j},\beta_{1j},\ldots,\gamma_{ij},\beta_{ij},\ldots,\alpha_{kj},\beta_{kj}) \quad (6) \\ & (\alpha_{ij}\leqq\gamma_{ij}\leqq\beta_{ij}). \end{aligned}$$

[Proof]
Consider Q_i, the projection of Q onto the domain of attribute A_i (Fig.2). The whole shaded area represents the queries accessing p_j in the figure. When this page is partitioned into $p_j^l(\gamma_{ij})$ and $p_j^r(\gamma_{ij})$ by the value γ_{ij}, $[\alpha_{ij},\beta_{ij}]$, the i-th component of p_j, is divided into $[\alpha_{ij},\gamma_{ij}]$ and $[\gamma_{ij},\beta_{ij}]$. Other components of p_j are left unchanged. In Q_i, the queries accessing these subpages

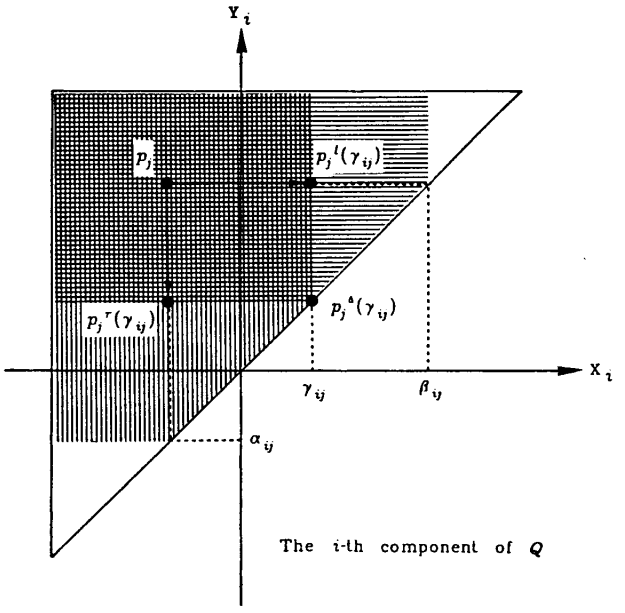


Fig.2 Relationship between Page Partioning and Page Boundaries

are depicted as vertically and horizontally shaded areas respectively in Fig.2. The cross-shaded area means the overlap of these two query sets, and can be regarded as the queries accessing the virtual page $p_j^{\Delta}(\gamma_{ij})=[\alpha_{1j},\beta_{1j}]\times\cdots\times[\gamma_{ij},\gamma_{ij}]\times\cdots\times[\alpha_{kj},\beta_{kj}]$. By the well-known property on the additiveness of the integration regions, clearly the theorem holds.

Suppose that tuples be partitioned into N pages $p_1,\ldots,p_N$. The average number of page accesses is expressed in (3). Now further partition the page p_j on attribute A_i by the value γ_{ij}. By theorem 2, we have the new average number of accesses as:

$$\begin{aligned} \bar{\pi}' &= H(p_1) + \cdots + H(p_{j-1}) \\ &\quad + H(p_j^l(\gamma_{ij})) + H(p_j^r(\gamma_{ij})) \\ &\quad + H(p_{j+1}) + \cdots + H(p_N) \\ &= \sum_{j=1}^{N} H(p_j) + H(p_j^{\Delta}(\gamma_{ij})) \\ &= \bar{\pi} + H(p_j^{\Delta}(\gamma_{ij})). \end{aligned}$$

Therefore, the average number of accesses increases by $H(p_j^{\Delta}(\gamma_{ij}))$ by this page split. It should be noticed that if the discriminator value is selected so that the set of tuples frequently accessed is partitioned, then $p_j^{\Delta}(\gamma_{ij})$ is virtually accessed by queries frequently issued, hence $H(p_j^{\Delta}(\gamma_{ij}))$ becomes bigger. From this observation, we have the following algorithm:

Input:
N, the number of pages the tuples fit into, and D, the space to be partitioned.

