Evaluation of Top-k OLAP Queries Using Aggregate R–Trees*

Nikos Mamoulis[1], Spiridon Bakiras[2], and Panos Kalnis[3]

[1] Department of Computer Science, University of Hong Kong, Pokfulam Road, Hong Kong
nikos@cs.hku.hk
[2] Department of Computer Science, Hong Kong University of Science and Technology, Clear Water Bay, Hong Kong
sbakiras@cs.ust.hk
[3] Department of Computer Science, National University of Singapore
kalnis@comp.nus.edu.sg

Abstract. A top-k OLAP query groups measures with respect to some abstraction level of interesting dimensions and selects the k groups with the highest aggregate value. An example of such a query is "find the 10 combinations of product-type and month with the largest sum of sales". Such queries may also be applied in a spatial database context, where objects are augmented with some measures that must be aggregated according to a spatial division. For instance, consider a map of objects (e.g., restaurants), where each object carries some non-spatial measure (e.g., the number of customers served during the last month). Given a partitioning of the space into regions (e.g., by a regular grid), the goal is to find the regions with the highest number of served customers. A straightforward method to evaluate a top-k OLAP query is to compute the aggregate value for each group and then select the groups with the highest aggregates. In this paper, we study the integration of the top-k operator with the aggregate query processing module. For this, we make use of spatial indexes, augmented with aggregate information, like the aggregate R–tree. We device a branch-and-bound algorithm that accesses a minimal number of tree nodes in order to compute the top-k groups. The efficiency of our approach is demonstrated by experimentation.

1 Introduction

Data warehouses integrate and summarize large amounts of historical information, accumulated from operational databases. On-line Analytical Processing (OLAP) refers to the set of operations that are applied on a Data Warehouse to assist analysis and decision support. Data warehouses are usually modeled by the star schema [8], where some measures (e.g., sales) are analyzed with respect to some interesting dimensions (e.g., products, stores, time, etc.), representing business perspectives. A *fact* table stores records corresponding to transactions that have been consolidated in the warehouse. One or more columns in the fact table capture the measures, while each remaining attribute stores values for a dimension at the *most refined* abstraction level. For example, a tuple

* Supported by grant HKU 7380/02E from Hong Kong RGC.

C. Bauzer Medeiros et al. (Eds.): SSTD 2005, LNCS 3633, pp. 236–253, 2005.
© Springer-Verlag Berlin Heidelberg 2005

in the fact table stores a transaction for a particular product-id sold at a particular store-id at some particular time instant. A dimensional table models multi-level hierarchies of a particular dimension. For example, a tuple in the dimensional table `product` stores information about the color, type, manufacturer, etc., for each product-id. Data analysts are interested in summarizing the fact table information with respect to the interesting dimensions at some particular level of their hierarchies, e.g., "retrieve the total sales per month, product color, and store location".

The star schema was extended in [6] to include spatial abstraction levels and dimensions. The location of stores where products are sold is an example of a spatial attribute, with respect to which the sales could be analyzed (possibly together with non-spatial attributes of other dimensions). We can also define hierarchies for spatial attributes. In general, hierarchies of spatial and non-spatial ordinal attributes can be defined either by predefined decompositions of the value ranges (e.g., exact location, city, county, state, country, etc.) or by ad-hoc partitioning techniques (e.g., by a regular spatial grid of arbitrary granularity).

An ideal method to manage a data warehouse, in order to answer OLAP queries efficiently, is to materialize all possible groupings of the measures with respect to every combination of dimensions and hierarchies thereof. In this way, the result of each OLAP query could directly be accessed. Unfortunately, this technique is infeasible, because huge space is required for storing the results for all possible combinations and long time is required to maintain these combinations after updates in the warehouse. In view of this, several partial materialization techniques [7,6] select from the complete hierarchy of possible hyper-cubes those that assist the evaluation of most frequent OLAP queries and at the same time they meet the space and maintenance time constraints. Nevertheless these techniques still cannot deal with ad-hoc groupings of the dimensional ranges, which may still have to be evaluated directly on base tables of the data warehouse. This is particularly the case for spatial attributes, for which the grouping hierarchies are mostly ad-hoc.

Papadias et al. [14] proposed a methodology that remedies the problem of ad-hoc groupings in spatial data warehouses. Their method is based on the construction of an *aggregate* R–tree [10] (simply aR–tree) for the finest granularity of the OLAP dimensions (i.e., for the fact table data). The aR–tree has similar structure and construction/update algorithms as the R*–tree [3]; the difference is that each directory node entry e is augmented with aggregate results on all values indexed in the sub-tree pointed by e. Accordingly, the leaf node entries contain information about measures for some particular combination of dimensional values (i.e., spatial co-ordinates or ordinal values of other dimensions at the finest granularity). This index can be used to efficiently compute the aggregate values of ad-hoc selection ranges on the indexed attributes (e.g., "find the total sales for product-ids 100 to 130 between 10 Jan 2005 and 15 Feb 2005"). In addition, it can be used to answer OLAP group-by queries for ad-hoc groupings of dimensions by spatially *joining* the regions defined by the cube cells with the tree.

