Progressive Approximate Aggregate Queries with a Multi-Resolution Tree Structure *

Iosif Lazaridis
Dept. of Information and Computer Science
University of California, Irvine, USA
iosif@ics.uci.edu

Sharad Mehrotra
Dept. of Information and Computer Science
University of California, Irvine, USA
sharad@ics.uci.edu

ABSTRACT

Answering aggregate queries like SUM, $COUNT$, MIN, MAX, AVG in an approximate manner is often desirable when the exact answer is not needed or too costly to compute. We present an algorithm for answering such queries in multi-dimensional databases, using selective traversal of a Multi-Resolution Aggregate (MRA) tree structure storing point data. Our approach provides 100% intervals of confidence on the value of the aggregate and works iteratively, coming up with improving quality answers, until some error requirement is satisfied or time constraint is reached. Using the same technique we can also answer aggregate queries exactly and our experiments indicate that even for exact answering the proposed data structure and algorithm are very fast.

1. INTRODUCTION

We deal with the problem of answering aggregate queries in a multi-dimensional space containing point data items. The data space is $R_{space} \subseteq \Re^d$ where d is the dimensionality. Data items are pairs $\langle loc, values \rangle$ where $loc \in R_{space}$ is a d-dimensional point and $values$ is a tuple of attributes associated with the data point.

Databases of this type are quite common in spatial applications, e.g. temperature/waterfall readings from sensors embedded in space, point-of-sale data on a geographical map, objects that are mobile in space. They are also found in On-line Analytical Processing (OLAP) applications; in such applications the dimensions correspond to attributes of the data and the $value$ tuple holds the attributes of particular interest. An example would be $values$ being "total purchases" with the dimensions being "age" and "income"; each customer is represented in the age-income space with a single point with attribute value equal to his total purchases from our store.

*This work was supported in part by the National Science Foundation under Grant No. IIS-0086124 and in part by the Army Research Laboratory under Cooperative Agreements No. DAAL-01-96-2-0003 and No. DAAD-19-00-1-0188.

ACM SIGMOD 2001 May 21-24, Santa Barbara, California, USA

Aggregate Queries are a very common and interesting type of queries over databases of this sort. An aggregate query consists of a region $R^Q \subseteq R_{space}$ and one or more aggregates to be determined for all data items in that region. The most common aggregates are the ones traditionally provided by SQL, i.e., $COUNT, SUM, MIN, MAX, AVG$. Thus, the user is interested to answer queries like: "Maximum total sales in any store in Orange County", or "Average measured rainfall in Siberia", "total purchases of customers of age 20-25 and income between 25,000-45,000 USD" and so on.

Answering aggregate queries is straightforward, but can be computationally expensive; a large part of the database (namely all points lying in the query region) has to be examined to precisely determine the aggregate. In the next section we review the techniques that have been proposed to tackle the computational cost of aggregate queries. If a precise answer is expected, then the relevant portion of the database must be examined. If the query region is relatively small, a spatial index can be used to gather up all relevant tuples. For queries that cover a large part of the data space, an expensive linear scan can be performed. In both cases at least all the tuples that fall in the query region have to be read. There is great interest in methods that provide *approximate* answers to aggregate queries at a significantly lower computational cost, by providing a "good" estimate without accessing large portions of the database or index.

The approach presented in this paper uses a tree structure called *Multi-Resolution Aggregate tree* (MRA-tree). Our algorithm selectively traverses nodes of this tree based on reasonable assumptions on which nodes, if examined, will most likely reduce the uncertainty on the value of the aggregate. Tree nodes are augmented with aggregate information for all data points indexed by them. Thus, if a node is completely contained in the query region, no further traversal of its children is needed. If it partially overlaps with the query, its aggregate information can be used both to estimate an approximate answer and to select a "good" node to explore at each stage of the algorithm. Note that the goal is not simply to approximate the shape of the query using the nodes of the MRA-tree; rather it is to explore nodes intersecting with the query region with the objective of producing the best answer to the query in the least amount of time. MRA-tree is a generic structure that can be instantiated with any tree index for which the data space covered by each tree node is contained in the space covered by its parent. All conventional tree indexes, both of the space-partitioning (e.g., quadtree [12], K-D-B tree [10]) and of the data-partitioning (R-Tree [5]) variety satisfy the above property. Thus, we can

have an MRA-quadtree, MRA-RTree etc. No assumptions are made either for the spatial or attribute value distribution of the data in the estimation of the potential error.

The rest of the paper is organized as follows: In Section 2 we survey the techniques proposed in the literature for aggregate query estimation. In Section 3 we describe the MRA-tree and how it differs from regular tree indexes. In Section 4 we propose a progressive algorithm for aggregate queries that includes error estimation, uncertainty bounds and traversal policy of the data structure. In Section 5 we detail how different aggregate types are handled by the generic strategy. We present our experimental results over real and synthetic data in Section 6 using both a space- and data-partitioning (MRA-quadtree/R-Tree) data structure. Our work is summarized in Section 7.

2. RELATED WORK

Recently, much work has been done on approximate aggregate queries in multi-dimensional databases and various techniques have been explored, including sampling, wavelets and histograms. Most of this work was motivated from On-line Analytical Processing (OLAP) applications; in such applications aggregate queries are quite common (e.g., "What is the average salary of all employees aged 25 to 35") and the database size makes their exact answering extremely costly. In many of the proposed techniques, an estimator is built from the whole database and is stored at a small fraction of the database size. Queries are subsequently evaluated efficiently against the estimator.

A limiting factor of the proposed techniques is that no guarantee as to the estimate's discrepancy from the true answer is usually given. Moreover, with a fixed-size estimator, providing progressively improving quality answers for applications with different quality requirements is impossible (beyond some level of precision). For some aggregation applications (e.g., "total vote count") a perfect-quality answer is expected. On the other hand, in a Decision Support system with OLAP data, it is often desirable to receive a reasonable answer in a few seconds, rather than a perfect-quality one that takes hours to compute. A second problem with preprocessed off-line estimators is that they cannot adapt the estimation to the particular query posed to the system. For each query, a unique estimate is given based on the data summary (e.g., the histogram); the summary captures the interesting features of the data but cannot adapt to the query load. Even if there was a method for detecting that the estimate is very poor for some particular query, there is no way (by means of the estimator) of improving it.

Sampling techniques essentially take a small random sample of the database and compute the aggregate on the sample. In off-line sampling, a sample is extracted from the database and queries are subsequently run against the sample. In on-line sampling, tuples are read randomly from the database in response to a query and aggregation on the extracted sample is performed at the same time. In Hellerstein et. al. [6] confidence intervals are given based on statistics about the tables kept in the database. This technique is similar to ours in that it also aims to produce improving quality estimates in a progressive manner. The method presented in this paper differs in that it offers deterministic guarantees of error and can also be used for determining the exact answer of the aggregate query by performing a very small number of I/Os. Lately Gunopulos et. al. [4] proposed kernel estimation as an extension to simple sampling for selectivity estimation (i.e., $COUNT$ aggregate) that improves the quality of the given answers. While sampling reduces the I/O cost of answering aggregates by visiting a random sample of the database, our approach uses a hierarchical data structure to only examine data that are relevant to the user's query.

Histogram techniques (Ioannidis and Poosala [8], Gunopulos et. al. [4]) work by subdividing the data space into a number of buckets; aggregate information about the buckets is kept. The estimation is given by calculating the overlap of the query region with the various buckets and aggregating over all overlapping buckets. Commonly, a uniformity assumption is made about the distribution of data points within each bucket. In the GENHIST technique proposed in [4] it was shown how for a given number of buckets, the data space can be approximated at a higher resolution by allowing histogram buckets to overlap. The histogram techniques suffer from the *dimensionality curse* as the number of buckets required to approximate the data space increases exponentially with the dimensionality and the uniformity assumption does no longer hold at high dimensionalities. Since no error bounds are usually given and no way to improve the estimation for additional computational cost has been proposed, the value of these techniques is diminished at higher dimensionalities. Incidentally, the techniques for error estimation presented in this paper can be straightforwardly adapted to provide error bounds for histogram-based estimation techniques. However, there is no way to use these guarantees for progressive, improved-quality answers (since histograms are an off-line estimator).

Wavelets (Vitter and Wang [13], Chakrabarti et. al. [3]) are used to hierarchically decompose functions. The idea is to apply wavelet decomposition to the input data collection (OLAP cube or attribute columns) which produces as an output a number of *wavelet coefficients* that require minimum storage space and can be queried on directly. Experimental results in [13] indicate that processing the input data to produce wavelet coefficients can be done fast and the resulting estimates are fairly accurate even for a small number of coefficients.

Barbará et. al. [2] have addressed the problem of approximate query answering in data warehouses. In their approach, the cells of the Data Cube are preclassified into error bins; subsequently data is brought in from the bins progressively to refine the answer.

