TBSAM: An Access Method for Efficient Processing of Statistical Queries

JAIDEEP SRIVASTAVA, MEMBER, IEEE, JACK S. EDDY TAN, AND VINCENT Y. LUM, SENIOR MEMBER, IEEE

Abstract—**Most research in database access methods has been aimed at providing efficient support for business data processing applications. New database applications such as Computer Aided Design (CAD), Computer Aided Manufacturing (CAM), and Computer Vision, have demonstrated the unsuitability of the popular access methods. This paper targets the domain of Statistical and Scientific Databases, and considers the class of aggregate queries, which are very often encountered in this domain. Such a query is aimed at retrieving some aggregate characteristics of the raw data. In this paper, we present TBSAM, an access method that provides support for the efficient processing of aggregate queries. It is related to the B^+-tree, and also possesses the latter's efficient update properties. Complementing TBSAM is the provision of a grouped update algorithm for minimizing expensive indexed database updates.**

Index Terms—**Aggregate queries, database access methods, descriptive and order statistics, grouped update algorithm, sampling, Statistical and Scientific Databases.**

I. Introduction

A NUMBER of access methods, along with their index structures and query processing algorithms, have been reported in the literature. Some of the classical ones are VSAM [1], B-tree [2], GRID-file [3], and Linear Hashing [4], while the newer ones include the BANG-File [5] and R-tree [6]. These access methods have significantly reduced the time required to access data from physical storage. However, almost all of these are aimed at speeding up the processing of relational operators, primarily the join operator. The main reason for this has been the success of the relational model in the field of business data processing.

The past few years have witnessed an increasing use of databases in new application areas. These include Computer Aided Design (CAD), Computer Aided Manufacturing (CAM), Computer Aided Software Engineering, Statistical and Scientific Analysis, Planning and Forecasting, Medical Diagnosis, and Molecular Biology. The inadequacy of classical access methods for these new applications has been demonstrated by the emergence and success of new techniques such as the R-tree for CAD databases and Quad-tree [7] for image databases.

Manuscript received March 1, 1989; revised August 1, 1989.

J. Srivastava and J. S. E. Tan are with the Department of Computer Science, University of Minnesota, Minneapolis, MN 55455.

V. Y. Lum is with the Computer Science Division, Naval Postgraduate School, Monterey, CA 93943.

IEEE Log Number 8931612.

These new kinds of database applications give rise to entirely new kinds of queries, whose representation by relational operators is, at the very best, cumbersome. One class of queries, which this paper addresses, is called *aggregate queries*. An example of an aggregate query is

Calculate *set-of-aggregates* of all data items

such that *boolean qualification*.

Here, aggregates are some overall characteristics of all the qualifying data items. Examples of such aggregates include descriptive and order statistics. Descriptive statistics are the moments of various orders, i.e., mean, variance, kurtosis, etc. Measures such as the median, quartiles, deciles, percentiles, etc., are called order statistics. The occurrence of such queries in business data processing is evidenced by the existence of *count*, *sum*, and *average* operations in most SQL implementations. Even the relational model has been extended to include aggregates [8]. However, the occurrence is infrequent, and hardly any attention has been paid to designing specialized access methods for their efficient processing. Notable exceptions include the work by Ghosh [9] and Olken and Rotem [18].

This class of queries arises very naturally in applications such as Scientific Data Analysis, Planning and Forecasting, Marketing Research, and Actuarial Studies. As has been pointed out by Shoshani [11] and Olken *et al.* [18], a successful implementation of large-scale Statistical and Scientific Databases (SSDBS's) requires the capability for efficient processing of aggregate queries, among others. As we shall show in this paper, the cost of processing aggregate queries with the very popular B^+-tree is prohibitive.

In this paper, we present a Tree Based Statistics Access Method (TBSAM), that is designed to efficiently process a class of aggregate queries. TBSAM is based on the B^+-tree [12], and it exploits all the benefits of a B^+-tree's dynamic nature. It provides facilities for efficient evaluation of the arithmetic mean and higher moments of one or more attributes. The B^+-tree index structure provides an ordering of the tuples of a relation on some attribute (the index attribute). The aim is the efficient retrieval of a tuple, given the value of its index attribute. However, there is no proviso for retrieving a tuple whose *rank* in the order is specified instead of its index

attribute value. This is the basic operation required in finding the median and other order statistics for a set of data items. This operation is supported in a natural and efficient manner by TBSAM. We show how it can be used for performing statistical sampling on a relational database.

TBSAM is a dynamic index, and thus can support insertion/deletion/modification of tuples in a relation. We show that these operations can be performed very naturally, and the cost is almost the same as that for the B^+-tree. Overhead for updates to a relation can be expensive, especially if the update rate is high. A grouped algorithm for installing updates in a batched manner is presented and analyzed. The problem of batched updates on tree indexes has been studied by a number of researchers. Schneiderman and Goodman [13] and Palvia [14] presented approximate average case analyses for batched updates on the B-tree, while Piwowarski [15] has presented an exact average case analysis. Batory and Gotlieb [16] have presented an approximate average case analysis for the B^+-tree. We have carried out an exact worst case analysis for the B^+-tree. To our knowledge, an exact average case analysis of the B^+-tree is still an open problem.

Section II of this paper describes the structure of the TBSAM index, and Section III presents the query processing algorithms. Section IV analyzes the query processing costs and compares them to those of the B^+-tree. Section V discusses basic algorithms for dynamic index maintenance to handle updates, while Section VI presents and analyzes an efficient grouped update algorithm. Conclusions are presented in Section VII.

II. The Index Structure of TBSAM

This section gives a detailed description of the index structure of TBSAM, an access method similar to the B^+-tree [12]. Each nonroot node of TBSAM of order m contains k keys, where $\lceil m/2 \rceil \leq k \leq m$. In addition, it also contains $k + 1$ pointers P_j, $0 \leq j \leq k$, that point to its subtrees. P_i points to the root of the subtree containing all keys K_j such that $K_i \leq K_j < K_{i+1}$. The leaf nodes point to data pages that contain tuples of the relation being indexed by the tree.

Fig. 1 shows a TBSAM index created on the key attribute K. The index provides an ordering of tuples on the key attribute. If the index is clustered, the physical storage of tuples is also in key order. Our scheme is applicable to both clustered and unclustered indexes.