An interesting OLAP query generalization is the *iceberg* query [5]; the user is only interested in cells of the cuboid with aggregate values larger than a threshold t (e.g., "find the sum of sales for each combination of product-type and month, only for combinations where the sum of sales is greater than 1000"). In this paper, we study a variant

of iceberg queries, which, to our knowledge, has not been addressed in the past. A top-k OLAP query groups measures in a cuboid and returns only the k cells of the cuboid with the largest aggregate value (e.g., "find the 10 combinations of product-type and month with the largest sum of sales"). A naive way to process top-k OLAP queries (and iceberg queries) is to perform the aggregation for each cell and then select the cells with the highest values. Previous work [5] on iceberg queries for ad-hoc groupings employed hashing, in order to early eliminate groups having small aggregates and minimize the number of passes over the base data.

We follow a different approach for top-k OLAP query evaluation, which operates on an aR-tree that indexes the fact table. We traverse the tree in a branch-and-bound manner, following entries that have the highest probability to contribute to cells of large aggregate results. By detecting these dense cells early, we are able to minimize the number of visited tree nodes until the termination of the algorithm. Our method can also be applied for iceberg queries, after replacing the floating bound of the k-th cell by the fixed bound t, expressed in the iceberg query. As we show, our method can evaluate ad-hoc top-k OLAP queries and iceberg queries by *only a part of the base data, only once*. Therefore, it is more efficient than hash-based methods [5] or spatial joins [14], which require *multiple* passes over the *whole* fact table. The efficiency of our approach is demonstrated by extensive experimentation with real datasets.

The remainder of the paper is organized as follows. Section 2 reviews related work. Section 3 formally defines top-k OLAP queries. In Section 4, we describe in detail our proposed solution. Section 5 experimentally demonstrates the efficiency of the proposed algorithm. Finally, Section 6 concludes the paper.

2 Related Work

To date, there is a huge bibliography on data warehousing and OLAP [13], regarding view selection and maintenance [7,12], modeling [8,2], evaluation of OLAP queries [1], indexing [9], etc. In this section, we discuss in more detail past research on indexing spatial data for evaluating aggregate range queries and OLAP queries in the presence of spatial dimensions. In addition, we review past work on iceberg queries and top-k selection queries and discuss their relation to the problem studied in this paper.

2.1 Spatial OLAP

Methods for view selection have been extended for spatial data warehouses [6,15], where the spatial dimension plays an important role, due to the ad-hoc nature of groups there. Papadias et al. [14] proposed a methodology, where a spatial hierarchy is defined by the help of an aggregate R–tree (aR–tree). The aR–tree is structurally similar to the R*–tree [3], however, it is not used to index object-ids, but *measures* at particular locations (which could be mixtures of spatial co-ordinates and ordinal values of non-spatial dimensions at the finest granularity). The main difference to the R*–tree is that each directory node entry e is augmented with aggregate results for all measures indexed in the sub-tree pointed by e. Figure 1 shows an exemplary aR–tree (the ids of entries at the leaf level and the contents of some nodes are omitted). The value shown under each

non-leaf entry e_i corresponds to an aggregate value (e.g., sum) for all measures in the subtree pointed by e_i.

The tree can be used to efficiently compute *aggregate range queries*, which summarize the measures contained in a spatial region. These queries are processed similarly to normal range queries on a R–tree. Tree entries (and their corresponding subtrees) are pruned if they do not intersect the query region. If the MBR of an entry e partially overlaps the query, it is followed as usual, however, if e's MBR is totally covered by the query range, the augmented aggregate result $e.agg$ on e is counted and the subtree pointed by e needs not be accessed. For example, consider an aggregate sum query q indicated by the dashed rectangle of Figure 1. From the three root entries, q overlaps only e_2, so the pointer is followed to load the corresponding node and examine entries e_7, e_8, e_9. From these, e_7 is pruned and e_8 partially overlaps q, so it is followed and 10, 5 are added to the partial result. On the other hand, e_9 is totally covered by q, so we can add $e_9.agg = 20$ to the query result, without having to visit the leaf node pointed by e_9. The aR–tree can also be used for *approximate* query processing, if partially overlapped entries are not followed, but their aggregate results are scaled based on the overlapped fraction of e.MBR [10].

