Improving Min/Max Aggregation over Spatial Objects

Donghui Zhang* Vassilis J. Tsotras†

Abstract

We examine the problem of computing MIN/MAX aggregate queries 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 a dynamic version of the MSB-tree, an index that has been proposed for the MIN/MAX computation over 1-dimensional interval objects.

*CONTACT AUTHOR, Computer Science Department, University of California, Riverside, CA 92521. donghui@cs.ucr.edu

†Computer Science Department, University of California, Riverside, CA 92521, tsotras@cs.ucr.edu. This work has ben supported by NSF grants IIS-9907477, EIA-9983445 and by the Dept. of Defense.

Improving Min/Max Aggregation over Spatial Objects

Donghui Zhang Vassilis J. Tsotras

1 Introduction

Computing aggregates over objects with non-zero extents has received a lot of attention recently ([YW01, ZMT+01, PKZ+01, ZTM+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 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. Since computing the MIN is symmetric, in the following discussion we focus on MAX aggregation. Morover, 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-aggregate 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 r, 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 sub-tree. 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, ZTM+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 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 [YW01], can become dynamic by applying the *box-elimination* optimization. The MSB-tree efficiently solves the MIN/MAX problem for the special case of one-dimensional interval objects. Its original version is not fully dynamic, since it needs periodically to be reconstructed.

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. Finally, section 6 provides conclusions and problems for further research.

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 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 mult-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.

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 (for MIN and 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.

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. [ZTM+01] addressed box-sum aggregation over spatiotemporal objects. Furthermore, ([YW01]) 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. 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

We proceed with a discussion of the proposed optimizations. Given an index record r, let *r.box* denote its MBR and *r.value* denote the max value of all records in subtree(r). For example, the aR-tree [PKZ+01] stores one *r.value* inside each index record. Then if a query rectangle contains the MBR of index record r, the examination of the subtree(r) is avoided. To justify the use of the $k - max$ optimization we note that at higher levels of the aR-tree, the index records have large MBRs. So the MIN/MAX query is not likely to stop at higher levels of the tree. The $k - max$ optimization is an extension that keeps the information about the k objects with the higher values under index record r. As long as the query rectangle intersects with the MBR of any max-value object, the subtree(r) can be safely ignored.

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 k larger 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 eliminated 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 values). 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 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 a R*-tree to implement the MIN/MAX index. The result after applying all four optimizations will be the MR-tree.

Consider two leaf objects 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 MIN/MAX query. We thus say that o_2 becomes *obsolete* by o_1.

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 index records which are on different paths from the root to the 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 to large 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.

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 smaller value, then the insertion becomes obsolete.

Moreover, we observe that even if the MBR of an object to be inserted is not fully contained by an 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 values of both the existing objects are no smaller than the value of the new object, its insertion can be safely ignored.

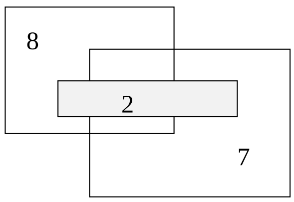


Figure 1: The new object becomes obsolete by the union of two existing objects.

To implement this technique, each index record r in the MIN/MAX index stores (i) the union (denoted by $r.union$) of MBRs of all the leaf objects in the subtree(r), and (ii) 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 the possibility that some max-value object stored in r covers $o.box$ and has a value no smaller than $o.value$, which 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's 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 erroneous 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 $S = \{b_1, \cdots, b_n\}$ where $n >> t$. The* **covered t-union** *of S is defined as a set of t boxes $\{a_1, \cdots, a_t\}$ such that (1) $\bigcup_{i=1}^{n} b_i$ covers $\bigcup_{i=1}^{t} a_i$; and (2) $\bigcup_{i=1}^{t} ab_i$ is maximal (i.e. there does not exist another set of t boxes $\{a'_1, \cdots, a'_t\}$ satisfying the first condition such that $\bigcup_{i=1}^{t} a'_i$ covers larger space.*

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 approxiate covered t-union be completely covered by the original n boxes. We hereby propose a $O(n)$ 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 detailed algorithm is omitted here but can be found in [ZT01]. Unless otherwise stated, in the following discussion the term *union* means the computed approximate covered t-union that is stored in the tree.