A technique proposed by Aoki [1], also uses selective traversal of a multi-dimensional index for the problem of selectivity estimation (corresponding to estimating the $COUNT$ aggregate in our case) by providing extensions for Generalized Search Tree (GiST, Hellerstein et. al. [7]). We deal with all SQL-type aggregates, supplying for each one methods for tree traversal and aggregate estimation. In our experimental study we investigate the MRA-tree's efficiency in varying dimensionality, database size, data type and index type. We also tackle the central problem of the tradeoff between space used to store the aggregates and speed of answer reporting in terms of I/O operations.

The present technique differs from most previous ones in that: (a) it handles all SQL-type aggregates, (b) it provides explicit bounds on the value of the aggregate and (c) it allows the incremental reporting of improved-quality answers all the way to the exact answer.

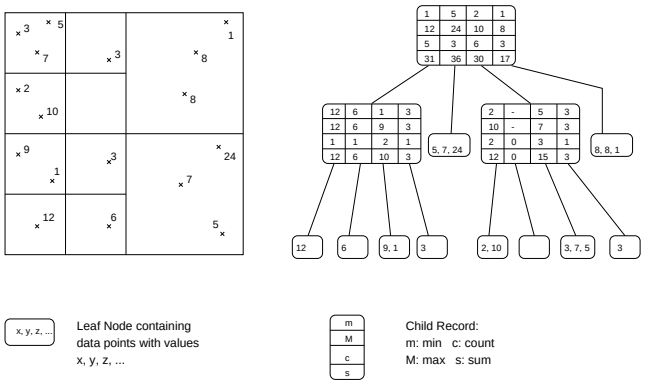

Figure 1: An MRA-quadtree: each non-leaf node contains MIN, MAX, COUNT, SUM of its four children

3. INDEX STRUCTURE ORGANIZATION

We consider the problem of an index structure storing point data of the form $\langle loc, values \rangle$. For all such points: $loc \in R_{space}$ and $values \in \mathcal{D}^v$. $R_{space} \subseteq \mathcal{R}^d$ is the data space of dimensionality d and $\mathcal{D}^v$ is the value domain. As an example R_{space} might be a 2D map of the United States and $\mathcal{D}^v = [0, \infty)$ the value domain of rainfall measurement. The user provides a query region $R^Q \subseteq R_{space}$ and is interested in the value of some of the aggregates ($COUNT$, SUM, MIN, MAX, AVG) over the query region.

The proposed Multi-Resolution Aggregate (MRA) tree is a modified multi-dimenional index structure. Each node N of an MRA-tree is associated with a region R^N and indexes a set of data points contained in that region. Non-leaf nodes contain entries $\langle ptr, region, aggregates \rangle$ for each of their children nodes: ptr is a pointer to a child node, $region \subseteq R_{space}$ is the space covered by that node and $aggregates$ is a tuple $\langle MIN, MAX, SUM, COUNT \rangle$ of aggregate information over all data points indexed by that node. The number of children per node n_N (fanout) can be either constant (as in quadtree $n_N = 4$) or ranging up to a maximum number dictated by the page size (in paginated structures like R-Tree). The $region$ field may be stored explicitly (as in R-Tree) or it can be derived from the splitting point (quadtree) or hyperplane (K-D-B tree). A leaf node contains up to n_{leaf} tuples of $\langle loc, values \rangle$. An example of an MRA-quadtree is seen in Figure 1.

An MRA-tree can be created using any data structure that satisfies the property that for each node N with associated region R^N, if N^c is a child node of N, or (in the case that N is a leaf) P is a data point contained in N, then $R^{N^c} \subseteq R^N$ or $P.loc \in R^N$. This property is satisfied by all conventional indexes and guarantees that each MRA-tree node can be used to store the aggregate information for all points indexed by it.

We will now discuss how the algorithms for management of an MRA-tree differ from those of regular tree indexes. The main difference is introduced by the requirement that each node must store the aggregates for all data points it indexes. Thus, algorithms for searching using the MRA-tree are identical to those of the corresponding "plain" tree because search does not affect the aggregate values stored. However the algorithms for *insert*, *delete* and *update* are modified to consistently maintain the aggregate information.

Insertion: Given an MRA-tree T and a data point $P = \langle loc, val \rangle$ we want to insert P into T. The point will be inserted in some leaf node N_{leaf}. Nodes affected are all ancestors N of N_{leaf}. At each N the aggregates are updated as follows:

$$\begin{aligned} min^N &:= min\{min^N, val\} \\ max^N &:= max\{max^N, val\} \\ count^N &:= count^N + 1 \\ sum^N &:= sum^N + val \end{aligned}$$

When N_{leaf} is reached, the point is inserted if $count^{N_{leaf}} < n_{leaf}$ and no further action is required. Otherwise the node is split. The split algorithm is identical to that of the normal index tree with the added consideration that the aggregate information must be maintained. In a quadtree we need only create a new non-leaf node N_{new} and four new leaf nodes, redistribute the data points of N_{leaf} to the new leaves and post the aggregates to N_{new}. In a K-D-B tree or R-Tree the split may propagate all the way up to the root; thus we may have to update the aggregates for all affected nodes.

Deletion: We want to delete $P = \langle loc, val \rangle$ from MRA-tree T. P resides on leaf node N_{leaf} and is contained in all ancestors N of N_{leaf}. If for each such N: $min^N < val < max^N$ then the aggregates are updated as follows:

$$\begin{aligned} count^N &:= count^N - 1 \\ sum^N &:= sum^N - val \end{aligned}$$

If however $val = min^N$ then in addition to the above we must recalculate min^N. This is done as follows: find the MIN of all points indexed by N_{leaf} and update $min^{N_{leaf}}$ stored at the parent of N_{leaf}; let's call this parent N^p_{leaf}. Then find the $min^{N^p_{leaf}}$ by calculating the MIN of N_{leaf} and its siblings. If the new $min^{N^p_{leaf}}$ is changed then push that change to the parent of N^p_{leaf} and so on. The number of MIN values that have to be updated is upper-bounded by the number of nodes that contain P; in the worst case, if $min^T = val$, the MIN value of all points up to the root has to be updated. The same logic applies if $val = max^N$. If due to a deletion, nodes of the tree are merged, the new aggregates are easy to compute: if N_1 and N_2 join to form a new node N, then

$$\begin{aligned} min^N &:= min\{min^{N_1}, min^{N_2}\} \\ max^N &:= max\{max^{N_1}, max^{N_2}\} \\ count^N &:= count^{N_1} + count^{N_2} \\ sum^N &:= sum^{N_l} + sum^{N_2} \end{aligned}$$

Update: A value update consists of changing the value of point $P = \langle loc, val \rangle$ into $P' = \langle loc, val' \rangle$. An update can be seen as a deletion of P and insertion of P' into the same leaf node N_{leaf}. With regard to $COUNT$ and SUM the update of the aggregates for all ancestor nodes N of N_{leaf} is as follows:

$$\begin{aligned} count^N &:= count^N \\ sum^N &:= sum^N - val + val' \end{aligned}$$

We need to update the MIN or MAX aggregates for an ancestor node N only if $val = min^N \vee val = max^N$ or $val' < min^N \vee val' > max^N$. For the case of the MIN, if

```
(1)  class Query
(2)  {
(3)     R^Q: Region;
(4)     agg: enum {MIN, MAX, COUNT, SUM, AVG};
(5)     N^c:=∅,N^p:=∅: set of Node;
(6)  };

(7)  Query::Query(queryR: Region, root: Node, aggType)
(8)  {
(9)     R^Q := queryR;
(10)    agg := aggType;
(11)    if (R^Q ∩ R^root = R^root) {
(12)       N^c := {root};
(13)    else
(14)       N^p := {root};
(15) };

(16) Query::nextAnswer(var ans: Real, var low: Real, var high: Real)
(17) {
(18)    if (N^p = ∅) {
(19)       estimate ans from N^c;
(20)       low := high := ans;
(21)    }
(22)    else {
(23)       remove node N from N^p;
(24)       forall children N_i : i = 1, . . . , n_N of node N {
(25)          if (R^Q ∩ R^N = ∅)
(26)             ignore;
(27)          else if (R^Q ∩ R^N = R^N)
(28)             insert N_i into N^c;
(29)          else
(30)             if N_i is useful
(31)                insert N_i into N^p;
(32)       }
(33)       estimate ans from N^p, N^c;
(34)       calculate low, high from N^p, N^c;
(35)    }
(36) };
```

Figure 2: Progressive Aggregate Query Algorithm

$val = min^N \wedge val' < val$ then $min^N := val'$ and the change is propagated to the parent of N as we have previously described. If $val = min^N \wedge val' > val$ then min^N needs to be re-calculated from all points indexed by N (after removing P and inserting P'). If $val > min^N \wedge val' > min^N$ no action is necessary. Finally, if $val > min^N \wedge val' < min^N$ then we need to update $min^N := val'$ and once again propagate the change to the parent of N.

