

Improving Min/Max Aggregation over Spatial Objects

Donghui Zhang
Computer Science Department
University of California
Riverside, CA 92521
donghui@cs.ucr.edu

Vassilis J. Tsotras*
Computer Science Department
University of California
Riverside, CA 92521
tsotras@cs.ucr.edu

ABSTRACT

We examine the problem of computing MIN/MAX aggregates over a collection of spatial objects. Each spatial object is associated with a weight (value), for example, the average temperature or rainfall over the area covered by the object. Given a query rectangle, the MIN/MAX problem computes the minimum/maximum weight among all objects intersecting the query rectangle. Traditionally such queries have been performed as range search queries. Assuming that the objects are indexed by a spatial access method, the MIN/MAX is computed as objects are retrieved. This requires effort proportional to the number of objects intersecting the query interval, which may be large. A better approach is to maintain aggregate information among the index nodes of the spatial access method; then various index paths can be eliminated during the range search. In this paper we propose four optimizations that further improve the performance of MIN/MAX queries. Our experiments show that the proposed optimizations offer drastic performance improvement over previous approaches. Moreover, as a by-product of this work we present an optimized version of the MSB-tree, an index that has been proposed for the MIN/MAX computation over 1-dimensional interval objects.

Keywords

spatial aggregates, indexing, Min/Max

1. INTRODUCTION

Computing aggregates over objects with non-zero extents has received a lot of attention recently ([YW01, ZMT+01, PKZ+01, ZTG+01]). Formally, the general *box-aggregation* problem is defined as: “given n weighted rectangular objects and a query rectangle r in the d -dimensional space, find the aggregate weight over all objects that intersect r ”. In this paper we examine the problem of computing the MIN and

*This work was partially supported by NSF grants IIS-9907477, EIA-9983445, and the Department of Defense.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ACM GIS'01 11/01, Atlanta, GA USA

Copyright 2001 ACM 0-58113-235-2/01/0011 ...\$5.00

MAX aggregates (*box-max*) over spatial objects. Each object is represented by its Minimum Bounding Rectangle (MBR) and is associated with a weight (value) that we want to aggregate. A rectangle is also called a *box* and thus the name “box-aggregation”. Since computing the MIN is symmetric, in the following discussion we focus on MAX aggregation. Moreover, we assume that objects are indexed by a spatial access method (SAM) like the R-tree or its variants [Gut84, BKS+90, SRF87].

The box-max problem has many real-life applications. For example, consider a database that keeps track of rainfall over geographic areas. Each area is represented by a 2-dimensional rectangle and a box-max query is: “find the max precipitation in the Los Angeles district”. The database may also keep track of the time intervals of each rainfall, in which case we store 3-dimensional rectangles (one dimension representing the rainfall duration). A box-max query is then: “find the max precipitation in the Los Angeles district during the interval [1999-2000]”.

There have been three approaches towards solving box-aggregation queries. The straightforward approach is to simply perform a range search on the SAM that indexes the objects, and compute the aggregation as objects are retrieved. While readily available, this solution requires effort proportional to the number of objects that intersect the query rectangle, which can be large. Performance is improved if the SAM maintains additional aggregate information ([JL98, LM01, PKZ+01]). For example, the *Aggregation R-tree (aR-tree)* ([PKZ+01]) is an R-tree that stores the aggregate value of each sub-tree in the index record pointing to this subtree. While traversing the index, the aggregation information eliminates various search paths, thus improving query performance.

The third approach uses a specialized aggregate index built explicitly for computing the aggregate in question [YW01, ZMT+01, ZTG+01]. This index maintains the aggregate incrementally. While it is an additional index, it is usually rather compact (since it does not index the actual data but in practice a much smaller representative set) and provides the best query performance.

The main contributions of this paper are:

- We propose four optimizations for improving the MIN/MAX aggregation. One of our optimizations (the k -

max) attempts to eliminate more paths from the index traversal when the aggregate is computed. As such, it can be used either on the SAM that indexes the objects, or, on a specialized aggregate index. The other optimizations (*union*, *box-elimination* and *area-reduction*) eliminate or resize object MBRs when they do not affect the MIN/MAX computation. Thus they apply only to specialized MIN/MAX aggregate indices.