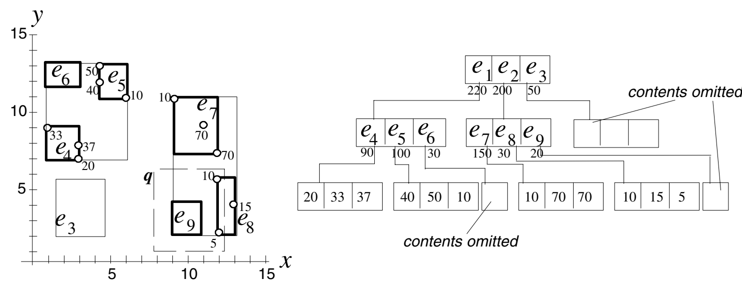


Fig. 1. An aR–tree

[14] showed how the aR–tree can be used to process OLAP group-by queries for groups defined by ad-hoc spatial regions. In this case, a *spatial join* is performed between the tree and the boundaries of the regions for which we want to compute aggregate results. Finally, if there is not enough space to fully materialize the tree, it can be *partially* materialized, by selecting levels that correspond to most significant grouped hierarchies (i.e., the ones that assist most queries).

Another query, related to the top-k OLAP query we study in this paper, is the top-k spatial join [16]. Given two spatial datasets A and B, the top-k spatial join retrieves the k objects from A or B that intersect the largest number of objects from the other dataset. A branch-and-bound algorithm that computes this join is proposed in [16], assuming that both A and B are indexed by R–trees. Our top-k OLAP queries could be considered as a variant of this join query, where one of the joined datasets is the set of regions from which we want to derive the ones with the top-k aggregate result. However, these regions in our problem are ad-hoc and we do not presume an index on them. In addition, the top-k join considers the *count* aggregate function only. Finally,

the algorithm proposed in [16] is a join method that essentially accesses both datasets at least once for most typical queries. On the other hand, we do not explicitly access a dataset corresponding to the interesting regions (we compute their results on-the-fly instead) and we access only a part of the base data (using an aR–tree index on them).

2.2 Iceberg Queries

The term *iceberg* query was defined in [5] to characterize a class of OLAP queries that retrieve aggregate values above some specified threshold t (defined by a `HAVING` clause). An example of an iceberg query in SQL is shown below:

```
SELECT product-type, store-city, sum(quantity)
FROM Sales
GROUP BY product-type, store-city
HAVING sum(quantity) >= 1000 ;
```

In this example, from all groups of product-types and store locations (cities), the user wants only those having aggregate result no smaller than $t = 1000$. The motivation is that the data analyst is often interested in exceptional aggregate values that may be helpful for decision support. A typical query optimizer would first perform the aggregation for each $\langle$`product-type,store-city`$\rangle$ group and then return the ones whose aggregate value exceeds the threshold. In order to avoid useless aggregations for the pairs which disqualify the query, [5] present several hash-based methods with output-sensitive cost. These techniques were later extended for selecting exceptional groups in a whole hierarchy of data cubes [4]. The iceberg query is similar to the top-k OLAP query we study in this paper. In our case, we are interested in the k groups with the largest aggregate values, instead of aggregates above a given threshold. As opposed to the methods in [5], our top-k OLAP algorithm is not based on hashing, but operates on an existing aR–tree that indexes the base data. As we show, our method can also be adapted for iceberg queries.

2.3 Top-*k* Aggregate Queries

[11] propose methods for processing top-k *range* queries in OLAP cubes. Given an arbitrary query range, the problem is to find the top-k measures in this range. This query is a generalization of *max* range queries (i.e., $k = 1$ for *max* queries). The data cube is partitioned into sufficiently small cells and the top-k aggregate values in each partition are pre-computed. These values can then be used to compute the top-k results in query regions that cover multiple cells. Top-k range queries are essentially different than top-k OLAP queries, since the latter deal with the retrieval of top-k aggregated values of *groups* (as opposed to top-k measures) in the whole space (as opposed to a particular range). To our knowledge, there is no prior work on the problem we study in this paper. In the next section, we formally define top-k OLAP queries and motivate their processing using aR–trees.

3 Problem Formulation

We assume a data warehouse with a star-schema, where the dimensional values recorded in the fact table correspond to either spatial locations (for spatial dimensions) or to ordinal (i.e., numerical) values at finest granularity. We also assume that the user is interested in computing aggregates based on a partitioning of the dimensional domains (e.g., "retrieve the total sales per month, product type, and store location (city)"). This partitioning could be ad-hoc or according to some known hierarchy (e.g., time instants are grouped to hours, days, weeks, months, etc.). We assume that each partition forms a *contiguous* range of values in the domain, and that partitions are disjoint and cover the complete domain of the dimension. Finally, we consider a single aggregate function (sum) on a single measure (e.g., sales quantity). We will later discuss how to extend our methodology for cases where these assumptions do not hold.