The tree management operations presented above have a compexity of $O(h)$ where h is the height of the MRA-tree. In applications were frequent updates are commonplace and run concurrently with queries, the above cost may be prohibitive. In many OLAP applications, the data is loaded into a data warehouse periodically and queries subsequently run over an essentially static data set. In high-update environments, e.g., sensor databases, real-time monitoring systems, etc., the problem can be addressed in either of two ways: (a) defer the updates for the nodes affected or (b) use a data-driven update strategy that percolates updates from the data sources (e.g., sensors) to the higher levels of the hierarchy. In both of these approaches, the higher-level nodes of the data structure hold an "approximate" picture of the aggregate information of the data points indexed by them. In our current work we investigate the 3-way tradeoff: computational cost vs. data representation quality vs. estimate quality. In the sequel we will assume that the update cost is manageable.

4. PROGRESSIVE ALGORITHM

We will now discuss a progressive algorithm for approximate aggregate queries. The query region is $R^Q \subseteq R_{space}$. During the course of the algorithm, we maintain two sets of tree nodes:

(i) $\mathcal{N}^c$: a set of nodes completely contained in the query, i.e. $N \in \mathcal{N}^c \Rightarrow R^N \cap R^Q = R^N$ and

(ii) $\mathcal{N}^p$: a set of nodes that either enclose or are partially overlapping with the query ($R^N \cap R^Q \neq R^N \wedge R^N \cap R^Q \neq \emptyset$)

In Figure 3 we see the possible relationships of a node to a query. Case (a) corresponds to $\mathcal{N}^c$ and cases (b),(c) correspond to $\mathcal{N}^p$.

(a) **Contained**. For every $N \in \mathcal{N}^c$ the aggregate information of all points indexed by it certainly affects the answer to the query since it is entirely contained therein. Further traversal of the node's subtree is unnecessary.

(b) **Partially Overlapping**. The node's aggregate information is derived both from $R^N \cap R^Q$ and $R^N - R^N \cap R^Q$. Further traversal of the node's subtree will give a better approximation of the node's contribution.

(c) **Enclosing**. The node's aggregate information is derived both from R^N and $R^N - R^Q$ and thus only partially affects the query. Again, further traversal will give a better approximation.

(d) **Disjoint**. The node is irrelevant to the query.

The progressive algorithm (Figure 2) works as follows: we initialize sets $\mathcal{N}^c$ and $\mathcal{N}^p$ with the root node *root* (lines 11-14): if the query covers the whole space then obviously $\mathcal{N}^c = \{root\}$, $\mathcal{N}^p = \emptyset$ and the query can be immediately answered by reading the specific aggregate about the root. Most queries however will be a subset of the whole space, thus we will have $\mathcal{N}^p = \{root\}$ and $\mathcal{N}^c = \emptyset$. The algorithm works iteratively, with successive calls to $nextAnswer()$ (line 16). Each iteration first checks if there are partially overlapping nodes left in the queue; if not then the query is answered exactly based only on $\mathcal{N}^c$. If $\mathcal{N}^p$ is not empty, $nextAnswer()$ does the following:

(a) removes a node N from $\mathcal{N}^p$ (line 23).

(b) visits the children of N and classifies them as one of the cases of Figure 3 (lines 24, 25, 27, 29).

(c) inserts the children of N appropriately either to $\mathcal{N}^p$ or $\mathcal{N}^c$ or discards them (lines 26,28,30,31). Note that the statement **is useful** on line 30 indicates that a partially overlapping node can also be discarded (as we will see in the case of MIN and MAX aggregates).

(d) provides an estimate on the aggregate (line 33).

(e) provides a 100% interval of confidence on the aggregate (line 34).

The algorithm as given in Figure 2 is generic and to instantiate it into a "real" algorithm we must specify a few things. In the following section we will show a reasonable instantiation of the algorithm for the SQL-type aggregates. For each aggregate type the following must be resolved:

- **Intervals of Confidence** — item (e) of above list; these are minimum-length 100%-confidence ranges that depend exclusively on the type of aggregate and on sets $\mathcal{N}^p, \mathcal{N}^c$ and are calculated independently of the data distribution.

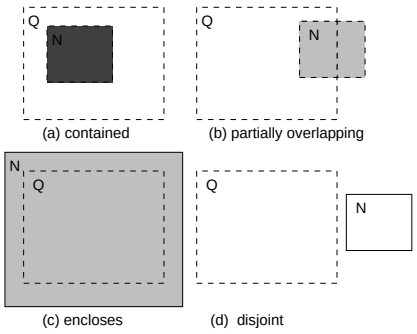

Figure 3: Relation of Node to Query

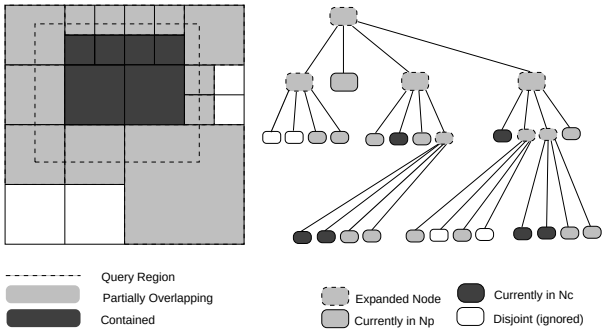

Figure 4: Query region is approximated by progressive subdivision of the MRA-tree

- **Estimation Policy** — item (d) of above list; provides the mechanism for estimating the value of the aggregate from $\mathcal{N}^p, \mathcal{N}^c$. Some assumption about the spatial/value-domain distribution of the data has to be made in order to provide an estimate.

- **Traversal Policy** — item (a) of above list; the order in which nodes of $\mathcal{N}^p$ are visited has to be determined.

In Figure 4 we see an MRA-quadtree data structure; nodes of the tree have been visited at different levels of the tree to answer a rectangular query.

A few words on the notation we will use: $\widehat{agg^{Q,X}}$ is the estimate for query Q of aggregate type agg over attribute X (value stored at each data point). The exact answer to the query is $agg^{Q,X}$. The 100% interval of confidence is denoted as $I_{agg}^{Q,X} = [agg_{low}^{Q,X}, agg_{high}^{Q,X}]$.

As will become clear the above algorithm will produce monotonically improving quality answers (i.e., $\|I_{agg}^{Q,X}\|$ will be decreasing over successive iterations) regardless of the traversal policy; at each stage of the algorithm the query region is better approximated via the MRA-tree grouping of points — no information is ever lost. The stopping criteria of the algorithm can be either based on error and/or time. Answers are returned via successive calls to $nextAnswer()$. It is up to the user to stop this process if some application-specific timing constraint has been reached or the given quality of the answer (as indicated by $I_{agg}^{Q,X}$) is sufficient. The iterative process will terminate on its own when $\mathcal{N}^p$ is exhausted, at which stage there is an exact answer.

In coming up with a suitable traversal policy the aim should be to reduce the uncertainty (quantified by $\|I_{agg}^{Q,X}\|$) as fast as possible. In other words, for any number k of nodes visited we want $\|I_{agg}^{Q,X}\|$ to be minimized. Finding an optimum way to handle this problem is impossible since we are always making a local decision. As an example suppose there are two partially overlapping nodes N_1, N_2 with MIN values 3 and 4 respectively. No matter which node we choose to visit first, there is always the possibility that the other one contains the actual minimum value for our query. The complexity introduced by the traversal policy has to be taken into account as well; if for instance our quality requirement involves visiting most of the partially overlapping nodes, it makes little sense to pay the computational cost to visit them in any particular order. Thus, we could just as well do a simple depth-first or breadth-first traversal of the data structure. On the other hand, if we want a rapid improvement of quality in as few iterations as possible, then a complex traversal policy like the ones we will propose makes more sense.

An approach similar to that used in many similar problems (e.g., [11]) would be to organize $\mathcal{N}^p$ as a priority queue keyed on each node's contribution of uncertainty to the answer of the aggregate query and expand nodes in the order of decreasing uncertainty. This approach is based on the expectation that the maximum benefit (in terms of reducing $\|I_{agg}^{Q,X}\|$) will be achieved by exploring nodes that maximally contribute to $\|I_{agg}^{Q,X}\|$.

5. INTERVALS OF CONFIDENCE, AGGREGATE ESTIMATION AND TRAVERSAL POLICY

In this section we will show how the generic algorithm previously described can be instantiated for the various aggregate types. For each aggregate type we will show how the minimal 100% intervals of confidence $I_{agg}^{Q,X}$ are calculated. In addition, we will describe how the estimation $\widehat{agg^{Q,X}}$ can be derived from $\mathcal{N}^p, \mathcal{N}^c$ under the spatial uniformity assumption. We will also quantify the potential error contribution of each node in $\mathcal{N}^p$ to the query answer; this can then be used as a priority in a priority-based traversal algorithm.