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 level. If the insertion reaches a leaf object which intersects the new object and has a no smaller 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 with box intersecting with o.box and value 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. 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-shadowed area is found obsolete. Since the light-shadowed area is obsoleted by r_1 due to the optimization, there's no need to insert the record.

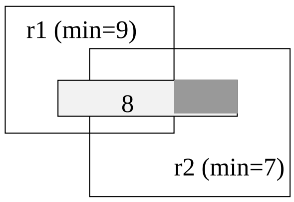


Figure 2: The area-reduction optimization helps to reduce overlaps.

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.

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 it 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 $\langle box,\ child;\ b_1,\ v_1, \cdots,\ b_k,\ v_k;\ union,\ low\rangle$. Here *box* and *child* has their usual meanings. The list $(b_1,\ v_1), \cdots,\ (b_k,\ v_k)$ is the k max-value leaf objects in the sub-tree of this index record, sorted by decreasing order of value. Here 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 t-union over all leaf MBRs), and *low* is the minimum value over all leaf objects in the sub-tree.

Algorithm *BoxMax*(Page N, Box b, Value v)

```
1.  for every record r in N where r.box intersects with b
2.      if r.v_1 > v then
3.          if N is leaf then
4.              v = r.v_1;
5.          else if there exists i in [1, k] such that r.b_i intersects with b
6.              Let i be the smallest one satisfying this condition;
7.              if r.v_i > v, set v = r.v_i;
8.          else
9.              v =BoxMax(Page(r.child), b, v);
10.         endif
11.     endif
12. endfor
13. return v;
```

BoxMax is a straighforward recursive algorithm. To find the box-max over box b, we should call BoxMax(root page, b, -∞). The main difference between this algorithm and the range query algorithm in an R-tree is in steps 5-7, which corresponds to box-elimination optimization. For an index record r, the algorithm checks the k max-value objects stored in r. If any of them intersects with b, there is no need to examine subtree(r).

Algorithm *Insert*(Box b, Value v)

```
S = {b};
N =root page;
while ( N is not leaf ) do
    for every record r in N where r.box intersects with b
        if r.box is contained in b and r.v_1 ≤ v then
            Remove subtree(r);
        else
            for every i such that r.v_i ≤ v
                Modify each box in S by subtracting r.b_i from it;
            endfor
            if r.low ≤ v, modify every box in S by subtracting r.union from it;
```

```
12.         endif
13.     endfor
14.     if N has zero record, goto step 19;
15.     if S is empty, goto step 20;
16.     N =ChooseChild(N, MBB(S)); //MBB(S) is the minimum bounding box of boxes in S
17. endwhile
18. Optimizations for a leaf page; similar to steps 3 through 12; omit;
19. if S is not empty, insert ⟨MBB(S), v⟩ into N;
20. while ( N is not root ) do
21.     if N overflows then
22.         Split(N);
23.     else if N underflows then
24.         Remove N and reinsert the records from N into the tree at N's level;
25.     endif
26.     Adjust Parent(N) and set N =Parent(N); // Parent(N) returns the parent page of N
27. endwhile
28. if N overflows then
29.     Split old root and create a new root;
30. else if N has only one record and N is not leaf then
31.     Remove N and set N's child as the new root;
32. endif
```