Output:
The partitioned page space and its associated GKD-tree. The node of GKD-tree consists of four fields: discriminator attribute field, discriminator value field, and two pointers to left and right sons.

Algorithm:
First select N-1 candidates for each of k attributes to get all of possible discriminator values. This is done by sorting tuples on each attribute, and then by selecting values which are on the page capacity boundaries. Then compute $H(p_j^{\Delta}(\gamma_{ij}))$ $(i=1,\ldots,k, j=1,\ldots,N\text{-}1)$ for each of $k\cdot(N-1)$ candidates. Next, take value and attribute such that they give the minimum value of $H(p_j^{\Delta}(\gamma_{ij}))$, then split the space by them into left and right subspaces as described above. Allocate the node and store this pair as the discriminator therein. These form one recursion step. Suppose the base space be divided at this step into left subspace (D_l) and right subspace (D_r) containing tuples of n and $N-n$ pages respectively. Then apply this step recursively to each of D_l and D_r with page number n and $N-n$ respectively, until the

given space contains the tuples of the page size or less. At every recursion step, two pointers in the nodes are arranged so that they correctly point to their left and right sons to maintain the recursive structure of page partitioning. When there exist more than one candidates which share the minimum, selected is the one which splits the given space most evenly to make the resultant tree balanced as much as possible.

This algorithm does not accompany the backtrack but goes straight by selecting the locally optimal discriminator at each recursion step. Therefore, it does not always achieve the exactly optimal partitioning. But it is guaranteed therein that the bunch of tuples frequently accessed together is never partitioned by utilizing the degrees of freedom on selecting discriminators. The large performance improvement is expected. Note that the algorithm is fully adaptive to the tuple distribution. This is implicitly incorporated when discriminator candidates are computed, that is, they are selected so that resulted pages are fully loaded.

3.3. Time Complexity

Assuming the time complexity of the computation of $H(p_j^{\Delta}(\gamma_{ij}))$ be 1, we show the whole time complexity (denoted by TC(N)) of the algorithm for partitioning the base space into N pages.

The best case is when tuples are always divided evenly. Then

$$TC(N) = k \cdot (N-1) + 2 \cdot TC(N/2),$$

that is,

$$TC(N) = k \cdot ((\log N - 1) \cdot N + 1)$$
$$= O(k \cdot N \cdot \log N).$$

The GKD-tree is then completely balanced.

The worst case corresponds to when one of two resultant spaces contains only tuples of one page at every recursion step. We have for this case,

$$TC(N) = k \cdot (N-1) + TC(N-1),$$

hence,

$$TC(N) = k \cdot N \cdot (N-1)/2$$
$$= O(k \cdot N^2).$$

The GKD-tree for such page partitioning grows into the linearly connected link rather than the balanced tree.

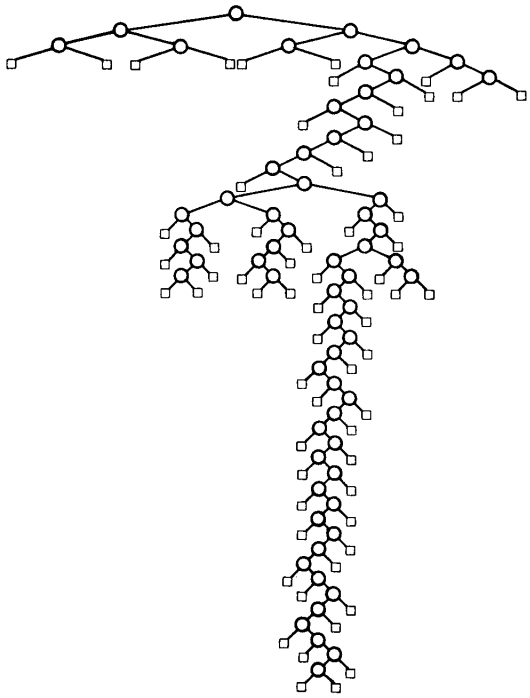
4. Performance Evaluation

4.1. Behavior of Algorithm

We first examine the behavior of GKD-tree algorithm while it partitions the base space. As mentioned before, the query distribution does not seem to be considered uniform in the real environment. Among the tuples, some are frequently accessed and others are not. This introduces the undulation of the access pattern in D: there are many "mountains" and, inevitably, many "valleys". (It is difficult for our intuition to grasp this pattern correctly since Q has the dimension two times larger than D. See section 4.3) Therefore, the query and tuple distributions are artificially generated to offer such various circumstances.