Let D be the total number of dimensions. As discussed in [7,14], we typically select a subset of the 2^D dimensional combinations to materialize/index. Consider such a combination of dimensions. We can build an aR–tree index on top of the corresponding cuboid, where we index the summarized information based on the finest granularity values recorded in the fact table. This index can be used to answer OLAP queries (both group-by's and range selections) related to this set of dimensions (or a subset thereof) for any combination of hierarchies (i.e., partitionings) in the individual dimensions. For example, an aR–tree on dimensions $\langle$`time, product, store-location`$\rangle$ could be used to compute the aggregate value for every combination of date/week/month, product-id/type, and street/city/county/state location of stores.

Selecting the combinations of dimensions to materialize can be done with existing techniques (e.g., [7]) and it is out of the scope of this paper. While results for all or some combinations of predefined dimensional partitionings could be pre-computed and materialized, we assume that only the finer granularity summaries for the selected dimensional sets are materialized. The rationale is that (i) it is expensive to store and maintain pre-computed results for all possible combinations of dimensional partitionings, (ii) there could be ad-hoc partitionings, especially in the space dimension (as discussed in [6,15,14]), and (iii) the aR–tree can handle well arbitrary partitionings of the multi-dimensional space [14].

Now, we are ready to formally define the top-k OLAP query:

Definition 1. Let $\mathcal{D} = \{d_1, \ldots, d_m\}$ be a set of m interesting dimensions and assume that the domain of each dimension $d_i \in \mathcal{D}$ is partitioned into a set $R_i = \{r_i^1, \ldots, r_i^{|R_i|}\}$ of $|R_i|$ ad-hoc or predefined ranges based on some hierarchy level. Let k be a positive integer. An OLAP top-k query on $\mathcal{D}$ selects the k groups $g_1, \ldots, g_k$ with the largest aggregate results, such that $g_j = \{r_1, \ldots, r_m\}$ and $r_i \in R_i \forall i \in [1, m]$.

An example top-10 OLAP query could be expressed by the SQL statement that follows. Here, the partition ranges at each dimension are implicitly defined by levels of hierarchy (type for products and city for stores).

```
SELECT product-type, store-city, sum(quantity)
FROM Sales
GROUP BY product-type, store-city
```

```
ORDER BY sum(quantity)
STOP AFTER 10;
```

A naive method to process a top-k OLAP query is to compute the aggregate result for each *cell* (i.e., group of ranges from different dimensions) and while doing so maintain a set of the top-k cells. This method has several shortcomings. First, many cells with small aggregate values will be computed and then filtered out, wasting computations and I/O accesses. Second, since the definition of the dimensional ranges may be ad-hoc, measures within a given cell may be physically located far in the disk. As a result, it may not be possible to compute the aggregates for all cells at a single pass. In order to alleviate the problem, hashing or chunking techniques can be used. Alternatively, a spatial join can be performed if the base data are indexed by an aR–tree, as discussed. Nevertheless it is still desirable to process the top-k OLAP query without having to access all data and without having excessive memory requirements.

Assume that we have only two (spatial) dimensions x and y, with integer values ranging from 0 to 15. In addition, assume that each dimension is partitioned into value ranges $[0, 5)$, $[5, 10)$, $[10, 15]$. Figure 2 shows a set of measures indexed by an aR–tree (the same as in Figure 1) and the 3×3 groups (cells) $c_1, \dots, c_9$ defined by the combinations of partition ranges. Based on the information we see in the figure (some contents are omitted), we know that $c_1.agg = 120$, since e_6 with $e_6.agg = 30$ is totally contained in c_1. In addition, we know that $c_4.agg \geq 90$ and $c_4.agg \leq 90 + e_3.agg = 140$. Similarly, $c_9.agg \leq 20 + e_9.agg = 40$. Observe that result of a top-1 OLAP query (i.e., the cell with the highest aggregate result, assuming *sum* is the aggregate function) is c_6, with $c_6.agg = 150$, because there is no other cell that can reach this result in any case. Thus, by having browsed the tree partially *we can derive some upper and lower bounds for the aggregate results at each cell*, which can help determining early the top-k OLAP query result. This observation is used by our branch-and-bound algorithm described in the next section.