- We present a specialized aggregate index, the *Min/Max R-tree* (MR-tree) that uses all four optimizations. We further present an experimental comparison among a plain R-tree, the aR-tree, the aR-tree with the *k*-max optimization and the MR-tree. Our experiments show drastic improvements when the proposed optimizations are used.
- As a by-product of this research, we discuss how a specialized aggregate index, the MSB-tree [YW00], can be improved by applying the *box-elimination* optimization. The MSB-tree efficiently solves the MIN/MAX problem for the special case of one-dimensional interval objects. The optimization allows the MSB-tree to avoid frequent reconstructions that were needed in its original version.

The rest of the paper is organized as follows. Section 2 discusses related previous work. Section 3 identifies the special characteristics of the box-max problem and presents the optimization techniques. Section 4 summarizes the MR-tree while section 5 presents the results from our experimental comparisons. Section 6 discusses the MSB-tree improvement and section 7 presents conclusions.

2. RELATED WORK

There are two variations of the box-aggregation problem, depending on whether objects have zero extent (*point objects*) or not. Aggregation over point objects is a special case of the orthogonal range searching which has received vast attention in the past 20 years in the field of computational geometry. For more details, we refer to the surveys [Meh84, PS85, Mat94, AE98]. Most of the solutions utilize some variation of the *range-tree* ([Ben80]) following the multi-dimensional divide-and-conquer technique. In the database field, [JL98] proposed the R_a^* -tree which stores aggregated results in the index. [Aok99] proposed to selectively traverse a multi-dimensional index for the problem of selectivity estimation (corresponding to the COUNT aggregate). [LM01] proposed the *Multi-Resolution Aggregate Tree* (MRA-tree) which augments the index records of an R-tree with aggregate information for all the points in the record's sub-tree. The MRA-tree also uses selective traversal to provide an estimate aggregation result. The result can be progressively refined. [JL99] proposes a performance model to estimate the performance of index structures with and without aggregated data.

A special case of the point aggregation problem is the work on data cube aggregation for OLAP applications. A data cube ([GBL+96]) can be thought of as a multi-dimensional array. [RKR97] proposed the *cubetree* as a storage abstraction of the cube and realized it using packed R-trees to efficiently support cube and group-by aggregations. [HAM+97]

addressed both the box-max and the *box-sum* (for SUM, COUNT and AVG) queries over data cubes. The solution to the box-max query was based on storing precomputed max values in a balanced hierarchical structure. This solution was further improved by [HAM+97b]. The solution to the box-sum query was based on pre-computing the *prefix sum*, which is the aggregate over a range covering the smallest cell of the array. This solution was improved by [GAE+99, CI99, GAE00]. Specifically, [GAE00] proposed the *dynamic data cube* which has the best update cost. Recently, [CCL+01] proposes the *dynamic update cube* which further improves the update cost to $O(\log k_u)$ where k_u is the number of changed array cells.

For aggregations over objects with non-zero extents, [YW01] presented the *SB-tree* which solved the box-sum query in the special case of one-dimensional time intervals. The SB-tree was extended to the *Multi-version SB-tree* (MVSB-tree) in [ZMT+01] to efficiently support temporal box-sum aggregation queries with key-range predicates. [ZTG+01] addressed box-sum aggregation over spatiotemporal objects. Furthermore, [YW00] presented the *MSB-tree* for the box-max query over one-dimensional interval objects. The aR-tree ([PKZ+01]) was originally proposed to index the spatial dimension in a spatial data warehouse environment, but can be used to solve both the box-sum and the box-max queries over spatial data with non-zero extent. Similarly with the MRA-tree, the aR-tree is an R-tree which stores for each index record the aggregate value for all objects in its sub-tree. Since the aR-tree was built for the support of both box-sum and box-max queries, it is not fully optimized towards box-max queries. The aR-tree is used here as a starting point for our optimizations and it is included in our experimentation for comparison purposes. Also related is [AS90] which answers window queries on top of the *pyramid data structure*. Aggregations are used for the existence/non-existence of image features and the visibility in terrain data. Last, in the spatial-temporal data warehouse environment, [PKZ+01b] proposed the *aggregate R-B-tree* (aRB-tree) which uses an R-tree to index the spatial dimension and each record r in the R-tree has a pointer to a B-tree which keeps historical data about r .