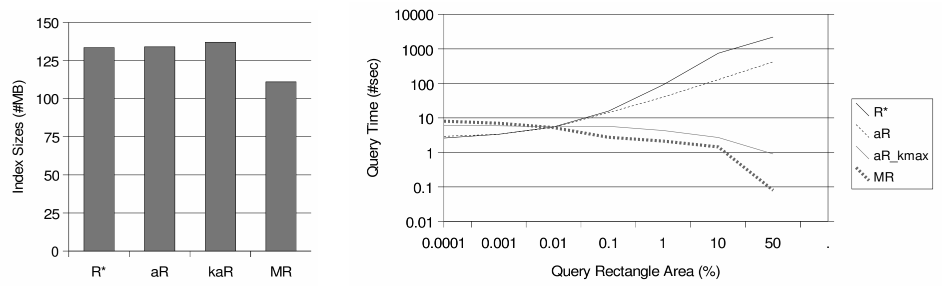
Generally, the insertion algorithm follows a single path from the root to a leaf. Reorganizations may follow the path back up to the root. The optimizations are applied when going down the tree. Steps 5 and 6 correspond to the k-max optimization which removes a sub-tree if the newly inserted object has a larger value and spatially contains the sub-tree. Steps 8 to 11 correspond to the area-reduction optimization which tries to reduce the size of the box to be inserted. Step 14 deals with the rare case when all sub-trees in some index page N become obsolete due to the insertion of an object. This may occur only when N is a root page, since otherwise the index record pointing to N in the parent page would be obsolete before N has a chance to be examined. For this case, the algorithm results in a tree with a single page and a single record. Step 15 means that the object to be inserted is obsolete and no recursive insertion into lower levels are needed. Step 16 chooses a child page to recursively insert into. We use the same algorithm as in the R*-tree. In steps 20 to 27, the path of pages is examined backwards. The way to split an overflowed page or to reinsert entries in an underflowed page is identical to the approaches in the R*-tree, plus the maintenance of the additional information kept along with each index record. Steps 28 through 32 handles overflow/underflow of the root page. As a consequence, the tree height may increase/decrease.

5 Performance

We compare the performance of the proposed MR-tree against the plain R*-tree, the aR-tree and the aR-tree/k-max. 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-tree/k-max 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 page.

We present results with two datasets, each containing 5 million square objects randomly selected in a two-dimensional space. The space in both dimensions is [1, 1 million]. The first data set was used to test the performance in the presence of small objects. The size of each object was randomly choosen from 10 to 1000. The second data set contains medium sized objects. The size of each object was randomly choosen from 10 to 10,000.



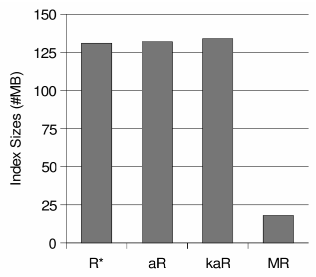
(a) Index Sizes. (b) Query performance.

Figure 3: Comparing the performance for small object dataset.

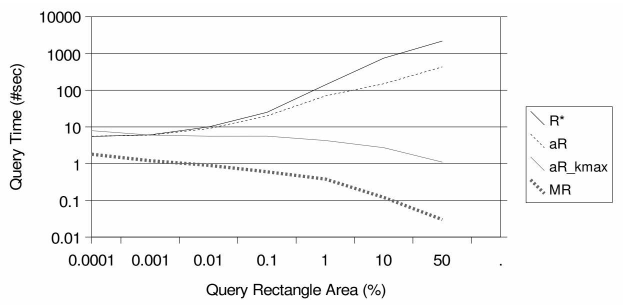
Figure 3 compares the performance for the small object dataset. The MR-tree uses about 25other methods. This is because some obsolete records were removed from the index. The aR-tree/k-max occupies the most space, since compared with the R*-tree and the aR-tree it stores more information in each index record.

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. Figure 3b shows the average time per query for various query sizes. While all methods are comparable for small query rectangles (since few objects satisfy the query anyways), the MR-tree is clearly the best as the query rectangle size increases. Note that the query time scale is logarithmic, so the actual difference in query speeds is drastic (for example, with query size of 1MR-tree is about 20 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-tree/k-max with the plain aR-tree. The MR-tree performs better than the aR-tree/k-max 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.