Each node of the TBSAM index contains some information in addition to the keys and pointers. Fig. 2 shows the structure of a nonleaf node of the TBSAM index. The three types of fields are as follows:

- K_i, $1 \leq i \leq k$ The boundary key between the $(i-1)$th and ith partition of the key space.
- P_i, $0 \leq i \leq k$ Pointer to the root of the ith subtree.
- S_i, $0 \leq i \leq k$ Statistics metadata for the ith subtree.

The structure of each S_i is shown in Fig. 3, where *count* is the number of leaf nodes in the subtree pointed to by

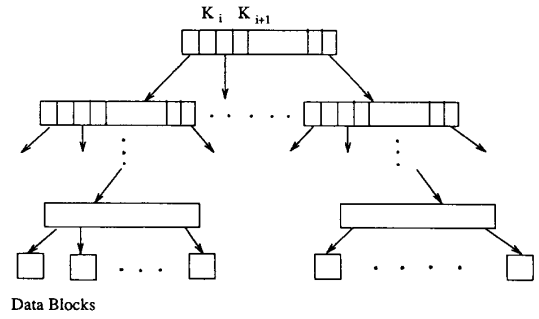


Fig. 1. TBSAM index tree.

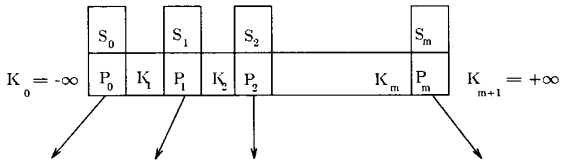


Fig. 2. TBSAM index node. S_i has statistical information about subtree pointed to by P_i.

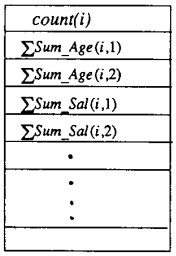


Fig. 3. Data structure of statistics metadata S_i.

P_i. Statistics metadata describing different moments are stored in subsequent fields.

Statistics Metadata: The term *metadata* means *data about data*. Statistics metadata [9] is metadata that facilitates the processing of statistical queries. Consider the employee relation with some numeric attributes, e.g., age, height, and salary. Further, assume that a large number of statistical queries are range queries, with range specification on age. These queries are of the following form:

SELECT < *set_of_statistics* >
FROM *EMPLOYEE*
WHERE *Min_Age* ≤ *Emp.Age* ≤ *Max_Age*
< *set_of_statistics* >: univariate descriptive statistics.

A TBSAM index structure, using age as the key attribute, will facilitate the processing of such queries. Assume that the mean and variance of age and salary are usually required. As shown in Fig. 3, the statistics metadata stored in the index node are *count* and partial sums of various orders for different attributes. Here, *count*(i) is the number of leaf nodes in the ith subtree of the current node, i.e.,

$$count(i) \equiv \textit{number of data tuples in ith subtree},$$
$$0 \leq i \leq m.$$

Similarly, for *Sum_Age*, *Sum_Sal*, etc., we have

$$Sum\_Age(i, p) = \sum_{\text{all tuples in ith subtree}} (Tuple.Age)^p,$$
$$0 \le i \le m, p = 1, 2.$$

The tree has been created on the age attribute, i.e., a left to right scan of the leaf nodes of the index corresponds to listing the tuples by nondecreasing age. This property of the tree can be used to efficiently process queries requiring order statistics, i.e., median, quantiles, percentiles, etc., on the age attribute.

Aggregate Property: Our choice for the statistics metadata above is not arbitrary. It is based on the *aggregate property* which all of the chosen quantities exhibit. A metadata S_i, associated with an index node p, satisfies the aggregate property if

$$S_i \equiv \sum S_j, \quad \text{where } S_j \in \{\textit{metadata in ith child of } p\}.$$

Count and *Sum* are examples of metadata that exhibit the aggregate property, while *Mean* and *Variance* are ones that do not.

Simply stated, the cumulative statistical results with any aggregate property stored in a node p's statistical metadata S_i reflects the sum of all of p's child nodes' metadata. This aggregate property recursively holds true for the child nodes of p's child nodes, and so on.

III. Query Processing Using TBSAM

This section shows how the processing of various types of statistical queries can be facilitated by the use of the TBSAM index. Discussed below are the techniques for computing descriptive and order statistics, as well as creating samples.

A. Descriptive Statistics

The metadata stored in the statistics field of the index node depends on the moments required by the application. Since there is potentially an unbounded number of moments that can be requested, it is practically impossible to store precomputed information to support all of them. Our approach is to let the user specify the metadata required, and store exactly those. If other moments are requested, they have to be computed from the basic data. New metadata can be added at any time while old ones can be dropped. However, these operations are expensive, since traversal of the whole index may be required. Consider the example query from the previous section, with TBSAM index on the *Age* attribute of the *EMPLOYEE* relation. We have

SELECT < *set-of-statistics* >
FROM *EMPLOYEE*
WHERE *Min_Age* ≤ *Emp.Age* ≤ *Max_Age*.

Here, < *set-of-statistics* >: univariate descriptive statistics.

Let the first moments of *Age* and *Salary*, i.e., μ_{Age} and μ_{Sal}, be usually required. The statistics metadata to be maintained at any index node are

$$Sum\_Age[i, 1] \equiv \sum_{j \in CHILD[i]} Sum\_Age[j, 1]; \quad 0 \le i \le k$$
$$Sum\_Sal[i, 1] \equiv \sum_{j \in CHILD[i]} Sum\_Sal[j, 1]; \quad 0 \le i \le k$$
$$count[i] \equiv \sum_{j \in CHILD[i]} count[j]; \quad 0 \le i \le k$$

where $CHILD[i]$ is the ith child of the node.

Thus, each index node stores statistics metadata for its children. Without loss of generality, to simplify the query processing algorithms, we assume that all numeric metadata for nonexistent children is zero. The qualifier *Min_Age* ≤ *EMP.Age* ≤ *Max_Age* is a *range restriction* of the tuples on the *Age* attribute. The algorithm shown in Fig. 4 calculates the first moment, i.e., the arithmetic mean, of the attribute *Attr*. Other descriptive statistics of the tuples specified in the range can be obtained if the appropriate metadata are collected. The algorithm starts with the metadata for the entire index tree, and trims out parts of it as it descends each level, looking for exact matches for the keys *Min_Age* and *Max_Age*.