A notation that we will use is the following: A_N or A_{R^N} is the area (or volume) overed by node N, and $agg^{c,X}$, $agg^{p,X}$ are the value of the aggregates for the data points indexed by all nodes in $\mathcal{N}^c$ and $\mathcal{N}^p$ respectively.

5.1 Min and Max

We want to find $min^{Q,X}$ or $max^{Q,X}$. We will only discuss the case for MIN, but also give the formulas for the MAX case which can be similarly derived to the MIN one.

Interval of Confidence.— Since nodes in $\mathcal{N}^c$ are completely contained in the query, the value of $min^{Q,X}$ can be no higher than $min^{c,X}$. If $min^{p,X} \geq min^{c,X}$ then all partially overlapping nodes can never give a lower minimum than $min^{c,X}$. Thus, there is no need to traverse any of them. If $min^{p,X} < min^{c,X}$ then potentially one of the nodes in $\mathcal{N}^p$ includes a data point with a value less than $min^{c,X}$ and thus further exploration of such nodes is necessary. The interval

of confidence will be:

$$I^{Q,X}_{min} = [min\{min^{p,X}, min^{c,X}\}, min^{c,X}]$$
$$I^{Q,X}_{max} = [max^{c,X}, max\{max^{p,X}, max^{c,X}\}]$$

Estimation.— The estimation given, depends on the semantics required by the user. Two potential estimations are the following:

- Lower bound on the value of $min^{Q,X}$: $\widehat{min^{Q,X}} = min^{p,X}$.
  Similarly upper bound for $max^{Q,X}$: $\widehat{max^{Q,X}} = max^{p,X}$
- Best worst case error (choose middlepoint of $I^{Q,X}_{min}$) : $\widehat{min^{Q,X}} = min^{p,X} + \frac{\|I^{Q,X}_{min}\|}{2}$.
  Similarly, $\widehat{max^{Q,X}} = max^{p,X} - \frac{\|I^{Q,X}_{max}\|}{2}$.

Traversal Policy.— Given a set $\mathcal{N}^p$ of partially overlapping nodes; the task is to pick the "best" node N_b from the set. First of all, it is easy to see that all nodes $N : min^{N,X} \geq min^{c,X}$ can be safely disregarded. Our goal is to reduce the length of $I^{Q,X}_{min}$. This can be done either by (i) increasing the $min^{p,X}$ or by (ii) decreasing the $min^{c,X}$. Case (i) can only happen if we pick the node $N_b : min^{N_b,X} = min^{p,X}$ and some child of N_b contributing the low $min^{p,X}$ value is found to be outside the query region. Case (ii) will occur for any candidate node if some child node N^c_b of N_b with $minN^c_b, X < min^{c,X}$ is found to be contained within the query region. Based on this observation, the node chosen to be expanded will be: $N^{MIN}_b \in \mathcal{N}^p$ and $N^{MAX}_b \in \mathcal{N}^p$, such that $\forall N \in \mathcal{N}^p$:

$$min^{N^{MIN}_b,X} < min^{c,X} \quad \wedge \quad min^{N^{MIN}_b,X} \leq min^{N,X}$$
$$max^{N^{MAX}_b,X} > max^{c,X} \quad \wedge \quad max^{N^{MAX}_b,X} \geq max^{N,X}$$

Implementing this traversal policy is straightforward if we maintain the nodes in a priority queue in ascending MIN order. A single node N_b is dequeued and its value is checked against $min^{c,X}$. If $min^{N_b,X} \geq min^{c,X}$ then the exact answer is found ($min^{Q,X} = min^{c,X}$), otherwise the children of N_b are pushed into the queue and a new node is popped.

5.2 Count

We want to find $count^Q$. We use notation $count^Q$ instead of $count^{Q,X}$ because it is independent of attribute X.

Interval of Confidence.— The lower bound on $count^Q$ is the total count of nodes in $\mathcal{N}^c$ (if all points in partially overlapping nodes fall outside the query region) and the upper bound is the combined count of both contained/partially overlapping nodes (if all points happen to fall within the query region). Thus the interval of confidence is:

$$I^Q_{count} = [count^c, count^c + count^p]$$

Estimation.— We define the percentage of overlap of node N with query Q as $P_N = \frac{A_{R^N \cap R^Q}}{A_{R^N}}$. If we assume that data points are distributed uniformly within R^N then we expect that $P_N \times count^N$ points from that node to be within the query region. Thus, our estimate will be:

$$\widehat{count^Q} = count^c + \sum_{N \in \mathcal{N}^p} P_N \times count^N$$

Traversal Policy.— A simple traversal policy is based on the potential maximum contribution of a node to the length of I^Q_{count}. Each node $N \in \mathcal{N}^p$ contributes exactly $count^N$ to the length of this interval. Therefore we choose the best node to expand at each stage of the algorithm as the one with the highest $COUNT$:

$$N^{COUNT}_b \in \mathcal{N}^p : \forall N \in \mathcal{N}^p : count^{N^{COUNT}_b} \geq count^N$$

It is again easy to implement this traversal policy using a priority queue of nodes in descending $COUNT$ order. Unlike the previous case, none of these nodes can be discarded if an exact answer is to be achieved. For a leaf node we scan the data points it contains and add them to $count^c$ if they happen to be inside the query region.

5.3 Sum

We want to find $sum^{Q,X}$ for the query region. This is the sum of the values of attribute X for all points contained in R^Q.

Interval of Confidence.— Using the same rationale as above, this can be as low as $sum^{c,X}$ and as high as $sum^{c,X} + \sum_{N \in \mathcal{N}^p} sum^{N,X}$. Thus the interval of values is

$$I^{Q,X}_{sum} = [sum^{c,X}, sum^{c,X} + sum^{p,X}]$$

Estimation.— The estimation given is based on the assumption that for node N we expect $P_N \times count^N$ data points to be in the query region, and to have on average the same value as the node's average X attribute value or, $avg^{N,X} = \frac{sum^{N,X}}{count^N}$. Based on the above:

$$\widehat{sum^{Q,X}} = sum^{c,X} + \sum_{N \in \mathcal{N}^p} P_N \times sum^{N,X}$$

Traversal Policy.— The traversal policy is similar to the policy for $COUNT$. Each node N again contributes $sumN, X$ to the length of the interval $I^{Q,X}_{sum}$. The best node to expand will be:

$$N^{SUM}_b \in \mathcal{N}^p : \forall N \in \mathcal{N}^p : sum^{N^{SUM}_b,X} \geq sum^{N,X}$$

5.4 Average

We want to find $avg^{Q,X} = \frac{sum^{Q,X}}{count^Q}$. The estimation for this case is simple, but the interval of confidence and traversal of policy are complicated by the fact that the AVG is the fraction of two uncertain, mutually dependent quantities.

Estimation.— The estimation under the spatial/value uniformity assumption is easy to come up with and it can be expressed as a function of the estimates for $COUNT$ and SUM previously given:

$$\widehat{avg^{Q,X}} = \frac{\widehat{sum^{Q,X}}}{\widehat{count^Q}}$$

Interval of Confidence.— The problem can be stated as follows: given sets $\mathcal{N}^p, \mathcal{N}^c$ find legal possible distributions $\mathcal{D}_{low}, \mathcal{D}_{high}$ of data points to the nodes of these sets, such that $avg^{Q,X}$ is maximum/minimum. The interval will then be:

$$I^{Q,X}_{avg} = [avg^{Q,X}_{\mathcal{D}_{low}}, avg^{Q,X}_{\mathcal{D}_{high}}]$$

```
(1)  AVG-Upper-Bound(N^p:set of Node, N^c:set of Node):Real {
(2)    UpperBoundSet := N^c;
(3)    forall N in N^p calculate h^N_MAX, h^N_MIN, δ; /* See Lemma 1 */
(4)    sort (h^N_MAX, max^N), (h^N_MIN, min^N), (1, δ) on max^N, min^N, δ;
(5)    examine each (number, value) of sorted list in descending order {
(6)      if value > avg^UpperBoundSet
(7)        insert number points with value value into UpperBoundSet;
(8)      else
(9)        return avg^UpperBoundSet;
(10)   }
(11)   return avg^UpperBoundSet;
(12) }

(13) AVG-Lower-Bound(N^p:set of Node, N^c:set of Node):Real {
(14)   LowerBoundSet := N^c;
(15)   forall N in N^p calculate l^N_MAX, l^N_MIN, δ; /* See Lemma 2 */
(17)   sort (l^N_MAX, max^N), (l^N_MIN, min^N), (1, δ) on max^N, min^N, δ;
(18)   examine each (number, value) of sorted list in ascending order {
(19)     if value < avg^LowerBoundSet
(20)       insert number points with value value into LowerBoundSet;
(21)     else
(22)       return avg^LowerBoundSet;
(23)   }
(24)   return avg^LowerBoundSet;
(25) }
```