3. THE PROPOSED OPTIMIZATIONS

In this paper we focus our discussion on the MAX aggregate. The discussion for the MIN aggregates is symmetric and is omitted. Our goal is to solve the box-max problem, where we have a set of objects, each of which has a box and a value; given a query box q , we want to find the maximum value of all objects intersecting q . Assume the objects are indexed by a tree-like structure (e.g. the R-tree) where the objects are stored in leaf nodes and where the MBR of an internal node contains the MBRs of all its children. Using such an index, a box-max query can be answered by performing a range search. In this section we propose four optimizations that improve the performance.

We first introduce some notations. An index/leaf record is an entry in an internal/leaf node of the tree. Given an leaf record r , let $r.box$ and $r.value$ denote the MBR and the value of the record, respectively. Given an index record r , let $r.box$ denote its MBR, $r.value$ denote the maximum value of all records in subtree(r) and $r.child$ denote the child

page pointed by r .

3.1 The k -max optimization

The *aR-tree* is an R-tree where each index record stores the maximum value of all leaf records in the sub-tree. If a query box contains the MBR of an index record, the value stored at the record contributes to the query answer and the examination of the sub-tree is omitted. However, the index records at higher levels of the aR-tree have large MBRs. So the box-max query is not likely to stop at higher levels of the aR-tree. The k -max optimization is an extension that keeps constant number of leaf objects along with each index record such that even if the query box does not contain the MBR of an index record, the examination of the sub-tree may be omitted.

The k -max optimization *Along with each index record r , store the k objects (for a small constant k) which are in $subtree(r)$ and have the largest values among the objects in the subtree. When examining record r during a box-max query, if the query box intersects with any of the k max-value objects in r , the examination of $subtree(r)$ is omitted.*

Clearly, the k -max optimization allows for more paths to be omitted during the index traversal. However, the benefit of k -max on the query performance is not provided for free. The overall space is increased (since each node stores more information) as well as the update time (effort is needed to maintain the k objects). Hence in practice the constant k should be kept small. In our experiments, we found large improvement in query time even for a small $k = 3$.

As pointed out, the next three optimizations apply for an index explicitly maintained for the MIN/MAX aggregation (to avoid confusion we call such an explicit index the MIN/MAX index). Since the MIN/MAX problem is not incrementally maintainable when tuples are deleted from the database [YW01], the following discussion assumes an append-only database (i.e., spatial objects are inserted in the database but never deleted). When a spatial object o with MBR $o.box$ and value $o.value$ is inserted in the database, $o.box$ accompanied by $o.value$ is inserted as a leaf record in the MIN/MAX index as well. However, as we will describe, some of these insertions may not be applied to the MIN/MAX index, or may cause existing MBRs to be deleted or altered from the MIN/MAX index. As such, we can use an R*-tree to implement the MIN/MAX index. The result after applying all four optimizations will be the MR-tree.

3.2 The box-elimination optimization

Consider two leaf records o_1 and o_2 , where $o_1.box$ contains $o_2.box$ and $o_1.value \geq o_2.value$. There is no need to maintain o_2 in the MIN/MAX index since it will not contribute to any MAX query. We thus say that o_2 becomes *obsolete* due to o_1 , or o_1 makes o_2 obsolete.

The box-elimination optimization *If during the insertion of an object o , a (leaf or index) record r is found such that $o.box$ contains $r.box$ and $o.value \geq r.value$, remove r from the MIN/MAX index; if r is an index record, remove $subtree(r)$ as well.*

The above optimization will reduce the size of the MIN/

MAX index, since sub-trees may be removed during an insertion. There is a tradeoff between the time to update the MIN/MAX index and the overall space occupied by this index. A newly inserted object may make obsolete more than one existing records which are on different paths from the root to leaves. The MIN/MAX index can be made very compact if all these obsolete records (and their sub-trees) are removed. However, this may result in expensive update processing. If the update is to be kept fast, we can choose to remove only the obsolete records met along the insertion path (which is a single path since we use a R*-tree to implement the MIN/MAX index). The complexity of the insertion algorithm remains $O(\log(n))$ where n is the number of MBRs in the MIN/MAX index (which in practice is much smaller than the total number of spatial objects in the collection). Another choice is to choose c paths and remove the obsolete records met along these paths, where c is a constant. The space occupied by the obsolete sub-trees can be re-used.