Example: Consider the query

SELECT μ_{Age}, μ_{Sal}
FROM *EMPLOYEE*
WHERE 20 ≤ *Emp.Age* ≤ 49.

The hatched lines in Fig. 5 show the path taken to reach the leaf nodes containing the tuples with the *Age* attribute between 20 and 49. *TotalSum_Age* [1], *TotalSum_Sal* [1], and *TotalCount* dynamically maintain the metadata corresponding to the successive refinements of the range. Thus, we have

$$\begin{aligned} TotalSum\_Age[1] &= (208 + 161) - (0 + 0) \\ &\quad - (34 + 61) - (0 + 50) = 224 \\ TotalSum\_Sal[1] &= (225.3 + 139.8) - (0.0 + 0.0) \\ &\quad - (24.0 + 48.9) - (0.0 + 51.6) \\ &= 270.6 \\ TotalCount &= (8 + 3) - (0 + 0) - (2 + 1) \\ &\quad - (0 + 1) = 7. \end{aligned}$$

Thus,

$$\mu_{Age} = TotalSum\_Age[1]/TotalCount = 224/7 = 32$$
$$\mu_{Sal} = TotalSum\_Sal[1]/TotalCount = 270.6/7 = 38.66$$

Other Descriptive Statistics: The example above shows how the arithmetic means, μ_{Age} and μ_{Sal}, for attributes *Age* and *Salary*, can be calculated efficiently. It is important to note that any kind of descriptive statistics of tuples in

Procedure Descriptive_Statistics

begin
Min_Node_Ptr := Root ;
Max_Node_Ptr := Root ;

$Total\_Sum\_Attr := \sum_{l=0}^{m} Root.Sum\_Attr[l,1];$

$Total\_Count := \sum_{l=0}^{m} Root.Count[l,1];$

repeat
for Min_Node ; select i such that $K_i \leq Min\_Key < K_{i+1}$;
for Max_Node ; select j such that $K_j \leq Max\_Key < K_{j+1}$;

$Total\_Sum\_Attr := Total\_Sum\_Attr - \sum_{l=0}^{i-1} Min\_node.Sum\_Attr[l,1] - \sum_{l=j+1}^{m} Max\_Node.Sum\_Attr[l,1];$

$Total\_Count := Total\_Count - \sum_{l=0}^{i-1} Min\_node.Count[l,1] - \sum_{l=j+1}^{m} Max\_Node.Count[l,1];$

$Min\_Node := Min\_Node.Ptr[i];$
$Max\_Node := Max\_Node.Ptr[j];$
until Node.type = data ;

$\mu_{Attr} = \frac{Total\_Sum\_Attr}{Total\_Count};$
end /* descriptive statistics */

Fig. 4. Descriptive statistics algorithm.

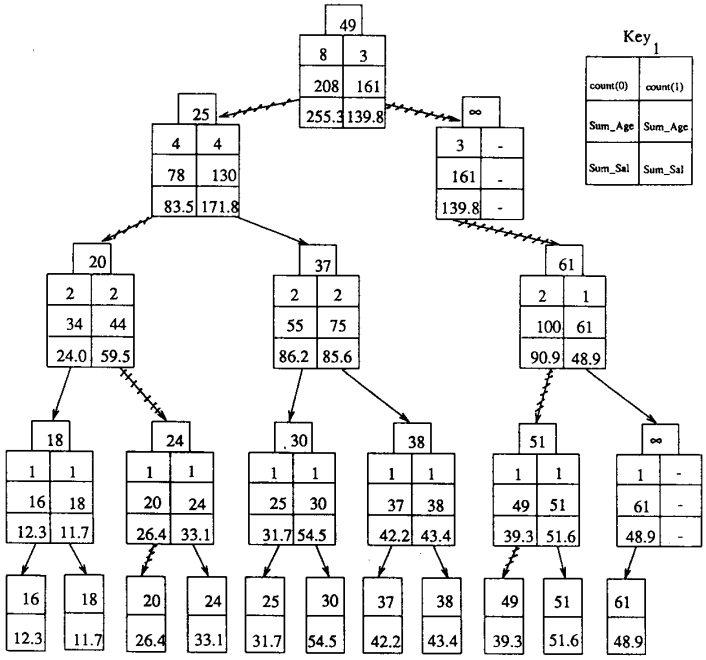


Fig. 5. Calculation of descriptive statistics using TBSAM.

the range [*Min_Age*, *Max_Age*], i.e., higher order moments of *Age* and *Salary*, and moments of other attributes, can be calculated at no additional cost.

B. Order Statistics

These are based on some ordering of the tuples in a relation, and their efficient processing requires the ability to access the tuple with a specified rank in a rapid manner. This is provided by TBSAM very naturally using the *count* metadata. The algorithm is shown in Fig. 6.

Assume that the rth order statistic, i.e., the tuple with *absolute rank* r, is to be retrieved. The idea is to traverse down the index tree, searching for the *relative rank* $r - t$ in the subtree which contains the absolute ranks

```
Procedure Order_Statistics

begin
  Node = Root ;
  repeat

    find i such that Σ_{j=0}^{i-1} Node.Count[j] < Rank ≤ Σ_{j=0}^{i} Node.Count[j];
    Rank := Rank - Σ_{j=0}^{i-1} Node.Count[j];
    Node := Node[i].Ptr ;

  until Node.type = data ;
end /* Order Statistics */
```

Fig. 6. Order statistics algorithm.

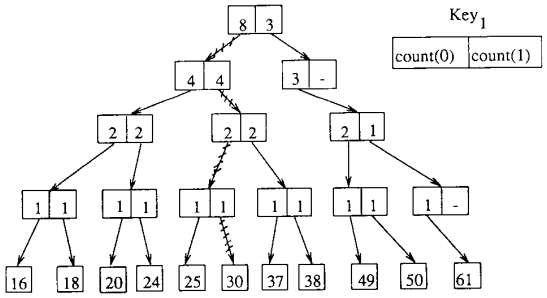


Fig. 7. Evaluation of order statistics.