Example I: We examine the behavior of the algorithm for the following query and tuple distributions. We make "mountain"s in the query distribution by overlapping five subdistributions each of which contains one "mountain" in the different area, as illustrated in Fig.3 where $k=2$. The details on this query distribution can be found in Appendix. By GKD-tree method, the base space is partitioned into 64 pages. Tuple distribution is assumed to be uniform for simplicity. Therefore, all of pages have the same size of area. Fig.3 also shows the GKD-tree produced for these distributions. We observe that as a whole, the algorithm behaves in the way we want it to do. The algorithm first tries to divide the base space D so as not to cut across the top of the "mountain". In other words, the base space is cut along the "valleys". In this phase, the corresponding GKD-tree grows to become balanced. After that, it reluctantly begins to cut the mountain areas, while still obeys the order that the summit area of the mountain should not be cut. So it creeps and goes around the skirts of the mountain to climb and cut. The GKD-tree resulted in this phase has the shape of almost linearly connected link. As a consequence, the GKD-tree is so produced that it is globally balanced but locally grown straight. Note that this shape directly reflects the query distribution. Thus, since the leaves at rather low level point the pages near the top of the mountain, the lower level the leaves exist, the more frequently the corresponding pages are accessed,

Example II: Next, we examine the case in which the tuple distribution is also biased. In the query distribution, one "mountain" is simply placed at the center of the base space. On the other hand, the tuple distribution function has $2 \cdot k$ "mountain"s at symmetric places near boundaries of the base space. This situation is illustrated in Fig.4, where $k=2$. The details about these two distributions are specified in Appendix. Since only the center of the base space is frequently accessed, the space is partitioned so that the center area is surrounded by many small pages. The GKD-tree produced is essentially the linear chain, and never balanced. It should be noticed that GKD-tree technique skillfully makes use of the fact that tuples and query accesses are accumulated in the different areas. Tuples are accumulated in the area to which accesses rarely occur, hence, partitioned into rather many but small pages. On the other hand, the area frequently accessed is split into only a few pages.



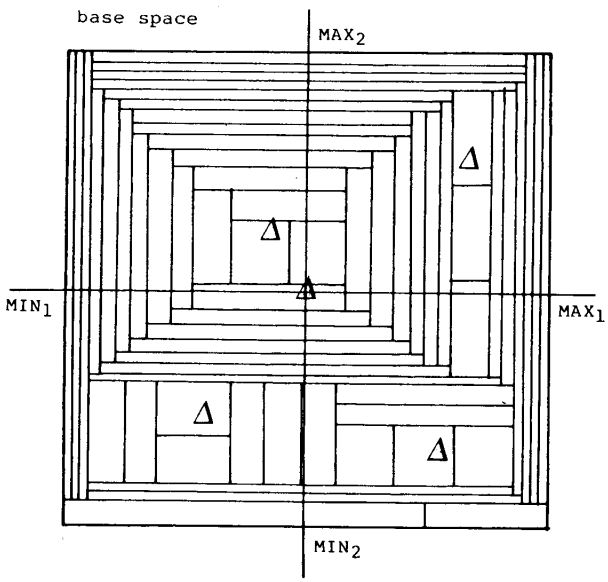
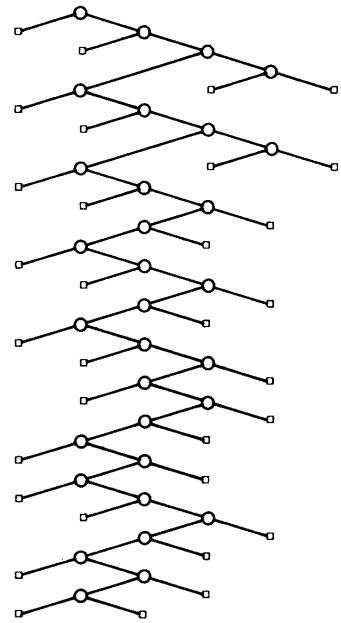