3.3 The union optimization

While the box-elimination optimization focuses on making obsolete existing records in the index, the *union* optimization focuses on making obsolete objects before they are inserted in the MIN/MAX tree. First we note that the MBR of an object should not be inserted in the MIN/MAX index if there is an existing leaf object in the index whose MBR contains it and has a larger value. Such an insertion can be safely ignored for the purposes of MIN/MAX computation. To fully implement this test, all the paths that may contain this object have to be checked; at worst this may check all leaf objects in the MIN/MAX tree. A better heuristic is to use the k max-value MBRs. If the new object is contained by any of the k max-value MBRs found along the index nodes in the insertion path, and has a smaller value, then there is no need to perform the insertion.

Moreover, we observe that even if the MBR of an object to be inserted is not fully contained by any existing leaf object, we still might safely ignore it. This is the case when the new object's MBR is contained in the union of MBRs of several existing objects. As illustrated in figure 1, the shadowed box represents the new object to be inserted and the other two rectangles represent two objects already in the MIN/MAX index. Since the new object is contained in the union of the two existing objects with a smaller value, its insertion can be safely ignored.

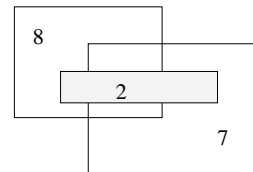


Figure 1: The new object becomes obsolete by the union of two existing objects.

To implement this technique, each index record r in the MIN/MAX index stores (1) the union (denoted by $r.union$) of MBRs of all the leaf objects in $subtree(r)$, and (2) the minimum value (denoted by $r.low$) of all the objects in the $subtree(r)$. The overall optimization is described below:

The union optimization If during the insertion of object o , an index/leaf record r is found such that $o.box$ is covered by $r.union/r.box$ and $o.value \leq r.low/r.value$, the insertion is ignored. Moreover, check whether some max-value object stored in r covers $o.box$ and has a value no smaller than $o.value$; this also makes o obsolete.

A remaining question with the above optimization is how to store the union of all leaf objects under an index record. At worst, this union may need space proportional to the number of leaf objects that create it. Given that each index record has limited space, we store an *approximate* union. In particular, we store a good approximation that can be represented with t boxes (MBRs), where t is a small constant. What is important is that the approximation should affect only the query time, but not the correctness of the query result. If the approximate union covers some area not covered by the actual union, it may erroneously make obsolete some new object. So the approximate union should be completely covered by the actual union. We formally state the problem as follows.

Definition 1. Given constant t and a set of n boxes $S = \{s_1, \dots, s_n\}$ where $n \gg t$. The **covered t -union** of S is defined as a set of t boxes $\{a_1, \dots, a_t\}$ such that (1) $\cup_{i=1}^t s_i$ covers $\cup_{i=1}^t a_i$; and (2) $\cup_{i=1}^t a_i$ is maximal, i.e. there does not exist another set of t boxes $\{a'_1, \dots, a'_t\}$ satisfying the first condition such that $\cup_{i=1}^t a'_i$ covers larger space than $\cup_{i=1}^t a_i$.

To find the exact answer with an exhaustive search algorithm in the two-dimensional case takes $O(n^{8t})$ time, which is clearly unacceptable. So we need to find an efficient algorithm to compute a good approximation of the covered t -union. Again, in order for the box-max query to give correct result, we require that the approximate covered t -union be completely covered by the original n boxes. We propose a $O(n \log n)$ algorithm. The idea is to pick t boxes from the original n boxes and try to expand each one of them as much as possible. To choose the i^{th} box ($1 \leq i \leq t$), choose the one which has the largest area not covered by the $i - 1$ boxes computed so far. To expand a chosen box, try to expand along all directions parallel to the axes. For space limitations the algorithm is omitted but can be found in [ZT01]. In the following discussion the term *union* means the approximate covered t -union.