$t+1; t+2, \cdots, r-1, r, r+1, \cdots$. Accessing the median, the first quartile, and the first decile entails accessing the tuples with absolute ranks $\lceil N/2 \rceil$, $\lceil N/4 \rceil$, and $\lceil N/10 \rceil$, respectively. Fig. 7 shows the retrieval of the sixth ranked tuple using the TBSAM index of Fig. 5. For simplicity, only the *count* metadata is shown.

C. Sampling

Performing statistical calculations directly on large populations is prohibitively expensive [18]. To circumvent this, statisticians often perform studies on a *sample* of the population and extrapolate the results to the entire population. This requires creating a sample of the population, a process called *sampling*. Cochran [17] discusses in detail the various sampling techniques and their statistical significance. Some of the commonly used sampling techniques are Random Sampling, Stratified Sampling, and Systematic Sampling. The following discussion shows how these can be performed efficiently using the TBSAM index structure.

Random Sampling: This sampling technique requires some ordering of tuples, which is provided by TBSAM for the key attribute. A random number is drawn, and its value is used as the rank of the next tuple to be selected. This process is carried out until the required sample is obtained. Since accessing a tuple with a specified rank is the basic operation, the TBSAM structure facilitates efficient random sampling in a very natural manner. For example, suppose a four-figure random number 4367 is chosen. The tree index is traversed to find the 4367th ranked tuple.

Stratified Sampling: Quantiles, Deciles, Percentiles, etc., are a natural stratification (partition) of the tuples in a relation. Random sampling is performed within a strata to generate a stratified sample. Supposing a sample of size 1000 has to be drawn, with the data divided into 100 strata. A simple way to obtain the sample is to use TBSAM to select random samples of size 10 from the portion of the relation between successive percentiles.

Systematic Sampling: Systematic sampling involves choosing tuples periodically from the entire relation. An example would be to choose the mth, $(m+i)$th, $(m+2i)$th, $\cdots$, $(m+999i)$th tuples, if a sample of size 1000 is required. Here too, the basic operation is accessing a tuple of a given rank. Thus, the TBSAM structure once again is useful.

Statistical sampling from relational databases has been discussed in detail by Olken [10]. However, the approach taken is quite different. The query language has to be extended by adding a set of sampling operators. In this context, TBSAM can be looked upon as an efficient implementation of these sampling primitives. Ghosh [9] has also proposed algorithms for obtaining samples on a large database. His approach is similar in nature to the one proposed here.

IV. Cost of Query Processing

The unit of cost considered is the *number of disk accesses*. Since CPU computation is cheap relative to disk accesses, operations like *addition, subtraction*, and *key comparison* do not contribute significantly to query processing cost. The costs are different for clustered and unclustered TBSAM indexes due to the differences in their heights, as shown in Fig. 8. For clarity in comparison, both clustered and unclustered index trees are shown as symmetrically sharing the same data blocks.

For notational clarity, the following are defined.

N	Number of data blocks in the relation.
d	Average fanout, i.e., number of children of an index node; also the blocking factor, i.e., number of tuples in a data block.
Nd	Number of tuples in the relation.
N/d	Number of blocks in the lowest level of a clustered index tree.
N	Number of blocks in the lowest level of an unclustered index tree.
$\lceil \log_d N/d \rceil$	Number of index blocks accessed in a root-leaf path traversal for a clustered index.
$\lceil \log_d N \rceil$	Number of index blocks accessed in a root-leaf path traversal for an unclustered index.
$d \approx 0.7m$	B^+-trees (and TBSAM) index nodes have an average fill factor of approximately 70 percent [19].
S	Required sample size for the sampling operations.

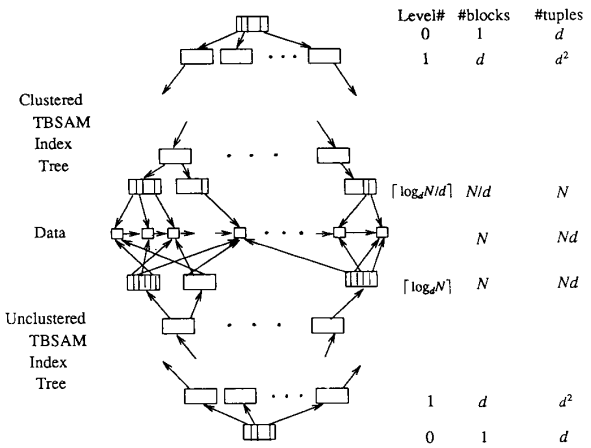


Fig. 8. Clustered versus unclustered TBSAM index tree.

Max_Dom	Maximum key value of the domain.
Min_Dom	Minimum key value of the domain.
B	Expected number of tuples in the range [*Min_Key, Max_Key*], i.e.,

$$B = Nd. \frac{(Max\_Key - Min\_Key)}{(Max\_Dom - Min\_Dom)}$$

Descriptive Statistics: Collection of metadata for answering a range query involves precisely two *root-leaf* path traversals, once each for *Min_Key* and *Max_Key*, of the canonical query considered in Section III. Thus, the cost of index traversals for a clustered TBSAM index is $2\lceil \log_d (N/d) \rceil$. In addition, two data blocks have to be fetched, making the total cost $2\lceil \log_d N \rceil$.

It is important to note that exactly two root-leaf traversals are sufficient to calculate all required descriptive statistics, for one or more attributes, for a range specified on the key attribute. Thus, the cost of calculating *one or more* descriptive statistics is $2\lceil \log_d N \rceil$. The cost for an unclustered index is $2(\lceil \log_d N \rceil + 1)$.

Order Statistics: To calculate the rth order statistic on the key attribute *Key_Attr*, a tuple T_r has to be found such that there are $r - 1$ tuples T_i, whose *Key_Attr* value is less than $T_i.Key\_Attr$. Since the index is created on a sorted order of keys, this can be achieved by a *root-leaf* path traversal. The cost of index traversal and fetching the data block is $\lceil \log_d N/d \rceil + 1 = \lceil \log_d N \rceil$ for clustered and $\lceil \log_d N \rceil + 1$ for unclustered TBSAM indexes.