Figure 5: Calculating the AVG bounds

The distributions that provide these bounds are computed algorithmically as in the algorithms of Figure 5. The algorithm makes use of the results of Lemmas 1, 2:

LEMMA 1. *A node N with aggregates min^N, max^N, $count^N$, sum^N can have as many as $h^N_{MAX} = \lfloor \frac{sum^N - count^N min^N}{max^N - min^N} \rfloor$ data points with max^N values. In that case there can be as many as $h^N_{MIN} = count^N - n^N_{MAX} - \epsilon, \epsilon \in \{0,1\}$ points with min^N values and ϵ points with value $\delta = sum^N - h^N_{MAX} max^N - (count^N - h^N_{MAX} - 1) min^N$.*

Proof:
If there are k values equal to max^N then there would be $count^N - k$ data points left over, with a sum of $sum^N - k \times max^N$. The average of these data points would be $\frac{sum^N - k \times max^N}{count^N - k}$. If k is legal, it must be that $\frac{sum^N - k \times max^N}{count^N - k} \geq min^N \Leftrightarrow k \leq \frac{sum^N - count^N min^N}{max^N - min^N}$. Thus, we have proven that at most there will be $h^N_{MAX} = \lfloor \frac{sum^N - count^N min^N}{max^N - min^N} \rfloor$ data points with max^N values. If $\frac{sum^N - h^N_{MAX} max^N}{count^N - h^N_{MAX}} = min^N$ then the average of $count^N - h^N_{MAX}$ is min^N therefore all the $count^N - h^N_{MAX}$ points have value min^N. If $\frac{sum^N - h^N_{MAX} max^N}{count^N - h^N_{MAX}} > min^N$ there can be at most $count^N - h^N_{MAX} - 1$ such points (clearly if all $count^N - h^N_{MAX}$ had the value min^N then the average would also have to be min^N). To have $count^N$ in total, there must be an additional point with value $\delta = sum^N - h^N_{MAX} max^N - (count^N - h^N_{MAX} - 1) min^N$. □

LEMMA 2. *A node N as in Lemma 1 can have as many as $l^N_{MIN} = \lfloor \frac{count^N max^N - sum^N}{max^N - min^N} \rfloor$ data points with min^N values. In that case there can be as many as $l^N_{MAX} = count^N - l^N_{MIN} - \epsilon, \epsilon \in \{0,1\}$ points with max^N values and ϵ points with value $\delta = sum^N - l^N_{MIN} min^N - (count^N - l^N_{MIN} - 1) max^N$.*

Proof:
Similar to above by positing that k' values min^N are in node N and stating that $\frac{sum^N - k' min^N}{count^N - k'} \leq max^N$.

We calculate $h^N_{MAX}, h^N_{MIN}, \delta$ for each partially overlapping node (line 3) and sort the pairs (number, value) in descending order (line 4). For instance if a node has $min^N = 10, max^N = 50, count^N = 3, sum^N = 100$ then $h^N_{MAX} = 1, h^N_{MIN} = 1, \delta = 100 - 50 - 10 = 40$ and this node contributes pairs $(1, 100), (1, 40), (1, 10)$ We visit these pairs for all the nodes in decreasing order of value (line 5) and attempt to add them to our UpperBoundSet (lines 6-7); this corresponds to picking only the high values to go into the query region because we want the upper bound on the average. The algorithm stops when the average would not increase any further by adding the data points of a (number, value) pair (lines 8-9). A symmetric approach is used to compute the lower bound of the average. The correctness of this algorithm is explained in [9].

Traversal Policy.— A simple traversal policy would be based on a node's $COUNT$. As $count^c$ (sum of $COUNT$ of contained nodes) increases, any given partially overlapping node can minimally perturb the established average. An alternative policy would keep nodes in $\mathcal{N}^p$ sorted in two queues: *minqueue* in ascending order based on $min^{N,X}$ and *maxqueue* in descending order based on $max^{N,X}$. Nodes are popped out of either queue. The choice of queue is based on the greatest distance from the current average by performing the test of $avg^{c,X} - min^{N,X}$ against $max^{N,X} - avg^{c,X}$. Nodes with exceptionally high MAX or low MIN are more likely to increase or decrease the average. We can maintain the two queues by having them point at a single element per node which is invalidated once it is dequeued; thus, if it is dequeued again from the other queue it will be ignored.

6. EXPERIMENTS

6.1 MRA-quadtree

We tested our algorithm using an MRA-quadtree. This type of data structure is often used in spatial applications involving point data. The first test was done with synthetic data. We generate a dataset with 5×10^6 data points in the following manner: we create 1000 clusters each containing a number (taken from a normal distribution ($\mu = 5000, \sigma = 1000$)) of points. Each cluster spans 10% of the $[0,1]^2$ space on each dimension. Its centroid is uniformly distributed around the space. Each point's value is randomly taken from distribution ($\mu = 100, \sigma = 50$) and only in the range $[0, 200]$ (to avoid negative values).

The quadtree we use has $n_{leaf} = 64$. Each leaf node splits when it exceeds n_{leaf}, creating 4 new leaf nodes as its children. We observed that the minimum/maximum depth of the tree (distance from root node to any leaf node) was 6/11. This indicates that the spatial distribution is clearly non-uniform.

We generate query sets of 200 queries, with centroids uniformly distributed in space at spatial selectivities 1%, 2%, 5%, 10%, 25% and with query shapes such that one of the sides of the query rectangle is drawn from the normal distribution ($\mu = \sqrt{sel}, \sigma = 0.5\sqrt{sel}$). This is done so as to produce queries of various aspect ratios. For each query shape we ask 5 queries, one for each of the five aggregates. We keep track of the number of nodes of the quadtree visited to get the exact answer and the number of nodes that intersect with the query; a simple quadtree scan would have to visit all nodes that overlap with the query. The results

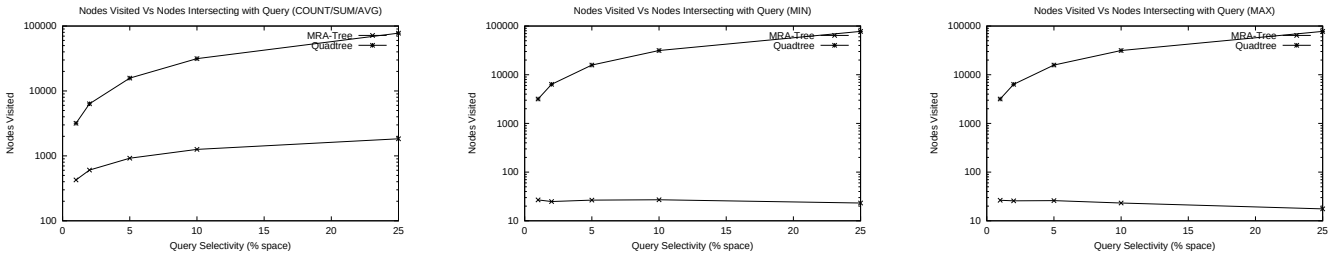

Figure 6: Nodes Visited (Exact Answer) Vs. Nodes Intersecting with Query (Synthetic Data)

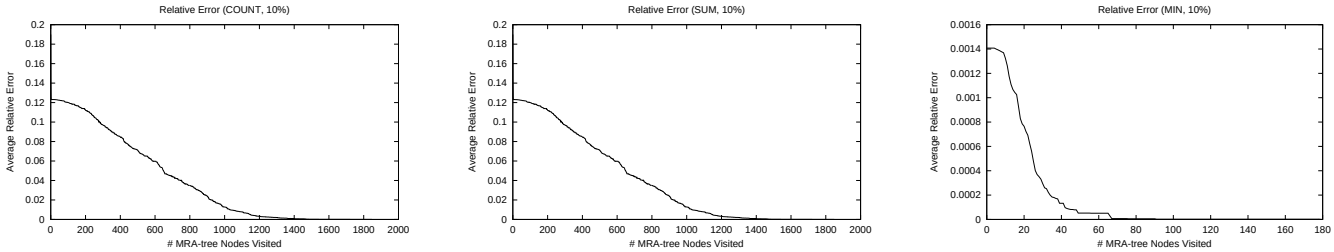

Figure 7: Relative error of estimate for successive node expansions (10% sel. Synthetic Data)

are shown in Figure 6. As the query selectivity increases the MRA-tree visits a decreasing portion of the nodes overlapping with the query. This is caused by the fact that only the border of the query is explored towards the leaf level. The same effect is observed as the database size increases because the quadtree becomes deeper and more work is saved in the interior of the query (corresponding to $\mathcal{N}^c$). For MIN and MAX the actual number of nodes visited decreases as selectivity increases. This is due to the fact that as the size of the interior becomes greater comparatively to the perimeter of the query, the probability that the minimum will be found there (i.e. in $\mathcal{N}^c$) increases.