Fig.3 Example of Page Partition and GKD-tree (I)
(N = 64, k = 2)



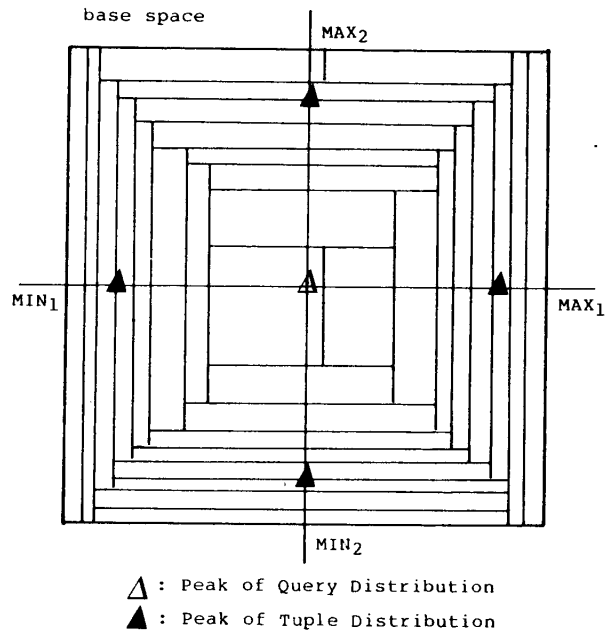


Fig.4 Example of Page Partitioning and GKD-tree (II)
(N = 32, k = 2)

4.2. Performance of GKD-tree Method

Here we evaluate the access performance of our algorithm. The query and tuple distributions examined here is the same as *Example* II, except that $k = 4$. For these distributions, the average number of page accesses is depicted in Fig.5. For comparison, the performance of KD-tree method is associated there. The figure shows that the average number of page accesses

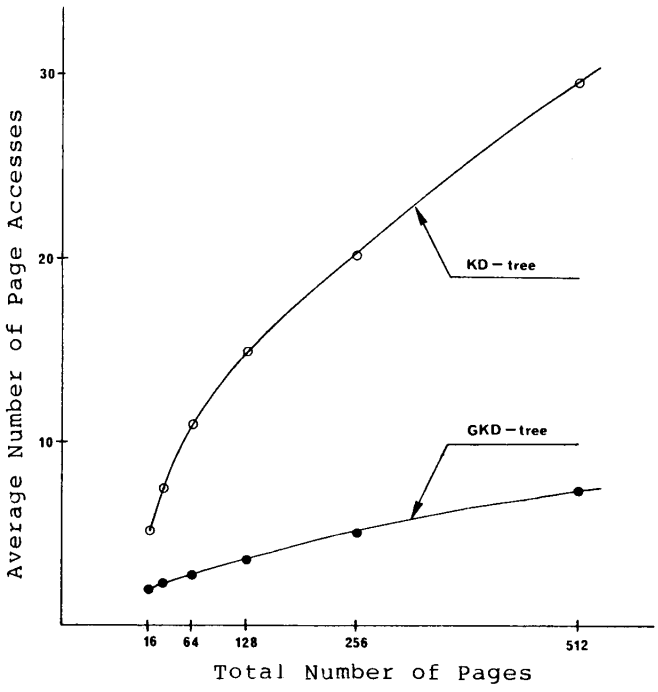


Fig.5 Performance Comparison between KD-tree Method and GKD-tree Method

is largely reduced by GKD-tree algorithm.