Sampling: As discussed in the previous section, the problem of finding a sample of size s can be reduced to finding s different order statistics. Thus, the cost of sampling is proportional to the size of the required sample. Hence, for a sample of size s, it is $s\lceil \log_d N \rceil$ for clustered and $s(\lceil \log_d N \rceil + 1)$ for unclustered indexes. From the above expression, sampling seems to be an expensive operation, especially if s is large. However, this estimate is overly pessimistic. In reality, the cost is substantially less due to the effects of buffering. Modeling buffering effects is beyond the scope of this paper, but is an important research direction for better cost estimates. The following optimization, however, can reduce the cost substantially, without even considering buffering.

First, generate all s random numbers and sort them. Next, retrieve data blocks using the TBSAM index, starting with the smallest random number as key. From each data block, retrieve all tuples whose key values match the random numbers. Let S_{avg} be the average number of samples obtained from each data block. Thus, the respective costs of retrieving a sample of size s for clustered and unclustered indexes are

$$\frac{s}{S_{avg}} \lceil \log_d N \rceil \quad \text{and} \quad \frac{s}{S_{avg}} (\lceil \log_d N \rceil + 1).$$

In general, this method of sampling is useful only for small sample sizes. For large sample sizes, S is large and scanning the entire database may be more efficient. The correct decision can only be made when the query is posed, and is best done by the optimizer.

Comparison to B⁺-Trees: All of the statistical operations described above can be performed using B⁺-trees. However, the cost can be quite substantial, making such queries *hard commands*.

The cost of performing the range query of Section III will be $\lceil \log_d N/d \rceil$ for the index file and $\lceil B/d \rceil$ for the data file, if the index is clustered and the data file is chained. If the index is unclustered, the cost calculation is more involved. From Yao [25], we know that the expected number of data blocks accessed in retrieving B records from a file containing Nd records, with d records per data block, is the Yao function given by

$$Y(Nd, N, B) = \frac{(Nd - d)!(Nd - B)!}{(Nd)!(Nd - d - B)!}.$$

It requires $\lceil \log_d N \rceil + 1$ disk accesses to retrieve each block (going through the B⁺-tree index). Thus, the total cost of retrieval is $Y(Nd\ N, B)(\lceil \log_d N \rceil + 1)$. However, it might be cheaper to do a sequential scan of the entire file, accessing N blocks. Thus, the cost is

$$\min\big(Y(Nd, N, B)(\lceil \log_d N \rceil + 1), N\big).$$

The B⁺-tree is well-suited for the retrieval of a tuple, given the value of its index attribute, i.e., the key value. However, there is no proviso for retrieving a tuple whose *rank* in the order is specified. Finding the median requires scanning the data blocks of the file until the one containing the median is found. For the clustering attribute, this requires $\lceil N/2 \rceil$ disk accesses since the file is sorted. For an unclustered attribute, some kind of median finding algorithm has to be run. Since this requires reading all the data blocks at least once, the cost is $O(N)$ disk accesses. Similarly, the costs for other order statistics can be derived. Since the cost of sampling is intimately tied up with that of finding a tuple given its rank, these costs can be derived analogously.

The costs of various statistical operations for the B⁺-tree and TBSAM are summarized in Table I.

TABLE I

Comparison of B$^+$-Trees and TBSAM

	Clustered		Unclustered	
Operation	B^+ tree	TBSAM	B^+ tree	TBSAM
Descriptive	$\lceil \log_d N/d \rceil + \lceil B/d \rceil$	$2\lceil \log_d N \rceil$	$min(Y(Nd, N, B)(\lceil \log_d N \rceil + 1), N)$	$2(\lceil \log_d N \rceil + 1)$
Order	$N/2$	$\lceil \log_d N \rceil$	$O(N)$	$\lceil \log_d N \rceil + 1$
Sampling	$min(\frac{S}{S_{avg}}(N/2), N)$	$\frac{S}{S_{avg}}\lceil \log_d N \rceil$	$min(\frac{S}{S_{avg}}N, N \log N)$	$\frac{S}{S_{avg}}(\lceil \log_d N \rceil + 1)$

V. Maintenance of a TBSAM Index

Generally, the most expensive operation an access method has to perform is handling updates, i.e., insertion and deletion. Thus, the usefulness of any access method for a dynamic database depends critically on its ability to perform these operations. One of the reasons why B$^+$-trees have been so popular is their capability of handling updates with a cost proportional to $\lceil \log_d N \rceil$ disk accesses. In this section, we show that the cost of handling updates in TBSAM is comparable to that for B$^+$-trees.

The insertion of a new tuple in a relation with a TBSAM index involves traversing a root-leaf path to the appropriate data block. The tuple is then inserted in its proper place. The extra work that is required compared to a B$^+$-tree is the appropriate modification of the statistics metadata of all the index nodes encountered. Since the aggregate property is satisfied, the values of the new tuple simply need to be added to the existing metadata. The deletion of a tuple can be done in an analogous manner, with the exception that metadata modification is just simple subtraction.

Insertion/deletion of tuples may cause index nodes to split/merge in a B$^+$-tree index. This is handled by redistributing keys and subtree pointers in nodes. A similar scheme can be used for the TBSAM index structure, where in addition to keys and subtree pointers, the appropriate statistics metadata needs to be redistributed [12]. Since each index node stores metadata for each of its subtrees *separately* and the statistics metadata satisfies the *aggregate property*, this redistribution is straightforward. Fig. 9 shows the algorithm for insertion/deletion of tuples.

Fig. 10 shows a node split for a TBSAM index node. The original node has keys K_i, $1 \le i \le m$, pointers P_i, $0 \le i \le m$, and statistics metadata S_i, $0 \le i \le m$. The node splits into two, with the left part receiving the keys K_i, $1 \le i < m/2$, the pointers P_i, $0 \le i < m/2$, and statistics metadata S_i, $0 \le i < m/2$, while the right part gets the rest.