Significantly fewer nodes have to be visited for an approximate answer. In Figure 7 we plot how the relative error, defined as $\frac{|agg^{Q,X} - \widehat{agg^{Q,X}}|}{Max(1, agg^{Q,X})}$ decreases over successive node expansions.

We also evaluated the MRA-tree strategy on real data taken from the US Geological Survey. Our dataset contained longitude/latitude measurements which we used as spatial dimensions and a variety of attributes; we chose the "elevation" attribute because most other attributes had many NULL values. The dataset had a total of 137,093 data points. We set the leaf-node capacity to 16 in order to get a tree deep enough for a proper evaluation. The quadtree had a minimum/maximum depth of 3/15 in this case. The results are shown in Figures 8,9 and are similar to the ones reported for synthetic data. The error in this case is more significant in initial iterations but decreases much faster due to the highly non-uniform nature of the data.

The time taken to visit a certain number of nodes is linear/close to linear to the number of nodes visited both in the Approximate Aggregate Query approach (AAQ) and in a plain quadtree-based approach (with no aggregate information stored at the tree nodes). However, because of the added complexity of maintaining the priority queue, more CPU time will be spent per quadtree node than in the plain quadtree case (in which we just perform an overlap test for each node). However, this is offset by the significant reduction in number of nodes that we end up visiting. Our experiments indicate that in terms of wall-clock time AAQ outperforms plain quadtree based approach even for exact answering; results are included in [9].

An important thing to note is that for SUM, COUNT and AVG queries, for which every node that intersects with the query affects the outcome, if we decide to get the exact answer then the cost of maintaining the priority queue can be eliminated (since all partially overlapping nodes have to be visited eventually). This will serve to improve CPU performance even more for exact aggregate queries. In MIN, MAX queries the algorithm typically finds the exact answer much faster than the other methods.

6.2 MRA-RTree

In our next set of experiments we used an R-Tree data partitioning structure. Since R-Trees are secondary memory data structures with each node corresponding to a disk page, we were able to measure the I/O performance of the MRA-RTree which is the main measure of performance for large databases in which approximate aggregation makes sense. A number of parameters, and their impact on performance were investigated: database size, dimensionality, dataset characteristics. We conducted our experiments for $COUNT$ queries. These (along with SUM and AVG queries) require for the aggregate information of all partially overlapping nodes to be examined — if a precise answer is needed. MIN and MAX queries, as we showed in the previous section, are evaluated at a fraction of the number of nodes of the $COUNT$ case, i.e., significantly faster.

We used both synthetic data sets and real-life datasets to conduct our experiments. The results did not vary significantly over all datasets examined. A primary observation is that the quality of the estimation increases as the spatial uniformity of the dataset increases. This is understandable since the estimation itself is based on this assumption. If the data is non-uniform then the error in the initial iterations of the algorithm is significant but drops much faster than the

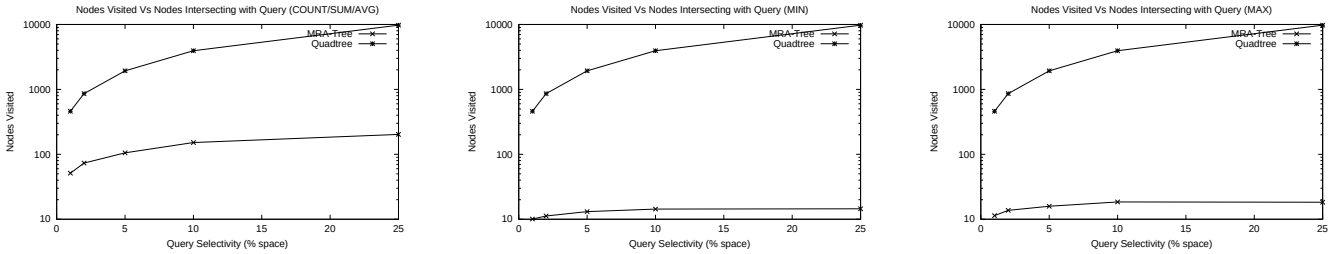

Figure 8: Nodes Visited (Exact Answer) Vs. Nodes Intersecting with Query (Real Data)

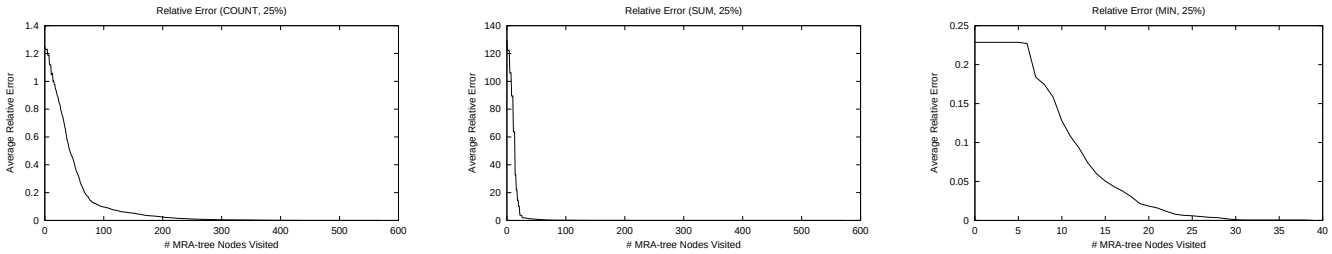

Figure 9: Relative error of estimate for successive node expansions (25% sel. Real Data)

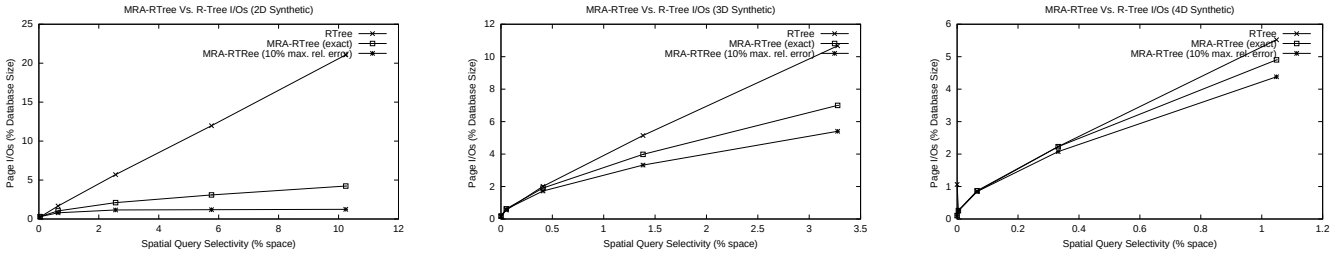

Figure 10: I/O Performance Vs. Query Selectivity (Synthetic Data)

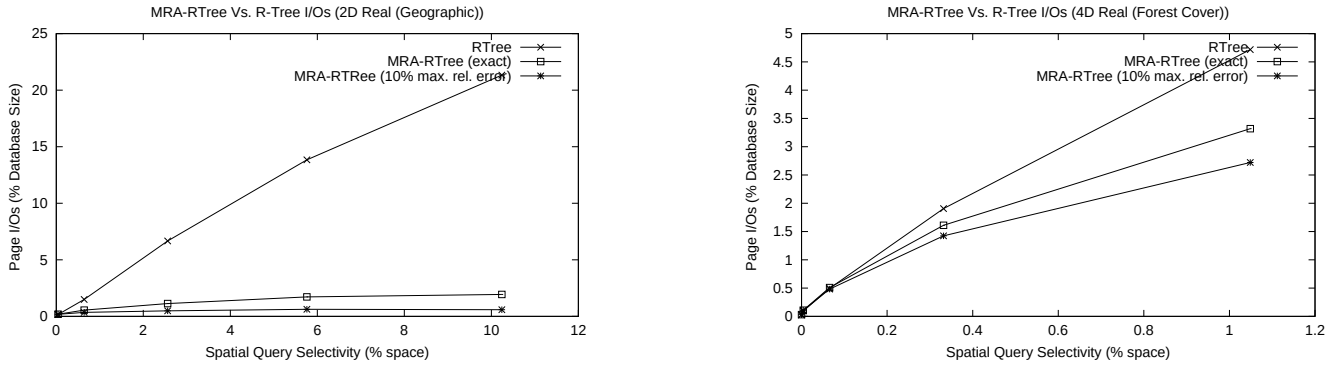

Figure 11: I/O Performance Vs. Query Selectivity (Real Data)