For medium-size objects, the performance improvement of the MR-tree over the other structures is even better (figure 4). The reason is that larger objects have more chances to contain other objects and thus make them obsolete; as a result, the MR-tree becomes more compact. This trend continued with datasets with larger objects (results not shown for brevity).



(a) Index Sizes.



(b) Query performance.

Figure 4: Comparing the performance over medium objects.

We also compared the index generation time. For small objects, the MR-tree needs more CPU time to generate (about 2.5 times more), but a little less I/O time. This is to be expected. Although the MR-tree occupies less space, the generation of it needs more CPU time to maintain the extra information stored in the index records. For medium objects, the MR-tree takes about the same CPU time but less I/O time as compared with the other structures, since it is much smaller. Due to space limitations the graphs are omitted but can be found in [ZT01].

6 Making the MSB-tree Dynamic

The MSB-tree as proposed in [YW01] is not fully dynamic in the sense that it needs to be periodically reconstructed. To reconstruct the MSB-tree, the whole tree is browsed in a depth-first manner to report every interval together with the aggregation value during this interval. The intervals thus reported are continuous to one another. Adjacent intervals with the same value are merged. All the intervals are then inserted into a second, initially empty MSB-tree which eventually replaces the original tree. Note that during the reconstruction phase, the MSB-tree is *off-line*, i.e. queries cannot be answered.

We found that the box-elimination optimization we proposed in section 3 can be used to avoid the reconstruction of the MSB-tree. The idea is that when a new interval is inserted, the sub-trees which are fully contained in the interval and which have smaller values are obsolete and should be removed. It can be proved that the update algorithm including the box-elimination optimization always maintain a *compact* tree, i.e. reconstruction will not reduce the number leaf-level records. It can also be proved that the cost of the update algorithm after it is optimized remains logarithmic. For more details, we refer to [ZT01].

7 Conclusion

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 maintainance of MIN/MAX aggregations. 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 make an existing aggregation index (the MSB-tree dynamic). An interesting open problem is whether efficient aggregation indices exist for higher dimensional data with non-zero extents.

Acknowledgements

We would like to thank Dimitrios Gunopulos for many helpful discussions. We also thank Dimitris Papadias for providing us valuable input on related work including their own. Finally, we are grateful to Bernhard Seeger for the R*-tree code.

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 Compututational Geometry*, (B. Chazelle, E. Goodman and R. Pollack eds.), American Mathematical Society, Providence, 1998.

[AS90] (11. spatial query) W. G. Aref and H. Samet, "Efficient Processing of Window Queries in the Pyramid Data Structure", *Proc. of PODS*, pp. 265-272, 1990.

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

[BKS+90] N. Bechmann, 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.

[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. Jurgens 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.

[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", *submitted for publication*, 2001.

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

[RKR97] N. Roussopoulos, Y. Kotidis and M. Roussopoulos, "Cubetree: 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.

[YW01] J. Yang and J. Widom, "Incremental Computation and Maintenance of Temporal Aggregates", to appear in *Proc. of ICDE*, 2001. (also available at http://www-db.Stanford.EDU/~junyang/research/pubs.html)

[ZMT+01] D. Zhang, A. Markowetz, V. Tsotras, D. Gunopulos and B. Seeger, "Efficient Computation of Temporal Aggregates with Range Predicates", to appear in *Proc. of PODS*, 2001. (also listed as *TimeCenter Tech Report 52*)

[ZT01] D. Zhang and V. J. Tsotras, "Improving Min/Max Aggregation over Spatial Objects", *Tech Report UCR_CS_01_03, CS Dept., UC Riverside*, http://www.cs.ucr.edu/research/tech_reports, 2001.

[ZTM+01] D. Zhang, V. Tsotras, A. Markowetz, M. Hadjieleftheriou, D. Gunopulos and B. Seeger, "Efficient Aggregation over Multidimensional Objects", *Tech Report UCR_CS_01_01, CS Dept., UC Riverside*, http://www.cs.ucr.edu/research/tech_reports, 2001, submitted for publication.