Cost of Updates: The cost of inserting/deleting a tuple in the TBSAM index is closely linked to that of the B$^+$-tree. In each case, the dominant cost is attributed to the root-leaf index traversal. Thus, the cost of index update is $\lceil \log_d N \rceil$ ($\lceil \log_d N \rceil + 1$) disk accesses for clustered (unclustered) TBSAM indexes. If a B$^+$-tree index node overflows/underflows, a split/merge takes place, and the cost of an update is higher. The TBSAM index tree is also affected in a similar manner. However, as reported in [19], the probability of an index node split is very small.

```
Procedure Insert-Delete

begin
  Node = Root ;
  repeat
    find i such that K_{i-1} ≤ Test_Key < K_i;
    case <op> of
      insert : add values to Stats[i] ;
      delete : subtract values from Stats[i] ;
    end /* case */
      Node := Node.Ptr[i] ;
  until Node.type = data ;
end /* Insert-Delete */;
```

Fig. 9. Algorithm for insertion and deletion.

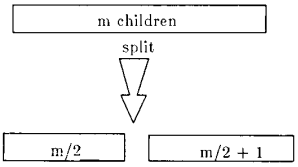


Fig. 10. Node split during insertion.

This observation shows that the propagation of a node splitting up the TBSAM index is an extremely rare occurrence just like B$^+$-trees. The theoretical worst case occurs when the insertion of a tuple with a new key value causes splits at all levels of the index. The worst case number of splits can be $O(\log_d N)$.

VI. Deferred Index Maintenance

The discussion in this section assumes a clustered index. Cost expressions for an unclustered index can be derived analogously. As shown in the previous section, the cost of installing an update using TBSAM is $\lceil \log_d N \rceil$. For installing n updates, the number of disk accesses required can be $n \lceil \log_d N \rceil$. For large values of N, this cost can be substantial, especially if the update rate is high. However it can be reduced if the updates are installed in a *deferred* manner. As updates arrive, they are stored in a temporary data structure instead of being installed immediately. Periodically, the collected updates are installed in the database in a *grouped* manner, and the temporary structure is purged.

This approach is discussed in detail for the TBSAM index structure. A grouped update algorithm is presented and analyzed. An expression for the savings gained by this approach is derived. If deferred maintenance is used, the database may not be up-to-date at all times. This section also discusses the required modifications to the query processing algorithm.

A. The Basic Algorithm (BA)

The problem is to install n updates, contained in a list L, in a database with TBSAM index T. Fig. 11 shows the basic approach, called the *Basic Algorithm (BA)*. Each tuple, *tup* in the list L is installed in the database using the function *INSTALL_B*(T, *tup*), which installs one tu-

```
Procedure BA(T,L);
 begin
      for each tup ∈ insertion-list do
   INSTALL_B(T,tup) ;
 end;
```

Fig. 11. The basic algorithm (BA).

ple using the TBSAM index T. Assuming $n \ll N$, the cost of the basic algorithm is

$$C^{BA} \approx n \lceil \log_d N \rceil .$$

B. The Grouped Algorithm (GA)

This algorithm works in two phases. The first phase is to sort the tuples in L. This can be performed in main memory and is not very expensive. The second phase installs the tuples of L in the database by means of a Quicksort-like recursive partitioning approach inserting the sublist L_i in subtree T_i. The algorithm is shown in Fig. 12.

We now derive an expression which is an approximation to C^{GA}, the cost of algorithm GA. The analysis makes the following assumptions.

1) Each nonroot node of the TBSAM index has the average B$^+$-tree fill-factor, i.e., d (= 0.7 m) keys. Thus, the fanout of each node is d.

2) The n tuples in the list L get separated from each other as soon as possible in the tree, as shown in Fig. 13. Thus, the analysis is worst case.

3) The sorting of L is carried out in main memory, and incurs negligible cost.

4) There is no buffering of index nodes.

The effects of removing assumptions 3) and 4) are discussed at the end of this section.

As shown in Fig. 13, the worst case of GA occurs when all the n tuples get separated into partitions of unit size early, and then continue to trickle down their respective subtrees. This case occurs when the partitioning is always of equal size, so that the initial sorting is made use of as little as possible.

Let i be the level of the TBSAM index at which all n tuples become separated for the first time. Thus, the top of the TBSAM index can be considered as a small tree that separates all tuples. The last level of this tree has d^{i-1} nodes (or d^i keys), where $d^{i-1} < (n/d) \leq d^i$. The height of this tree is $\lceil \log_d (n/d) \rceil - 1$. The number of nodes in this tree is

$$\frac{d^{\lceil \log_d (n/d) \rceil} - 1}{d - 1}.$$

Each of these nodes has to be brought into the main memory exactly once. Since the height of the whole tree, i.e., index and data levels, is $\lceil \log_d (N/d) \rceil + 1$, after getting isolated each tuple trickles down a distance of

```
Procedure INSTALL_G(T,L);
 begin
 if not(empty(L)) then
    begin
    (L0,L1, ··· Lm) ← SPLIT(L)
       for i ← 0 to m) do
          INSTALL_G(Ti,Li)
    end
 end

Procedure GA(T,L);
 begin
   L ← SORT(L) ;
   INSTALL_G(T,L) ;
 end
```

Fig. 12. The grouped algorithm (GA).

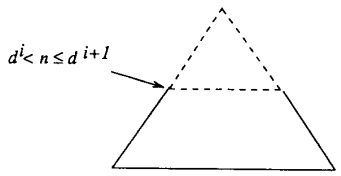


Fig. 13.

$\lceil \log_d (N/d) \rceil + 1 - (\lceil \log_d (n/d) \rceil - 1)$ levels. Since we have carried out a worst case analysis, the cost of algorithm GA is

$$C^{GA} \leq \frac{d^{\lceil \log_d (n/d) \rceil} - 1}{d - 1} + n\big(\lceil \log_d (N/d) \rceil - \lceil \log_d (n/d) \rceil + 2\big).$$

Let $S(n, d)$ be the savings obtained by using algorithm GA. Now,

$$\begin{aligned} S(n, d) &= C^{BA} - C^{GA} \\ &\geq n\big(\lceil \log_d (N/d) \rceil + 1\big) - \frac{d^{\lceil \log_d (n/d) \rceil} - 1}{d - 1} \\ &\quad - n\big(\lceil \log_d (N/d) \rceil - \lceil \log_d (n/d) \rceil + 2\big) \\ &\geq n \lceil \log_d (n/d) \rceil - n - \frac{d^{\lceil \log_d (n/d) \rceil} - 1}{d - 1}. \end{aligned}$$

Since $d^i < n \leq d^{i+1}$,

$$S(n, d) \geq n(i - 1) - \frac{d^i - 1}{d - 1}.$$

Choosing $i = 2$, i.e., $n > \approx d^2$, we get

$$\begin{aligned} S(n, d) &\geq n - \frac{d^2 - 1}{d - 1} \\ &= n - d - 1. \end{aligned}$$

In practice, B$^+$-tree indexes with 150–200 keys per node are quite common. If $n = 2d$, the savings obtained

is $2d - d - 1 = d - 1$ disk accesses. If $m \approx 183$, d is 128, and the savings are 127 disk accesses.