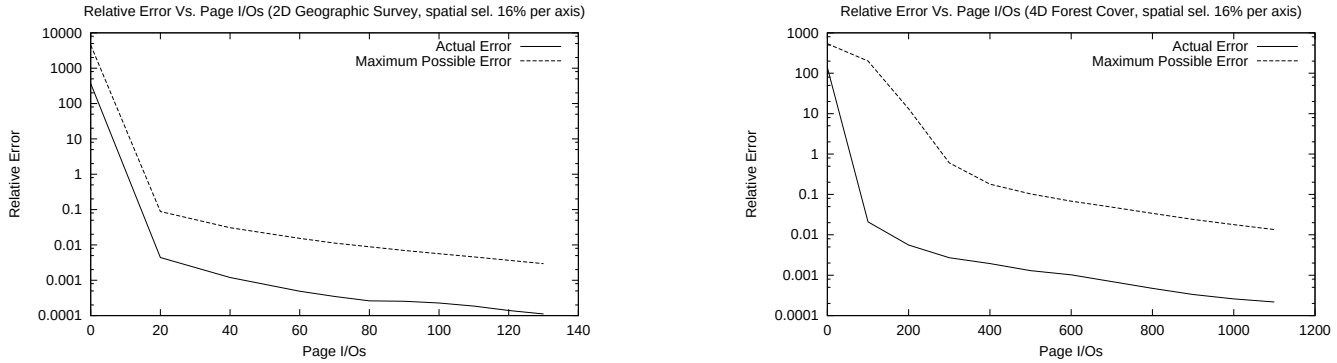

Figure 12: Absolute Relative Error Vs. Page I/Os (Real Data)

previous case. This is because the probability that a high-$COUNT$ node falling either inside or outside the query (the only two cases which shrink I^{COUNT}) increases.

Our synthetic data was generated in the fashion already mentioned. In the reported experiments, we used a 1.2×10^6-point database with 2400 regions each ranging 10% along each dimension. We also report results obtained for two real data sets. The first one is the US Geological Survey used in the MRA-quadtree experiments. The second one is a multi-dimensional Forest Cover dataset taken from the KDD archive at UCI; this was also used in [4]. We chose 4 of its real-valued attributes for our experiments. There are 581,012 data points in this data set.

We set the page size to 128 numbers and calculated the number of children records for each non-leaf node and the number of data points for each leaf node. If the dimensionality is d then each child record must contain $2d+5$ numbers (a d-dimensional hyper-rectangle, 4 aggregate values and 1 pointer to the child node). In a plain R-Tree $2d+1$ numbers are needed per child record. Leaf-level nodes need $d+1$ numbers for each data point (d for loc and 1 for the attribute value). In reality, fewer than $2d+5$ numbers might be necessary for the MRA-RTree if queries are not asked for all 5 SQL-Type aggregates. In our experiments we have seen that keeping all the aggregates does not significantly affect performance especially at high dimensionalities, because the fraction of space they use $\frac{4}{2d+5}$ decreases. Conversely, when the dimensionality is low, the space overhead is more pronounced but the likelihood that a tree node is contained in the query increases (more nodes partially overlap with queries in high dimensionality). Since we avoid (in MRA-RTree) to visit such nodes that partially overlap with the query, this benefit is more apparent in low dimensionalities.

The cardinality of the non-leaf nodes and leaf-nodes for the dimensionalities we investigated are given below:

	2D	3D	4D
R-Tree fanout: n_{R-Tree}	25	18	14
MRA-RTree fanout: $n_{MRA-RTree}$	14	11	9
leaf node (# of points): n_{leaf}	42	32	25

We assume that all 4 aggregate values are stored in the non-leaf nodes. The MRA-RTree created in our experiments, had a height of 7 for all 3D and 4D datasets, a height of 6 for the synthetic 2D dataset and a height of 5 for the small Geological Survey dataset. The plain R-Tree, due to its increased fanout, had a height 1 less than the above values for all our datasets.

The first set of experiments investigates the performance of the data structure over 2D, 3D, 4D synthetic data and our two real data sets for various query selectivities. The results are reported in Figures 10,11. We run sets of 500 queries at each selectivity level. We used queries of size 2%, 8%, 16%, 24% and 32% along each dimension, employing the same technique as in the MRA-quadtree experiments to distribute the aspect ratio of the queries while keeping the overall spatial selectivity (which, for dimensionality d and fraction f along each axis is f^d) constant. We plot the number of page I/Os required on average to answer the query load at each selectivity level for R-Tree, MRA-RTree (exact answer) and MRA-RTree, if we are willing to tolerate a maximum relative error of 10%. Instead of plotting n_{IO}, the absolute number of I/Os, we plot (as a percentage) $\frac{n_{IO}}{n_{DB}}$ where n_{DB} is the number of full data pages that are required to store all points in the database. If there are N points in the database in total then $n_{DB} = \frac{N}{n_{leaf}}$. Roughly speaking, all the database could be examined using a linear scan at the time it takes to do $0.1 n_{DB}$ random I/Os. In all our experiments the MRA-RTree found the exact answer to the queries (on the average) much faster than this figure.

For all dimensionalities and query selectivities that we examined, the performance of MRA-RTree was better than a plain R-Tree. Only in 3D and 4D and for very small query selectivity were we able, using an R-Tree, to arrive at an exact answer faster than with an MRA-RTree. This is explained by the fact that small selectivity queries (prevalent in high dimensional spaces) usually return very small $COUNT$ answers, therefore aggregation of the form we are suggesting is not really useful. However even in 4D, for the greater range of selectivities we investigated, MRA-RTree was still able to find the exact answer faster than a plain R-Tree scan. The number of I/Os required to find an answer with a 10% maximum relative error is significantly low in low dimensionalities. In higher dimensionalities the figure gets closer to the number of I/Os needed to get the exact answer. This is reasonable because many more nodes now partially overlap or contain the query. Moreover the query answer (which is in the denominator of the relative error expression) decreases, leading to higher relative errors. Overall, we observe a noticeable improvement of I/O performance if an error of the estimation is tolerated.

In the next set of experiments we wanted to investigate the accuracy of the estimation and how it decreases with

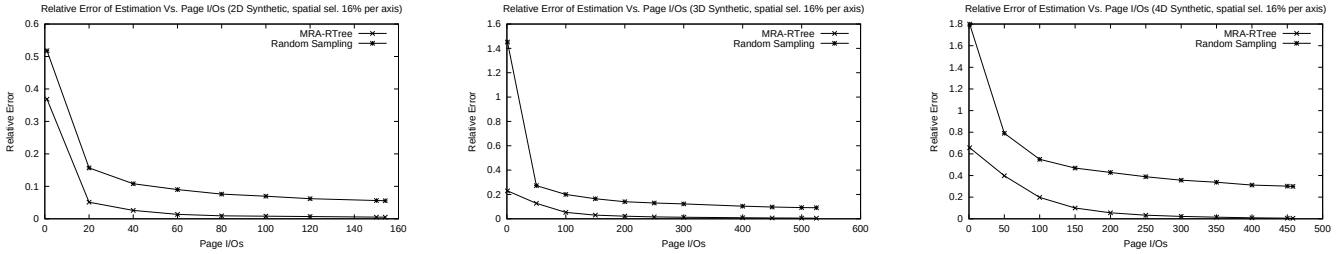

Figure 13: Absolute Relative Error Vs. Number of I/Os for MRA-RTree and Sampling (Synthetic Data)

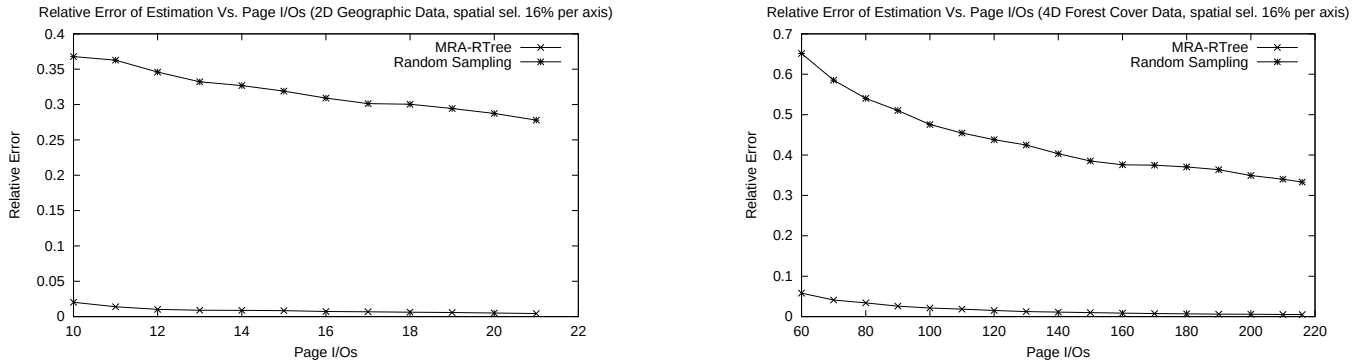

Figure 14: Absolute Relative Error Vs. Number of I/Os for MRA-RTree and Sampling (Real Data)

successive I/Os. The results are shown in Figure 12. We plot once more the average absolute relative error of the estimation (for the query load) as well as the average maximum possible relative error. To conserve space we only report the results for spatial query selectivity 16% along each dimension and for our real data sets. If a is the answer to the query and $\widehat{a}$ is our estimate and $I^{COUNT} = [l, h]$ is the interval of confidence, then the relative error of the estimation is $\frac{|a-\widehat{a}|}{Max(1,a)}$ while the maximum possible relative error is $Max(\frac{\widehat{a}-l}{Max(1,l)}, \frac{h-\widehat{a}}{Max(1,h)})$. In the logarithmic scale of the graph, it is apparent that the actual error is approximately two orders of magnitude less than the maximum error throughout the run of the algorithm. This is both a drawback and an advantage. It means, on the one hand, that we will usually have much less error than the maximum possible error given by the I^{COUNT} for each iteration of the algorithm. On the other hand it means that I^{COUNT} is wide. Some applications do require such a deterministic interval of confidence. In our current work we examine how we can additionally provide a probabilistic interval of confidence of the sort I_p^{agg} which is interval of confidence I^{agg} for aggregate agg with probability p.