In these examples, the query distribution was biased in each domain of attributes. As an example of the query distribution in which the distributions are rather different among attributes, consider some attributes are referred to by the exact match predicates through a number of queries, while others are accessed by predicates of rather large range intervals. For such distribution, the GKD-tree algorithm partitions the space only along the axes of attributes in the former category. This is because, for such an attribute A_i, any discriminator value γ_{ij} gives zero to $H(p_j^{\Delta}(\gamma_{ij}))$, which means that no page accesses increase when this attribute is used in partitioning. Therefore, the average number of accesses is kept about one. Of course the classical access method can be applicable in this case, but it should be noted that GKD-tree method relieves the physical database designers of complicated work on access method selection. That is, if the classical methods is fully applicable, the GKD-tree method then automatically produces the same index and page partitioning as they do.

4.3. Discussion

We conducted a number of other experiments to get the data on multidimensional clustering by supplying various query and tuple distributions. Some of them showed the GKD-tree method was completely superior to KD-tree method, which is what we expected. On the other hand, we confronted several cases in which GKD-tree method was defeated by KD-tree method, which at first was our surprise. Such phenomena likely occur when (1) the number of attributes used by clustering is relatively small, and (2) the number of "mountain"s is also relatively small, but rather large. This is explained by examining the *integration effect* of the query distribution. As an example of this effect, suppose the query distribution be flat on Q. Then, what is the intuitive shape of the access pattern to D? The answer can be given by partitioning the base space into infinite number of small pages to have the average number of *tuple* accesses for each tuple. Define the $\rho(t)=H(v_1,v_1,\cdots,v_k,v_k)$ for tuple $t=(v_1,\cdots,v_k)$. Then we have:[16]

$$\rho(t) = \lim_{\epsilon_i \to 0} H(v_1, v_1+\epsilon_1, \cdots, v_k, v_k+\epsilon_k)$$

$$= \left(\prod_{i=1}^{k} \left(2/\mathrm{MAX}_i^2 \right) \right)$$

$$\times \left(\prod_{i=1}^{k} \left(- \left(v_i - \mathrm{MAX}_i/2 \right)^2 + \mathrm{MAX}_i^2/4 \right) \right)$$

(For simplicity, $\mathrm{MIN}_i = 0$ for all i is assumed.)

Thus, even if the query distribution is flat on Q, its "real" access pattern is of parabolic function multiplied k times. Its summit is at the center of the base space $(\mathrm{MAX}_1/2, \ldots, \mathrm{MAX}_k/2)$, and is to be most frequently accessed. Similarly, if we use linear distribution function on Q, the access pattern on D becomes $3 \cdot k$-dimensional polynomial function. Then, the skirts area of the "mountain" is too narrow to get the large performance gain by splitting the rarely accessed area. The skirts area is quickly consumed after a few partitionings are executed. Most area of the "mountain" is, so to speak, on the top of the "mountain". But in such area, there still exist the slight differences among the $H(p_j^{\Delta}(\gamma_{ij}))$'s for discriminator candidates, so the GKD-tree proceeds partitioning quietly. This is apparently disadvantageous. (The biased tuple distribution as in Fig.4 relaxes this.)

We then began to use the reduced precision when comparing values of $H(p_j^{\Delta}(\gamma_{ij}))$'s to avoid such too strict selection of discriminators. In this improved algorithm, $H(p_j^{\Delta}(\gamma_{ij}))$'s are compared each other by ignoring lower digits of values. Since these values for the top or area near top of the "mountain" are expected to be slightly different each other, this modification causes the effect that the area around the top of the "mountain" appears completely flat. Recall that the last statement of the algorithm in section 3.2 which specifies the procedure when multiple candidates give the minimum comparison value. Due to this, when it partitions the area which is very frequently accessed, GKD-tree algorithm with reduced precision comparison incorporated picks up the candidate which splits the space most

evenly. As a consequence, the algorithm behaves itself like the KD-tree method only when it partitions the area around the top of the "mountain".