We now show, by means of an example, that removing assumption 3) does not affect our analysis in any significant manner. The sorting of n elements requires approximately $12n \log_2 n$ machine instructions [19]. Now, counting the number of machine instructions,

$$\begin{aligned} \textit{Cost of sorting} &= (12)(256)\log_2(256) \\ &= 24\ 576 \text{ machine instructions.} \end{aligned}$$

Assuming a disk drive with an access time of 25 ms, and a moderate CPU of 1 Mips, we have

$$\begin{aligned} \textit{Cost of sorting} &= 24\ 576 \text{ machine instructions.} \\ &= \frac{24\ 576}{\frac{(1\ 000\ 000)(25)}{1000}} \\ &= 0.98304 \\ &\approx 1 \text{ disk access.} \end{aligned}$$

This analysis shows that the cost of sorting, equivalent to 1 disk access, is negligible compared to the savings obtained by algorithm GA. The internal sorting also requires some main memory. Assuming a typical database tuple is 64 bytes long, the main memory requirement for sorting is (256)(64) bytes, i.e., 16 kbytes.

We now justify assumption 4). Various buffering strategies have been proposed for the B^+-tree index structure [21]. The main idea is to buffer as many index blocks as possible, starting from the root since the probability of accessing these blocks is high. The same techniques are also applicable to enhance the performance of the basic update algorithm (BA) for TBSAM. The savings in algorithm GA comes from the fact that blocks of the small tree in Fig. 10 do not have to be accessed repeatedly. If this tree can be completely buffered in main memory, BA will perform as well as GA. Thus, in an environment where buffer management is under Database Management System's (DBMS) control, GA may not be very advantageous. However, practically no operating system understands the data access patterns of DBMS's, treating them as any other user process, and using buffer management strategies that are not good for the DBMS [20]. Various studies [21], [22] have shown that well accepted operating system buffer management strategies such as Least Recently Used (LRU) and Working Set do not perform well for DBMS's. New models and algorithms including the Hot Set Model [22] and Query Locality Set Model [21] have been proposed as alternatives. While we wholeheartedly agree with the need for these smarter buffer management strategies, we believe that algorithms such as GA are very useful in their absence. A smart buffer management scheme with complete application knowledge will, to a large extent, do as well as GA.

C. Effect of GA on Query Processing

If a grouped update algorithm is used, the database is not completely up-to-date at all times. The database *ages* between successive runs of the grouped update algorithm, and in the presence of high update activity may be substantially out of sync. The solution we propose is to store all the incoming updates in an auxiliary main memory data structure. Each query is processed using the database as well as the auxiliary structure. A good structure to use is the *binary heap*, which provides the sorted order required by the first phase of the GA update scheme in a natural manner. Since the auxiliary structure is in main memory, searching is fast, and query processing cost increases only marginally. Details of this scheme are discussed in [23].

The problem in maintaining main memory structures is of reliability, because of the volatile nature of the medium. This may not turn out to be a major problem for statistical queries, since the results are *estimates* rather than precise answers. This is especially true for the domain of Statistical and Scientific Databases. However, if absolute precision is of concern and reliability is an issue, a second memory based differential file can be used as the auxiliary structure [24].

VII. Conclusions

We have presented a Tree Based Statistics Access Method (TBSAM) for the efficient processing of aggregate queries. It exhibits the efficient dynamic behavior of the B^+-tree, in addition to enhancing the performance of aggregate queries. The B-tree and its variations provide an ordering of tuples on the index attribute, but are not suitable for retrieving a tuple given its rank. TBSAM supports this operation very naturally. It was also shown how this operation can be used to perform sampling as well as calculation of order statistics.

TBSAM is a dynamic index, and thus can support insertion/deletion/modification of tuples in a relation. These operations can be performed efficiently by TBSAM, and the cost is almost the same as that for the B^+-tree. Overhead for updates to a relation can be expensive, especially if the update rate is high. A grouped algorithm for installing updates in a batched manner was presented and analyzed.

It is clear that TBSAM is suitable for applications such as SSDB's where aggregate queries form a significant fraction. However, we believe that the aggregate queries found in standard database applications would also benefit from it. The B^+-tree index is provided by almost all commercially available systems. A major advantage of the close link between B^+-tree and TBSAM is that the latter can be incorporated into existing systems without much effort.

Over the years statisticians have developed a myriad of tools and techniques for statistical analysis. Even though TBSAM can be used for efficiently processing some kinds of statistical queries, there are many more that are not covered. These include operators to provide support for

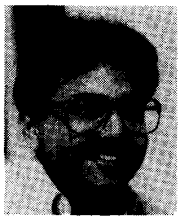
diverse statistical techniques such as Regression Analysis, Singular Value Factorization, Principal Component Analysis, etc. With the growing importance of SSDB's, more database support for these statistical analysis methods is required.

Acknowledgment

The authors would like to thank Dr. S. P. Ghosh of IBM Almaden Research Center, and Prof. C. V. Ramamoorthy of the University of California, Berkeley, for numerous helpful discussions and insightful criticism.