3.4 The area-reduction optimization

The last optimization we propose dynamically reduces the box area of the object to be inserted.

The area-reduction optimization If during the insertion of object o , an index record r is found such that $r.union$ intersects with $o.box$ and $r.low \geq o.value$, we reduce the size of $o.box$ by subtracting the area covered by $r.union$ from it before inserting it to the lower levels. If the insertion reaches a leaf object which intersects the new object and has an equal or larger value, the area of the new object is reduced accordingly. The box of the new object is similarly reduced if some max-value object stored in an index record r exists whose box intersects with $o.box$ and whose value is no smaller than $o.value$.

This optimization reduces the MBR of an object only if the reduced part is covered by some existing records in the tree with larger or equal values. Hence the correctness of the MIN/MAX aggregates is not affected. One benefit of this optimization is that overlapping among sibling records in the tree is reduced. Figure 2 shows an example. The two large boxes represent two index records r_1 and r_2 . Assume $r_1.union$ is equal to the MBR of r_1 . The combination of the light-shadowed and the dark-shadowed boxes represents an object to be inserted with value 8. The object should be recursively inserted into subtree(r_2). Without applying the area-reduction optimization, $r_2.box$ would need to be expanded to fully contain the new object. On the other hand, if we apply the optimization, the light-shadowed area is subtracted and thus we insert in subtree(r_2) a much smaller box (the dark-shadowed area) which is fully contained in $r_2.box$ and thus no expansion for r_2 is needed.

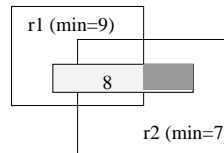


Figure 2: The area-reduction optimization helps to reduce overlaps.

Another benefit of the area-reduction optimization is that it can help to make new records obsolete. As an example, consider figure 2 again. It is possible that at some lower level in subtree(r_2), the dark-shaded area is found obsolete. Since the light-shaded area is already made obsolete by r_1 due to the optimization, there's no need to insert the record at all.

Note that the result of a box when some areas are subtracted from it may be a set of boxes rather than a single box. So an object to be inserted may be fragmented into several smaller boxes by this optimization. One choice to handle this is to follow the R^+ -tree ([SRF87]) approach, i.e. to insert every small box as a separate copy. But this choice increases the space overhead. Another choice is to maintain the list of small boxes in the execution of the insertion algorithm. As we go down the tree, some small boxes may become smaller or obsolete. Eventually at the leaf level, the MBR of the smaller boxes is inserted. Note that the MBR is at most as large as the original box to be inserted and in many cases, much smaller.

4. THE MIN/MAX R^* -TREE

The MR-tree is a dynamic, disk-based, height-balanced tree structure. There are two types of pages: leaf pages and index pages. All pages have the same size. Since the MR-tree is based on the R^* -tree, each page except the root has at most M records and at least m records. Each leaf record has the form $\langle box, v_1 \rangle$, where v_1 is the value of the record. Each index record has the form $\langle box, child; b_1, v_1, \dots, b_k, v_k; union, low \rangle$. Here box and $child$ have their usual meanings. The list $(b_1, v_1), \dots, (b_k, v_k)$ is the k max-value leaf objects in the sub-tree of this index record, sorted by decreasing order of value; b_i stands for the MBR and v_i for the value of the leaf object with the i^{th} largest value. The *union* stores t boxes (the approximate covered t -union over all leaf

MBRs), and *low* is the minimum value over all leaf objects in the sub-tree.

Due to space limitations we only give an outline of the box-max query and update algorithms of the MR-tree. For details, please refer to [ZT01].

To perform a box-max query of query box *b*, we start with the root page and we initialize a running value *current* to be $-\infty$. For every record *r* in the root page where *r.box* intersects *b* and $r.v_1 > current$, if the root page is a leaf page, then set $current = r.v_1$; otherwise, if *b* intersects with any of the *k* max-value objects, set *current* to be the larger one over *current* and the max-value object; otherwise, recursively examine the subtree(*r.child*).

The update algorithm of the MR-tree is similar to that of the R^* -tree. It first follows a constant number of paths from the root down to the leaf level. While browsing down the tree, the box-elimination, the union and the area-reduction optimizations are applied. Reorganizations may follow the paths back towards the root.