5. Implementation Issues

Here, we discuss the implementation issues in the database machine environment.

5.1. Query Distribution

The biggest problem on the implementation is how to maintain query distribution. It is practically infeasible to keep all of the selection predicates issued by users since they would become too large. But notice that only the some portion of distribution is required for the algorithm to work effectively: it would work well even if the distribution for the queries which are rarely issued is not accurate. This is because the average number of page accesses will be hardly affected by how to split the space to which queries rarely access. We have already developed the algorithm called *gradated statistics* to realize this strategy effectively. In this technique, the query distribution is represented by selection predicates stored in the table of fixed size. One entry of the table consists of $2k$ fields to represent ranges along with the count field which gives how many times the predicate issued. Hence the query distribution could be represented as the relation scheme $Q(x_1, y_1, \cdots, x_k, y_k, count)$. When the storage is overflowed with a lot of incoming predicates, the *gradation* begins: some entry is merged with the other by making lower bits of some attributes "don't care" so that the resulted entry has the smallest value in its count field among entries. For example, suppose k be 2, and now the table is overflowed. If summation of count fields of two queries $q_1=[12,15]\times[0,4]$ and $q_2=[12,14]\times[1,7]$ is minimum among other possible candidates, they are merged into the *gradated* query $[12,15]\times[1,7]$ by making the following bits "don't care": the least significant bit for higher value of range predicate for first attribute, the least significant bit for lower value of range predicate for second attribute, and the least significant two bits for higher value of range predicate for second attribute. (It is assumed that other entries cannot be merged by this operation. More than two entries are possibley merged by one gradation operation.) The number of "don't care" bits are associated for each entry. By this algorithm, the distribution for queries rarely issued is "gradated", while that for queries frequently entered is kept still accurate. The value of $H(p_j^{\Delta}(\gamma_{ij}))$ can be computed by summing up the count fields of entries which satisfy the condition (4) with gradation taken into account. That is, this value can be given by issuing the *page* $p_j^{\Delta}(\gamma_{ij})$ *as the query to* Q: "SELECT SUM(*count*) FROM Q WHERE condition (4) holds with gradation taking into account". We use as the access path to this storage, the multidimensional trie[11,12] because of its best affinity to the gradation. The multidimensional trie uses bits from most significant bit to partition the space, while gradation consumes bits from the least significant one. It also contributes to reduce the computational time of gradation operations. In GRACE, the gradated statistics is stored in the semiconductor memory in the dedicated disk module along with its access path.

5.2. Discriminator Candidate Selection

Another problem on the implementation is how to compute the candidates for discrimination values. This requires the sorting of a lot of tuples. Our database machine environment resolves this since sorting can be efficiently carried out by using hardware sorter[3] and relatively large semiconductor memory in the disk module in GRACE[1,2].

5.3. Index Search

Recall that GKD-tree generated by the algorithm is not always balanced. But it is not the serious problem: First, we notice that the traverse of the tree can often be omitted by making use of the property mentioned in the previous section. That is, the shape of the GKD-tree produced by our algorithm perfectly reflects the undulation of the query distribution. Hence, in GKD-tree, it is assured that the page pointed by the leaf node at the tip of some bulge in the tree is very frequently accessed. Then, in many cases, the desired page can be found by checking leaf nodes at rather lower level. In addition, we can again use the the semiconductor memory employed in the disk module in the database machine environment, and put the whole index into it. This makes it unnecessary to make the index balanced in principle.

5.4. Insertion/deletion Handling

As mentioned previously, we can use the local page split/merge technique when the insertion/deletion operation cause page overflow/underflow. The page underflow can be handled in a simple way. First the empty page causing underflow is deallocated. Then the leaf node pointing to this page and its parent node are removed. Last, the grandparent node is modified to point to the brother of the removed leaf node instead of its parent node. In case of page overflow, we cannot apply the algorithm in section 3.2 to split the page, since the space to be partitioned contains only V + 1 tuples. Hence the algorithm for overflow manipulation only considers the selection of discriminator attribute. The discriminator value is determined so that it divides the tuples evenly to keep the load factor high. Once discriminator pair is selected, then a new node and one page are allocated and space is partitioned by the pair. Then, the pair is stored in the new node and pointers are arranged. Last, the tuples in either of subspaces are moved to the new page.