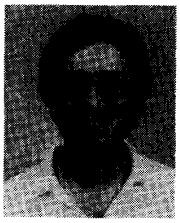
References

[1] IBM Manual, "System 360 operating system, index sequential access methods (programming logic)," Form Y28-6618.
[2] R. Bayer and C. McCreight, "Organization and maintenance of large ordered indexes," *Acta Inform.*, vol. 1, no. 3, pp. 173-189, 1972.
[3] J. Nievergelt, H. Hinterberger, and K. C. Sevcik, "The grid file: An adaptable, symmetric multikey file structure," *ACM Trans. Database Syst.*, vol. 9, pp. 38-71, Mar. 1984.
[4] W. Litwin, "Linear hashing: A new tool for file and table addressing," in *Proc. 6th Int. Conf. VLDB*, 1980.
[5] M. Freeston, "The bang file: A new kind of grid file," in *Proc. ACM SIGMOD Int. Conf. Management Data*, May 1987, pp. 260-269.
[6] A. Guttman, "R-trees: A dynamic index structure for spatial searching," in *Proc. ACM SIGMOD Int. Conf. Management Data*, 1984, pp. 47-57.
[7] H. Samet, "The quadtree and related hierarchical data structures," *ACM Comput. Surveys*, vol. 16, pp. 187-260, June 1984.
[8] A. Klug, "Equivalence of relational algebra and relational calculus query languages having aggregate functions," *J. ACM*, vol. 29, pp. 699-717, 1982.
[9] S. P. Ghosh, "SIAM: Statistics information access method," *IBM Res. Rep.*, RJ 4865 (51295).
[10] F. Olken, "Physical database support for scientific and statistical database management," in *Proc. 3rd Int. Workshop Statist. Sci. Database Management*, Luxembourg, July 22-24, 1986.
[11] A. Shoshani, "Statistical databases: Characteristics, problems, and some solutions," in *Proc. 8th Int. Conf. VLDB*, 1982, pp. 208-222.
[12] D. Comer, "The ubiquitous B^+-tree," *Comput. Surveys*, vol. 11, no. 2, pp. 121-138, 1979.
[13] B. Schneiderman and V. Goodman, "Batched searching of sequential and tree structured files," *ACM Trans. Database Syst.*, vol. 1, pp. 268-275, Sept. 1976.
[14] P. Palvia, "Expressions for batched searching of sequential and hierarchical files," *ACM Trans. Database Syst.*, vol. 10, pp. 97-106, Mar. 1985.
[15] M. Piwowarski, "Comments on batched searching of sequential and tree-structured files," *ACM Trans. Database Syst.*, vol. 10, pp. 285-287, June 1985.
[16] D. S. Batory and C. C. Gotlieb, "A unifying model of physical databases," *ACM Trans. Database Syst.*, vol. 7, pp. 509-539, Dec. 1982.
[17] W. G. Cochran, *Sampling Techniques*. New York: Wiley, 1953.
[18] F. Olken and D. Rotem, "Simple random sampling from relational databases," in *Proc. Int Conf. VLDB*, Aug. 1986.
[19] D. E. Knuth, *The Art of Computer Programming*, Vol. 3. Reading, MA: Addison-Wesley, 1973.
[20] M. Stonebraker, "Operating system support for database management," *Commun. ACM*, vol. 24, pp. 412-418, July 1981.
[21] G. M. Sacco and M. Schkolnick, "A mechanism for managing the buffer pool in a relational database system using the hot set model," in *Proc. 8th Int. Conf. VLDB*, Mexico, 1982, pp. 257-262.
[22] H.-T. Chou and D. J. Dewitt, "An evaluation of buffer management strategies for relational database systems," in *Proc. 11th Int. Conf. VLDB*, Stockholm, Sept. 1985.
[23] J. Srivastava and C. V. Ramamoorthy, "Efficient algorithms for maintenance of large database indexes," in *Proc. 4th Int. Conf. Data Eng.*, Los Angeles, CA, Feb. 1988.
[24] G. M. Lohman and D. G. Severance, "Differential files: Their application to the maintenance of large databases," *ACM Trans. Database Syst.*, vol. 1, pp. 256-267, Sept. 1976.
[25] S. B. Yao, "Approximating block accesses in database organizations," *Commun. ACM*, vol. 20, Apr. 1977.



Jaideep Srivastava (S'84-M'88) received the B.Tech. degree from the Indian Institute of Technology, Kanpur, India, in 1983, and the M.S. and Ph.D. degrees in computer science from the University of California, Berkeley, in 1985 and 1988, respectively.

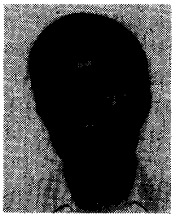
At present he is an Assistant Professor in the Department of Computer Science, University of Minnesota, Minneapolis, a position he has held since 1988. In 1983 he was a Research Engineer with Uptron Digital Systems, Lucknow, India. He has authored and refereed papers in the areas of databases, distributed systems, and computer networks. His research interests lie in the areas of active databases, real time systems, computer security, and parallel expert system architectures.



Jack S. Eddy Tan was born in Singapore in 1958. He received the B.S. and M.S. degrees in computer science from the University of Minnesota, Minneapolis, in 1983 and 1986, respectively.

He is currently a Ph.D. candidate in the Department of Computer Science, University of Minnesota, Minneapolis. His research interests include active databases, parallel computing, real-time systems, and fault tolerant systems.

Mr. Tan is a member of the IEEE Computer Society.



Vincent Y. Lum (S'65-M'66-SM'79) received the Bachelor of Applied Science in Engineering Physics degree from the University of Toronto, Toronto, Ont., Canada, in 1960, the M.S. degree in electrical engineering from the University of Washington in 1961, and the Ph.D. degree in electrical engineering from the University of Illinois in 1966.

At present he is a Professor in the Department of Computer Science at the Naval Postgraduate School (NPS), Monterey, CA. He joined NPS in 1985 after nearly 25 years of service with IBM. His main interests are in the areas of database management systems, data organization, system analysis, and office automation. He has authored and coauthored numerous technical articles in major journals, conference proceedings, and books. His early work in data organization has been widely referenced and stimulated much research in that area.

Dr. Lum is a member of ACM, the IEEE Computer Society, and Sigma Xi. He is active in professional affairs. He is the Co-Founder and the current Chairman of the IEEE Computer Society's TCOA, a Past Chairman of the TCDE, and a trustee of the VLDB Endowment. He has been an Associate Editor of the ACM TODS and the TOOIS, and a key organizer of various conferences including Conference Chairman and Program Chairman.