5. PERFORMANCE

We compare the performance of the proposed MR-tree against the plain R^* -tree, the aR-tree and the aR-tree with the *k*-max optimization (denoted as $aR\text{-tree}_{kmax}$). All the algorithms were implemented in C++ using GNU compilers. The programs run on a Sun Enterprise 250 Server machine with two 300MHz UltraSPARC-II processors using Solaris 2.8. The page size is 4KB. For space limitations we report the performance of the MR-tree only for $k = t = 3$, where *k* is the number of max-value objects kept in each index record and *t* is the number of boxes used to represent a union. Similarly, the $aR\text{-tree}_{kmax}$ uses $k = 3$. Each index utilizes an LRU buffer and a *path buffer*, which buffers the most recently accessed path. The total memory buffer we used for each program has 256 pages.

Our test dataset contains 5 million square objects randomly selected in a two-dimensional space. The space in both dimensions is [1, 1 million]. The size of each object was randomly chosen from 10 to 10,000.

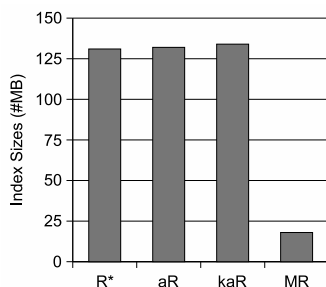


Figure 3: Comparing the index sizes.

Figure 3 compares the index sizes. In the figure we use R^* , *aR*, *kaR*, *MR* to represent the R^* -tree, the aR-tree, the $aR\text{-tree}_{kmax}$ and the proposed MR-tree, respectively. The MR-tree uses the least space, since obsolete records and sub-trees were removed from the index. The $aR\text{-tree}_{kmax}$ occupies the most space, since compared with the R^* -tree and the aR-

tree it stores more information in each index record.

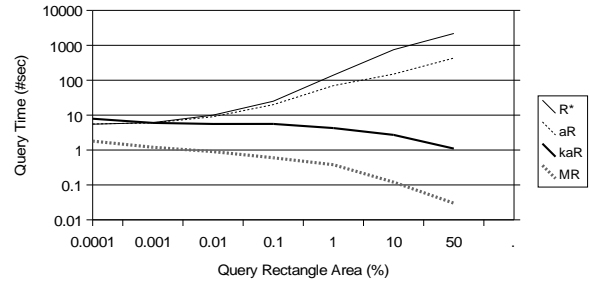


Figure 4: Comparing the query performance.

To evaluate the query performance, the query rectangle area varies from 0.0001% to 50% of the whole space. For each query rectangle size, we randomly generated 100 square queries and measured the total running time. This running time was obtained by multiplying the number of I/Os by the average disk page read access time (10ms), and then adding the measured CPU time. The CPU time was measured by adding the time spent in *user* and *system* mode as returned by the *getrusage* system call. As shown in figure 4, the MR-tree is clearly the best, especially if the query rectangle size is large. Note that the query time scale is logarithmic, so the actual difference in query speeds is drastic (for example, with query size of 1% the MR-tree is about 184 times faster than the aR-tree). The reason is that for large query rectangles, the MIN/MAX query has more chances to stop at higher levels in the MR-tree. In particular, the aR-tree will decide to omit examining a sub-tree only if the query rectangle contains the box of the whole sub-tree. On the other hand, the MR-tree search may omit traversing a sub-tree even if the query rectangle partly intersects with it. The usefulness of the *k*-max optimization can be seen when comparing the $aR\text{-tree}_{kmax}$ with the plain aR-tree. The MR-tree performs better than the $aR\text{-tree}_{kmax}$ for two reasons. First, due to the additional optimizations, the MR-tree stores fewer objects. Second, objects in the MR-tree have smaller area (the area-reduction optimization), and thus achieve better clustering.

We also compared the index generation time. The MR-tree takes about the same CPU time but about one third the I/O time as compared with the other structures. Furthermore, we have experimented with other datasets and got similar results. The comparison graphs are omitted due to space limitations. For more details, we refer to [ZT01].