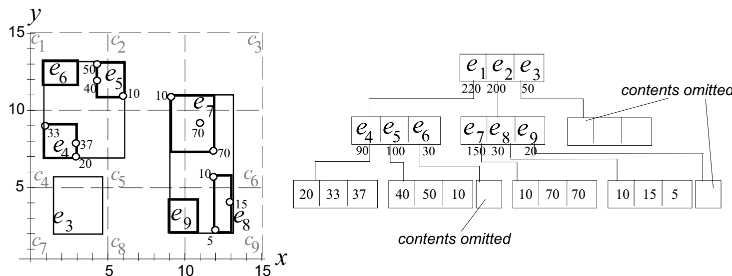


Fig. 2. Top-k grouping example

4 Processing Top-k OLAP Queries Using an aR–Tree

Given an aR–tree that indexes a set of dimensions at a finest granularity and a top-k OLAP query that is based on an ad-hoc partitioning of each dimensional domain, our

objective is to evaluate the query by visiting the smallest possible number of nodes in the aR–tree. Assume, for the moment, that we can maintain in memory information about all cells (i.e., the total number of cells is manageable). While traversing the aR–tree, we can compute (partial or total) results for various cells. For example, by visiting the leftmost path of the tree in Figure 2, we know that the result $c_4.agg$ for cell c_4 is between 90 (due to the contents of e_4) and 140 (due to e_3 that overlaps c_4). Thus, we can set *lower* $c_4.lb$ and upper $c_4.ub$ bounds for the aggregate result in c_4, and accordingly for all cells in space. In addition, based on the information derived by traversing the tree, we can maintain a set LB of k cells with the largest lower bounds. The k-th largest lower bound can be used to prune aR–tree entries (and the corresponding sub-trees) as follows:

Lemma 1. Let t be the k-th largest lower bound. Let e_i be an aR–tree entry. If for all cells c that intersect e_i, $c.ub \leq t$, then the subtree pointed by e_i cannot contribute to any top-k result, and thus it can be pruned from search.

The proof of the lemma is straightforward based on the definitions of lower and upper bounds. Intuitively, this lemma can be used to terminate the algorithm after we have computed exactly the contents of some cells and non-visited subtrees overlap only with cells that cannot end up in the top-k result. Now the question is how we can compute and update the lower and upper bounds while traversing the tree. Another question is *how* should we traverse the tree (i.e., in what order should the nodes be visited) if we want to maximize the pruning power of Lemma 1. It turns out that both questions are related; their answers are given by the algorithm described in the next subsection.

4.1 The Basic Algorithm

An entry e_i of a subtree (not visited yet) that intersects a number of cells can contribute at most $e_i.agg$ to the aggregate result of the cell. For example, in Figure 2, even though we do not know the contents of the subtree pointed by e_3, we know that c_4 can contribute at most 50 to this cell. In addition, for an entry e_i which is totally contained in a cell c, we know that it contributes exactly $e_i.agg$ to c, without having to access the subtree pointed by e_i. For example, visiting the leaf node pointed by e_4 is pointless, since the MBR of the entry is totally contained in c_4, thus we know that c_4 gets exactly 90 from this entry. These observations, lead to the design of our top-k OLAP algorithm, which is described in Figure 3.

During the *initialization* phase of the algorithm, we visit the root node of the aR–tree and compute upper and lower bounds for all cells based on their overlap with root entries (lines 1–9). In our running example, we use $e_1.agg, e_2.agg, e_3.agg$ to compute $c_1.ub = 220, c_2.ub = 420, c_3.ub = 200$, etc. In addition, $c_i.lb = 0$ for all cells c_i, since no entry is totally contained in one of them. Assuming that $k = 1$ (in this example) and based on the information so far, the algorithm cannot terminate, since the highest lower bound is smaller than some upper bound.

At this moment, we have to determine which node to visit next. Intuitively, an entry which intersects the cell with the greatest upper bound should be followed first, in order to decrease this upper bound and at the same time increase the lower bounds of other

cells, potentially leading to an early termination of the algorithm. In addition, from all entries intersecting the cell with the greatest upper bound the one with the largest $e.agg$ should be visited first, since it is likely to contribute most in the cells it overlaps. Thus, we *prioritize* the entries to be visited according to the above criteria, and follow a *best-first* search order. In other words, all entries (i.e., subtrees) of the aR-tree that have not been followed yet are organized in a heap H (i.e., priority queue). The entry e to be followed next is the one with the greatest $e.agg$ from those intersecting the cell with the greatest upper bound. In our example, after visiting the root, e_1, e_2, e_3 are inserted into H and e_1 becomes the top element, since it intersects c_2 (and c_5) having $c_2.ub = 420$ and $e_1.agg > e_2.agg$ (e_2 also intersects c_2 and c_5). Lines 10–12 of the algorithm compute the heap order key for the root entries and insert them to H.