In a third set of experiments we studied the effectiveness of the MRA-RTree for different database sizes. Intuitively, as the database size increases, the performance of MRA-RTree as determined by its $\frac{n_{IO}}{N_{DB}}$ will improve. This is because, for exact answering, the main benefit of the algorithm over a plain R-Tree scan is derived from nodes that are contained in the query region. As the size of the database increases, the subtree rooted at each such node deepens, containing more nodes. Thus, this benefit is realized. In Figure 15 we ask a set of 10% special selectivity queries over a 3D synthetic databases of varying size from 20,000 all the way to 1,280,000

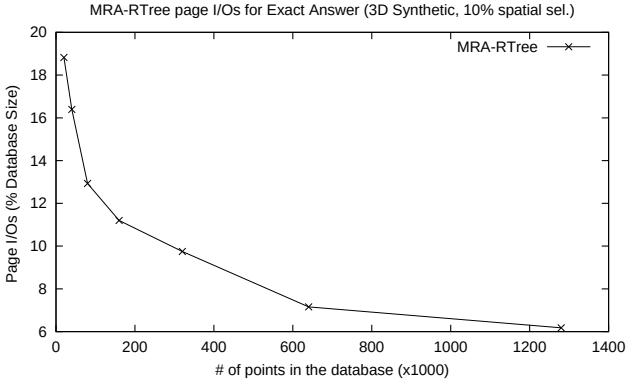

Figure 15: I/O Performance Vs. Database Size

points. We see that the number of I/Os required for an exact answer (as a fraction of the database size) initially drops significantly and then tends to approach some limit. This is due to the fact that the leaf nodes (and their antescendants) at the perimeter of the query have to be visited, hence for some query selectivity, a fraction of the nodes must always be visited no matter how deep the tree is.

Finally, we compare MRA-RTree with on-line random sampling. Our technique is not directly comparable with the off-line estimation techniques, since these approximate the data only at a fixed maximum resolution. We measure the absolute relative estimation error for different number of I/Os. Random sampling is done as follows: a fraction f of the pages in the database are read and $count_s$ is calculated on the sample. The estimate is then: $\widehat{count} = \frac{count_s}{f}$. If data is placed on disk randomly then on-line random sampling

can be performed with sequential I/Os that are faster than the random I/Os of the MRA-RTree technique. As noted in [6], most often data is placed on disk in some order, hence they must be read randomly to create a random sample. We assume that this is the case in our experiments, and so we can compare the accuracy of the two techniques for a given number of I/Os. The results are shown in Figures 13, 14. The right-hand side cutoff of the graphs is the point in which MRA-RTree has less than 0.5% error. We conclude that MRA-RTree provides a much better estimation than could be achieved with random sampling.

7. CONCLUSIONS

We have presented a new data structure, Multi-Resulution Aggregate tree for answering aggregate queries over point data. The MRA-tree is a modified tree index storing aggregate information about data at successive levels of resolution as defined by the space partitioning/data grouping hierarchy. We have shown how a progressive algorithm can iteratively give increasing quality answers until some error bound is satisfied or timing constraint is reached. The proposed approach can be used for any of the typical SQL aggregates and produces answers within 100% guaranteed bounds.

Our experiments indicate that the method can be used effectively to answer aggregate queries. For exact answering, MRA-tree outperforms both a simple index scan or a linear scan of the database. The performance improvement over these methods becomes more pronounced with an increase of either query selectivity or database size. As dimensionality increases, the performance of MRA-RTree worsens because of the deterioration of R-Tree performance in high dimensionality. In $4D$ which is the highest dimensionality we tested for, the performance of MRA-RTree is still satisfactory. In the future we will test MRA-RTree with the specialized index structures for high-dimensional data that have been been proposed in the past.

The quality of the estimation is generally good even after only a few iterations. It is significantly better than the estimation possible using simple on-line random sampling. It is often possible to receive a good guaranteed-quality answer after only a small fraction of the nodes required for the exact answer have been visited. Usually the actual estimation error is much less than the maximum possible error.

In the future we plan to incorporate MRA-tree in a virtual reality application. Such an environment has strict timing deadlines in order to maintain a high-frame rate and MRA-tree will be used to answer aggregate queries over the space as the user navigates through the environment. We also plan to use MRA-tree in conjunction with a mining/visualization tool and for OLAP applications in general. Our work is only an aspect of the general problem of *quality-aware database management systems*. Such systems will tackle the problems of high-update rates, large database sizes and expensive query processing introduced by novel applications, utilizing new ways to quantify imprecision in data and methods for query-answering at various levels of quality.

8. REFERENCES

[1] P. M. Aoki. How to avoid building datablades (r) that know the value of everything and the cost of nothing. In *11th International Conference on Scientific and Statistical Database Management, Proceedings, Cleveland, Ohio, USA, 28-30 July, 1999.*

[2] D. Barbará and X. Wu. Supporting online queries in rolap. In *Proceedings of the Second Int'l Conference on Data Warehousing and Knowledge Discovery, DaWaK 2000, London, UK, September 4-6, 2000*, pages 234–243.

[3] K. Chakrabarti, M. N. Garofalakis, R. Rastogi, and K. Shim. Approximate query processing using wavelets. In *VLDB 2000, Proceedings of 26th International Conference on Very Large Data Bases, September 10-14, 2000, Cairo, Egypt*, pages 111–122.

[4] D. Gunopulos, G. Kollios, V. J. Tsotras, and C. Domeniconi. Approximating multi-dimensional aggregate range queries over real attributes. In *SIGMOD Conference 2000, Dallas, Texas.*, pages 463–474.

[5] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *SIGMOD'84, Proceedings of Annual Meeting, Boston, Massachusetts, June 18-21, 1984*, pages 47–57.

[6] J. M. Hellerstein, P. J. Haas, and H. Wang. Online aggregation. In *SIGMOD 1997, Proceedings ACM SIGMOD International Conference on Management of Data, May 13-15, 1997, Tucson, Arizona, USA*, pages 171–182.

[7] J. M. Hellerstein, J. F. Naughton, and A. Pfeffer. Generalized search trees for database systems. In *VLDB'95, Proceedings of 21th International Conference on Very Large Data Bases, September 11-15, 1995, Zurich, Switzerland*, pages 562–573.

[8] Y. E. Ioannidis and V. Poosala. Histogram-based approximation of set-valued query-answers. In *VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases, September 7-10, 1999, Edinburgh, Scotland, UK*, pages 174–185.

[9] I. Lazaridis and S. Mehrotra. Progressive approximate aggregate queries with a multi-resolution tree structure. Technical Report TR-DB-01-02, Dept. of Information and Computer Science, University of California, Irvine, 2001.

[10] J. T. Robinson. The K-D-B-Tree: A search structure for large multidimensional dynamic indexes. In *Proceedings of the 1981 ACM SIGMOD International Conference on Management of Data, Ann Arbor, Michigan, April 29 - May 1, 1981*, pages 10–18.

[11] N. Roussopoulos, S. Kelley, and F. Vincent. Nearest neighbor queries. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data, San Jose, California, May 22-25, 1995*, pages 71–79.

[12] H. Samet. The quadtree and related hierarchical data structures. *Computing Surveys*, 16(2):187–260, 1984.

[13] J. S. Vitter and M. Wang. Approximate computation of multidimensional aggregates of sparse data using wavelets. In *SIGMOD 1999, Proceedings ACM SIGMOD International Conference on Management of Data, June 1-3, 1999, Philadephia, Pennsylvania, USA*, pages 193–204.