6. OPTIMIZING THE MSB-TREE

In this section we discuss how the box-elimination optimization can be used to improve the MSB-tree [YW00]. The MSB-tree can answer the box-max query for 1-dimensional interval data. The complexity for both the insertion algorithm and the query algorithm is $O(\log_B m)$, where *B* is the page capacity in number of records, and *m* is the number of leaf records. In order to maintain a small *m* [YW00] proposes to periodically reconstruct the MSB-tree. Such reconstructions “compact” the leaf records thus minimizing *m*. However, during reconstruction, the whole tree is browsed in a depth-first manner while a second, initially empty MSB-tree is created, which eventually replaces the original tree.

While being reconstructed, the MSB-tree is *off-line*, i.e. no new insertion can be made.

Each index record r in the MSB-tree stores an interval $r.i$, a max value $r.u$, a min value $r.v$ and some additional information. To insert an object o with interval $o.i$ and value $o.v$, if $o.i$ contains $r.i$ and $o.v \geq r.u$, the MSB-tree updates both $r.u$ and $r.v$ to be $o.v$ without altering the subtree(r). To apply the box-elimination optimization, if such an object o is inserted, the whole subtree(r) is removed from the index. For review of the MSB-tree and details of the optimized MSB-tree, refer to [ZT01].

The major benefit of the optimized algorithm is that it can result in a much smaller tree without periodic reconstruction. As an example, consider the insertion of object o where $o.i$ is the whole space and $o.v$ is larger than that of every existing record. The original algorithm simply updates the u and v values of all root-level records and thus the number of leaf records does not change. On the other hand, the optimized algorithm immediately decides that the whole tree is obsolete and thus results in a very compact tree: a tree with only one leaf record.

7. CONCLUSIONS

We examined the problem of computing MIN/MAX aggregation queries over spatial objects with non-zero extents. We proposed four optimization techniques for improving the query performance. We introduced the MR-tree, a new index explicitly designed for the maintenance of MIN/MAX aggregates. The MR-tree combines all proposed optimizations. An experimental comparison showed that our approach provides drastic improvement especially when query sizes increase. As a by-product, we showed how one of the optimizations can be applied to improve an existing aggregation index (the MSB-tree).

Acknowledgements

We would like to thank D. Gunopulos for many helpful discussions, D. Papadias for providing us valuable input on related work and B. Seeger for the R*-tree code.

8. REFERENCES

[Aok99] P. M. Aoki, "How to Avoid Building DataBlades that Know the Value of Everything and the Cost of Nothing", *Proc. of SSDBM*, pp. 122-133, 1999.

[AE98] P. Agarwal and J. Erickson, "Geometric Range Searching and Its Relatives", *Advances in Discrete and Computational Geometry*, (B. Chazelle, E. Goodman and R. Pollack eds.), American Mathematical Society, Providence, 1998.

[AS90] W. G. Aref and H. Samet, "Efficient Processing of Window Queries in the Pyramid Data Structure", *Proc. of PODS*, pp. 265-272, 1990.

[Ben80] J. L. Bentley, "Multidimensional Divide-and-Conquer", *Communications of the ACM* 23(4), pp. 214-229, 1980.

[BKS+90] N. Beckmann, H. P. Kriegel, R. Schneider and B. Seeger, "The R*-tree: An Efficient and Robust Access Method for Points and Rectangles", *Proc. of SIGMOD*, pp. 322-331, 1990.

[CCL+01] C. Chung, S. Chun, J. Lee and S. Lee, "Dynamic Update Cube for Range-Sum Queries", *Proc. of VLDB*, 2001.

[CI99] C. Chan, Y. E. Ioannidis, "Hierarchical Cubes for Range-Sum Queries", *Proc. of VLDB*, pp. 675-686, 1999.

[Gut84] A. Guttman, "R-trees: A Dynamic Index Structure for Spatial Searching", *Proc. of SIGMOD*, pp. 47-57, 1984.

[GAE00] S. Geffner, D. Agrawal and A. El Abbadi, "The Dynamic Data Cube", *Proc. of EDBT*, pp. 237-253, 2000.