If the insertion/deletion of tuples occur frequently, we cannot keep the GKD-tree best fitted for the distribution of selection predicates. Since how to partition tuples which are frequently accessed dominates the average number of accesses, it is the good heuristics

that when such a page causes the page overflow/underflow, we perform re-clustering of tuples in the page and nearby pages altogether.

6. Conclusion

In this paper, the new multidimensional clustering algorithm is proposed based on the theoretical analysis of the formula which expresses the average number of page accesses. The algorithm can fully exploit the distribution information of tuples and queries. It is shown that the algorithm can largely reduce the average number of page accesses in many situations. Furthermore, the algorithm is modified to behave itself as if it were KD-tree method only when it partitions the area frequently accessed. Implementation issues are discussed and gradated statistics technique is proposed which maintains the query distribution efficiently.

We can further improve the performance by integrating our clustering algorithm with *transposing* of relations[14], i.e., *projective partitioning of the base space*. It can be integrated with the transpose method as follows. First, the *attributes* are partitioned so that joins between the retrieved subtuples using *Tuple ID's* are less required. Then the GKD-tree algorithm is applied to all the subrelations. The first step of this method requires extended query model in which queries refer to multiple relations in general[17]. This directly leads to the modification of the cost function (3) so that it should take join cost into account. Discussion on this issue in the database machine environment can be found in [16].

Acknowledgement: The authors wish to thank for the thoughtful comments from referees.

[References]

[1] Kitsuregawa,M.,et. al. "*Relational Algebra Machine GRACE*", Lecture Notes in Computer Science 147, Springer-Verlag, pp.191-214 (1982).

[2] Kitsuregawa,M.,et. al. "*Application of Hash to Data Base Machine and Its Architecture*", New Generation Computing, 1, Springer-Verlag, pp.63-74, (1983).

[3] Kitsuregawa,M and Fushimi,S.,et. al. "*The Organization of Pipeline Merge Sorter*", Trans. of IECE Japan, J66-D, pp.332-339 (1983).

[4] Bentley,J.L., "*Multidimensional Binary Search Trees Used for Associative Searching*", CACM, Vol. 18, No. 9, pp. 509-517 (1975).

[5] Bentley,J.L., "*Multidimensional Binary Search Trees in Database Applications*", Trans. on SE, Vol. SE-5, No. 4, pp.333-340 (1979).

[6] Bentley,J.L., "*Data Structures for Range Searching*" Computing Surveys, Vol. 11, No. 4, pp.397-409 (1979).

[7] Duhne-Aguayo,R.A., "*Optimal Design of a Generalized File Organization*" Ph.D Thesis, Cornell Univ. (1977).

[8] Chang,J.M., and Fu,K.S., "*Extended K-D Tree Database Organization: A Dynamic Multi-Attribute Clustering Method*", Proc. of VLDB, pp.39-43 (1979).

[9] Robinson,J.T., "*The K-D-B-Tree: A Search Structure for Large Multidimensional Dynamic Indexes*", Proc. of ACM SIGMOD Conf., pp.10-18 (1981).

[10] Ouksel,M. and Scheuermann,P. "*Multidimensional B-Trees: Analysis of Dynamic Behavior*", BIT 21, pp.401-418 (1981).

[11] Orenstein,J.A., "*Multidimensional Tries Used for Associative Searching*" Information Processing Letters, Vol. 14, No. 4, pp. 150-157 (1981).

[12] Tanaka,Y., "*Adaptive Segmentation Schemes for Large Relational Database Machines*", in Database Machines, Leilich, H.-O. and Missikoff, M. ed. Springer-Verlag (1983).