When de-heaping an entry e from H, we visit the corresponding node n at the aR–tree. Let $\mathcal{C}$ be the set of cells intersected by e. The first thing to do is to decrease the upper bounds of cells in $\mathcal{C}$ by $e.agg$, since these bounds will be refined by the entries of the new node n. For each entry $e_i \in n$ again we consider two cases; (i) e_i is totally contained in a cell, or (ii) e_i overlaps more than one cells. In the first case, we only update the lower bound of the covering cell. Otherwise, we add $e_i.agg$ to the upper bounds of all cells that intersect e_i. Note that for entries at the leaf level only case (i) applies. After processing all entries, the upper bounds of all cells in $\mathcal{C}$ are updated. Based on these new bounds, we compute the heap key of the newly processed entries (only for case (ii) entries) and add them on H. In addition, for entries that are already in H and intersect any cell in $\mathcal{C}$, we change the positions in H, if necessary, considering the new upper bounds of these cells.

The algorithm terminates (line 15) if for the entry e that is de-heaped $e.ub \leq t$, where t is the smallest of the top-k results found so far (stored in LB). Indeed if this condition holds, no cell can potentially have higher aggregate value than the currently k-th result.

Consider again the example of Figure 2 and assume that we want to find the cell with the highest aggregate value (i.e., $k = 1$). We start by examining the three root entries. We add them on H, after having computed $e_1.ub = 420$, $e_2.ub = 420$, and $e_3.ub = 270$. e_1 becomes the top heap element, since $e_1.agg > e_2.agg$.

After de-heaping e_1, we load the aR–tree node pointed by it. First, we reduce the upper bounds of c_1, c_2, c_4, c_5 by $e_1.agg = 220$. Entry e_4 is totally covered by cell c_4, thus we now have $c_4.lb = 90$. Note that we will never have to visit the node pointed by e_4. c_4 now becomes the currently best result and $t = 90$. Entry e_5 overlaps cells c_1 and c_2, increasing their upper bounds by $e_5.agg$. Finally, e_6 is fully contained in c_1 and sets $c_1.lb = 50$. The upper bounds of e_2 and e_3 are updated to 300 (due to c_2) and 140 (due to c_4), respectively. In addition, e_5 has been added to H with $e_5.ub = 300$ (due to c_2). The next entry to be de-heaped is e_2. Since $e_2.ub > t$, the algorithm does not terminate and we load the corresponding node and examine its entries which are all added on H. The top heap entry is now e_7 with $e_7.ub = 250$ (due to c_2). Still $e_7.ub > t$ and we pop the pointed node by it, which is a leaf node. The currently best cell now becomes c_6 with $c_6.lb = 140$. In turn, the top heap entry is e_8, with $e_8.ub = 170$ (due to c_6). After visiting the leaf node pointed by e_8, $c_6.lb$ becomes 150, which is the current t. The algorithm now terminates because the next entry popped from H is e_3 with $e_3.ub = 140 < t$.

```
Algorithm TopkOLAP(aR–tree T, k)
1.  LB := ∅; t := 0; c.lb := c.ub := 0, for all cells;
2.  n := root(R);
3.  for each entry e_i ∈ n do
4.      if e_i.MBR is contained in a cell c then
5.          c.lb := c.lb + e_i.agg; c.ub := c.ub + e_i.agg;
6.          add/update c in LB; /*heap of lower bounds*/
7.          t := k-th largest value in LB;
8.      else /*not contained*/
9.          for each cell c intersected by e_i set c.ub := c.ub + e_i.agg;
10. for each entry e_i ∈ n do
11.     e_i.ub := max{c.ub, ∀ cells c intersected by e_i};
12.         add e_i on a max-heap H; /*organize H primarily by e_i.ub; break ties, using e_i.agg*/
13. while notempty(H) do
14.     e := H.top;
15.     if e.ub ≤ t then break; /*termination condition*/
16.     n := load aR–tree node pointed by e;
17.     C := all cells c intersected by e;
18.     for each cell c ∈ C set c.ub := c.ub − e_i.agg;
19.     for each entry e_i ∈ n do
20.         if e_i.MBR is contained in a cell c then /* always true if n is a leaf node */
21.             c.lb := c.lb + e_i.agg; c.ub := c.ub + e_i.agg;
22.             add/update c in LB;
23.             t := k-th largest value in LB;
24.         else /*not contained*/
25.             for each cell c intersected by e_i set c.ub := c.ub + e_i.agg;
26.     for each entry e_i ∈ n not contained in a cell do
27.         e_i.ub := max{c.ub, ∀ cells c intersected by e_i};
28.         add e_i on H;
29.     for each entry e_j ∈ H overlapping some cell in C do
30.         e_j.ub := max{c.ub, ∀ cells c intersected by e_j};
31.         update e_j's position in H, if necessary;
```