[GAE+99] S. Geffner, D. Agrawal, A. El Abbadi and T. Smith, "Relative Prefix Sums: An Efficient Approach for Querying Dynamic OLAP Data Cubes", *Proc. of ICDE*, pp. 328-335, 1999.

[GBL+96] J. Gray, A. Bosworth, A. Layman and H. Piramish, "Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Totals", *Proc. of ICDE*, pp. 152-159, 1996.

[HAM+97] C. Ho, R. Agrawal, N. Megiddo and R. Srikant, "Range Queries in OLAP Data Cubes", *Proc. of SIGMOD*, pp. 73-88, 1997.

[HAM+97b] C. T. Ho, R. Agrawal, N. Megiddo and J. J. Tsay, "Techniques for Speeding up Range-Max Queries in OLAP Data Cubes", *IBM Research Report*, 1997. URL=<http://www.almaden.ibm.com/cs/quest/PUBS.html>

[JL98] M. Jürgens and H. J. Lenz, "The R_a^* -tree: An Improved R-tree with Materialized Data for Supporting Range Queries on OLAP-Data", *DEXA Workshop*, 1998.

[JL99] M. Jürgens and H.-J. Lenz, "PISA: Performance Models for Index Structures with and without Aggregated Data", *Proc. of SSDBM*, pp. 78-87, 1999.

[LM01] I. Lazaridis and S. Mehrotra, "Progressive Approximate Aggregate Queries with a Multi-Resolution Tree Structure", *Proc. of SIGMOD*, 2001.

[Mat94] J. Matoušek, "Geometric Range Searching", *Computing Surveys* 26(4), pp. 422-461, 1994.

[Meh84] K. Mehlhorn, "Multi-dimensional Searching and Computational geometry", *Data Structures and Algorithms*, vol. 3, Springer-Verlag, Heidelberg, West Germany, 1984.

[PKZ+01] D. Papadias, P. Kalnis, J. Zhang and Y. Tao, "Efficient OLAP Operations in Spatial Data Warehouses", *Proc. of SSTD*, 2001.

[PKZ+01b] D. Papadias, P. Kalnis, J. Zhang and Y. Tao, "Indexing Spatio-Temporal Data Warehouses", *TechReport HKUST-CS01-20, CS Dept., Univ. of Sci. and Tech., Hong Kong*, 2001. URL=<http://www.cs.ust.hk/~dimitris/publications.html>

[PS85] F. Preparata and M. Shamos, *Computational Geometry: An Introduction*, Springer-Verlag, Berlin/Heidelberg, Germany, 1985.

[RKR97] N. Roussopoulos, Y. Kotidis and M. Roussopoulos, "Cube-tree: Organization of and Bulk Incremental Updates on the Data Cube", *Proc. of SIGMOD*, pp. 89-99, 1997.

[SRF87] T. K. Sellis, N. Roussopoulos and C. Faloutsos, "The R+-Tree: A Dynamic Index for Multi-dimensional Objects", *Proc. of VLDB*, pp. 507-518, 1987.

[YW00] J. Yang and J. Widom, "Incremental Computation and Maintenance of Temporal Aggregates" (full version), *TechReport, Stanford University*, 2000. URL=<http://www.cs.duke.edu/~junyang/research.html#pubs>

[YW01] J. Yang and J. Widom, "Incremental Computation and Maintenance of Temporal Aggregates", *Proc. of ICDE*, pp. 51-60, 2001.

[ZMT+01] D. Zhang, A. Markowetz, V. J. Tsotras, D. Gunopulos and B. Seeger, "Efficient Computation of Temporal Aggregates with Range Predicates", *Proc. of PODS*, 2001.

[ZT01] D. Zhang and V. J. Tsotras, "Improving Min/Max Aggregation over Spatial Objects", *TechReport UCR-CS.01.03, CS Dept., UC Riverside*, 2001. URL=<http://www.cs.ucr.edu/~donghui/publications/boxmax.full.ps>

[ZTG+01] D. Zhang, V. J. Tsotras, D. Gunopulos and A. Markowetz, "Efficient Aggregation over Objects with Extent", *TechReport UCR-CS.01.01, CS Dept., UC Riverside*, 2001. URL=<http://www.cs.ucr.edu/~donghui/publications/boxaggr.ps>