[13] Lee,D.T., "*Worst-Case Analysis for Region and Partial Region Searches in Multidimensional Binary Search Trees and Balanced Quad Trees*", Acta Informatica 9, pp.23-29 (1977).

[14] Batory,D.S., "*On Searching Transposed Files*", ACM TODS, Vol.4, No.4, pp.531-544 (1979).

[15] Fushimi,S., "*Multidimensional Clustering Technique for Large Relational Database Machines*", to appear in Proc. of International Conference on Foundations of Data Organization, Kyoto, Japan (1985).

[16] Fushimi,S., "*Design of Secondary Storage System for Relational Database Machine: Physical Database Organization Method Based on Multidimensional Clustering and Transposition*", Ms. Thesis, Univ. of Tokyo (1983) (In Japanese)

[17] Whang,K.-Y., Wiederhold,G., and Sagalowicz,D., "*Separability - An Approach to Physical Database Design*", Trans. on Comuters, Vol.C-33, No.3, pp.209-222 (1984)

Appendix

A. The distribution functions used in *Example* I:

(1) All of domains of attributes are assumed to be real numbers, and $MAX_i = -MIN_i = 2.0$ for all i's.

(2) Query distribution function:

$$Q(q) = (1/V) \cdot \sum_{m=1}^{5} Q_m(x_1, y_1, x_2, y_2)$$

where V is the normalization factor, and:

$$Q_1(x_1, y_1, x_2, y_2) = 1.0 \cdot (y_1 - x_1) \cdot (y_2 - x_2) \quad \begin{pmatrix} -0.8 \leqq x_1 \leqq y_1 \leqq 0.8 \\ -0.6 \leqq x_2 \leqq y_2 \leqq 0.6 \end{pmatrix}$$

$$Q_2(x_1, y_1, x_2, y_2) = 4.0 \cdot (y_1 - x_1) \cdot (y_2 - x_2) \quad \begin{pmatrix} -1.3 \leqq x_1 \leqq y_1 \leqq -0.4 \\ -1.7 \leqq x_2 \leqq y_2 \leqq -0.4 \end{pmatrix}$$

$$Q_3(x_1, y_1, x_2, y_2) = 4.0 \cdot (y_1 - x_1) \cdot (y_2 - x_2) \quad \begin{pmatrix} 0.4 \leqq x_1 \leqq y_1 \leqq 1.8 \\ -1.7 \leqq x_2 \leqq y_2 \leqq -1.0 \end{pmatrix}$$

$$Q_4(x_1, y_1, x_2, y_2) = 1.0 \cdot (y_1 - x_1) \cdot (y_2 - x_2)$$

$$\begin{pmatrix} -1.8 \leqq x_1 \leqq y_1 \leqq 1.2 \\ -0.7 \leqq x_2 \leqq y_2 \leqq 1.7 \end{pmatrix}$$

$$Q_5(x_1, y_1, x_2, y_2)) = 4.0 \cdot (y_1 - x_1) \cdot (y_2 - x_2)$$

$$\begin{pmatrix} 0.9 \leqq x_1 \leqq y_1 \leqq 1.8 \\ 0.4 \leqq x_2 \leqq y_2 \leqq 1.9 \end{pmatrix}$$

(3) The tuple distribution is assumed to be uniform.

B. The distribution functions used in *Example* II:

(1) All of domains of attributes are assumed to be real numbers, and $MAX_i = -MIN_i = 2.0$ for all i's.

(2) Query distribution function:

$$Q(q) = (1/V) \cdot (100.0 - 30.0 \sum_{i=1}^{k} (x_i^2 + y_i^2)),$$

where V is the normalization factor.

(3) Tuple distribution function:

$$D(t) = \sum_{i=1}^{k} (100.0 - 90.0 \cdot (\sum_{j \neq i}^{k} v_j^2 + (v_i - 1.6)^2) - 90.0 \cdot (\sum_{j \neq i}^{k} v_j^2 + (v_i + 1.6)^2))$$