Fig. 3. The basic algorithm for top-k OLAP queries

4.2 Minimizing the Memory Requirements

The pseudo-code of Figure 3 shows the basic functionality of our top-k OLAP algorithm. Due to the effective branch-and-bound nature of the algorithm, we can avoid accessing a large fraction of the aR–tree nodes, resulting in a sub-linear I/O performance. However, the basic version of algorithm has large space requirements, since for each cell we need to maintain and update a lower and upper aggregate bound. If the total number of cells is larger than the available memory (this can happen when we have very refined partitions at each dimension), then the algorithm is inapplicable. Therefore, it is crucial to minimize the number of cells for which we compute and maintain information. For this purpose, we use the following observations:

- *We need not keep information about cells that are intersected by at most one entry.* At the early phases of the algorithm, the MBRs of the high-level entries we have seen so far (e.g., the root entries) intersect a large number of cells. However, for a cell c that is intersected by only one entry e_i, we know that $c.ub = e_i.agg$, thus we do not have to have to explicitly compute and maintain this information. In addition, for cells intersected by no entry, we need not keep any information at all. Thus, we maintain information only for cells that are intersected by more than one entries. This holds only for cells with 0 lower bound; i.e., those for which no partial aggregate has been computed. On the other hand, we have to maintain any cell c with a partial aggregate (i.e., with $c.lb > 0$), if $c.ub > t$.
- *We can keep a single upper bound for all cells intersected by the same set of entries.* Consider two entries that are close in space and jointly affect a contiguous range of cells. We need not keep an upper bound for each cell, since we can use a single upper bound for the whole range. Later, if one of the two entries is de-heaped and the contents of its subtree are loaded, the range of cells is also broken and the (different) upper bounds for the individual cells are computed.
- *We need not keep information about cells that may not end up in the top-k result.* If, for a cell c, we know that $c.ub < t$, we can prune the cell and never consider it in computations of lower and upper bounds, for nodes that are visited next.

We implemented an advanced version of the top-k OLAP algorithm, which has small memory requirements, based on these observations. Initially, we do not keep explicit upper bounds for the cells, but compute $e_i.ub$ for the examined entries, according to the common cells they intersect. As soon as lower bounds (i.e., partial result) are computed for cells, we start maintaining cells with $c.lb > 0$, which may end-up in the top-k result. In addition, for every entry e_i in H, we keep pointers to all candidate cells (with $c.lb > 0$) that they overlap, but compute and maintain $e_i.ub$ on-the-fly, considering also cells with $c.lb = 0$, however, without explicitly maintaining those cells.

For example consider again the top-1 OLAP query on the tree of Figure 2 and assume that we are in the phase of examining the root entries (i.e., lines 3–12 of Figure 3). Instead of explicitly computing $c.ub$ for each cell overlapped by any entry, and then computing $e_1.ub, e_2.ub, e_3.ub$ from them, we follow an alternative approach that needs not materialize $c.ub$. For each entry (e.g., e_1), we compute on-the-fly $c.ub$ for all cells c it overlaps, by considering the influence of all other entries (e.g., e_2, e_3) in c. Then we set as $e_i.ub$ the largest $c.ub$. Later, after e_1 is de-heaped and the corresponding node is loaded, $c_1.lb$ and $c_4.lb$ are computed and stored explicitly, while these cells can end up in the top-k result.

4.3 Extensions for Related Problems and Generic Problem Settings

So far, we have described our basic algorithm and optimization techniques for it for the case of OLAP queries, where we look for the top-k cell in a cuboid with the greatest sum of a single measure. We now discuss variants of this query and how our algorithm can be adapted for them.

Iceberg queries. Our top-k OLAP algorithm can also be used to process iceberg queries (described in Section 2.2). We use exactly the same technique for searching the tree.

However, the threshold t used for termination is not floating, based on the current top-k result, but it is a fixed user parameter for the iceberg query. Entries for which $e_i.ub < t$ can be immediately pruned from search. In addition, cells for which the aggregate result has been computed and it is found no smaller than t, are immediately output. We do not need priority queues H and LB, but we can apply a simple depth-first traversal of the tree, to progressively refine the results for cells, until we know that the cell passes the threshold, or can never pass it, based on the potential aggregate value ranges. This method is expected to perform much better than the algorithm of [5] (which operates on raw data) because it utilizes the tree to avoid reading (i.e., prune) cells with aggregate values lower than t. Note that the algorithm of [5] requires reading all base data at least once.

Range-restricted top-k OLAP queries. Our algorithm can be straightforwardly adapted for top-k *range* OLAP queries, where the top-k cells are not searched in the whole space, but only in a sub-space defined by a window. For this case, we combine the window query with the top-k aggregation, by immediately pruning aR–tree entries and cells that do not intersect the window. Apart from that, the algorithm is exactly the same.

Arbitrary partitionings. In spatial OLAP, the regions of interest, for which data are aggregated and the top-k of them are selected, may not be orthocanonical (i.e., defined by some grid), but they could have arbitrary shape (e.g., districts in a city). Our algorithm can be adapted also for arbitrary regions as follows. When the aR–tree root is loaded, we spatially join the MBRs of the root entries to the extents of the regions. Thus, we define a bipartite graph, that connects regions to entries that overlap them. When an entry e is de-heaped, the graph is used to immediately find the regions it affects, in order (i) to compute upper bounds for the regions, based on e's children and (ii) to extend the graph by connecting the newly en-heaped entries (i.e., the children of e) that partially overlap some of these regions.

Query dimensionality. So far, we have assumed that there is an aR–tree for each combination of dimensions that could be in a top-k OLAP query. Nevertheless, as already discussed, it is usually impractical or infeasible to materialize and index even the most refined data level (i.e., the fact table) for *all* combinations of dimensions. Thus, a practicable approach is to index only certain dimensional combinations. Our algorithm can also be applied for top-k OLAP queries, where the set of dimensions is a *subset* of an indexed dimensional set. In this case, instead of cells, the space is divided into *hyper-stripes* defined by the partitionings of only those dimensions of the top-k OLAP query. For the remaining dimensions, the whole dimensional range is considered for each partition. For instance, consider an aR–tree on dimensions ⟨`time, product, store-location`⟩ and a top-k OLAP query on dimensions ⟨`product, store-location`⟩. We can use the aR–tree to process the query, however, disregarding the time dimension in the visited entries (and of course in the partitionings).

Non-contiguous ranges. We have assumed each partition of a particular dimension, defines a *contiguous range* on the base data. For example, in an OLAP query about

product-types, we assume that the product-ids are ordered or clustered based on product-type. However, this might not be the case, for all hierarchical groupings of the same dimension. For instance, we cannot expect the domain of product-ids to be ordered or clustered by both product-type and product-color. In order to solve this problem, we consider as a different dimension each ordering at the most refined level of an original dimension (according to the hierarchies of the dimension). In other words, we treat as different dimensions two orderings of product-ids; one based on product-type and one based on product-color. Given an arbitrary OLAP query, we use the set of dimensions, where each dimension is ordered (at the finest granularity level) such that the OLAP partitionings are contiguous in the domains of the individual dimensions.

Multiple measures and different aggregate functions. So far, we considered a single measure (e.g., sales quantity) and aggregate function (i.e., sum). As discussed in [10,14], the aR–tree could be augmented with information about multiple measures and more than one aggregate functions (i.e., sum, count, min, max). Our method is straightforwardly applied for arbitrary aggregations of the various measures, assuming that the aR–tree on which it operates supports the measure and aggregate function of the query.

5 Experimental Evaluation

We evaluated the efficiency of the proposed top-k OLAP algorithm, using synthetically generated and real spatial data. We compared our algorithm to the naive approach of scanning the data and computing on-the-fly the aggregate results for each cell, while maintaining at the same time the top-k cells with the largest aggregate results. Unless otherwise stated, we assume that we can allocate a counter (i.e., partial measure) for each cell in memory, a reasonable assumption for most queries. In the case where memory is not enough for these counters, the naive approach first hashes the data into disk-based buckets corresponding to groups of cells and then computes the top-k result at a second pass over these groups. We use I/O cost as a primary comparison factor, as the computational cost is negligible compared to the cost of accessing the data. The naive method and our top-k OLAP algorithm were implemented on a 2.4GHz Pentium 4 PC with 512 Mb of memory.

5.1 Description of Data

Our synthetic data are d-dimensional points generated uniformly in a $[1 : 10000]^d$ map. We use the following approach in order to generate the measure of each point. First, we randomly choose 10 anchor points in the data space. To generate the measure for a point, we fist find its nearest anchor point and its distance to it. All potential distances of points to their nearest anchor were discretized using 1000 bins. The measure assigned to a point follows a Zipfian distribution favoring small distances. The measure value corresponding to the largest distance is 1. The remaining measures were normalized based on this value. The generator simulates an OLAP application, where most of the transactions (i.e., points) have similar and small measures, whereas there are few, large transactions.

We also used a non-uniform dataset; a 2D spatial dataset containing 400K road segments of North America.[1] We assigned a measure at the center of each segment, using the same methodology as for the synthetic data described above. The resulting dataset models a collection of traffic measurements on various roads of a real map.

5.2 Experimental Results

In the first set of experiments, we compare the efficiency and memory requirements of our algorithm for top-k OLAP queries, compared to the naive approach on the synthetic data, for various data generation and query parameters. The default data generation parameter values are N=200K points (i.e., fact table tuples), d=2 dimensions, and θ=1 for the Zipfian distribution of measures. Unless otherwise stated, we set the page (and aR–tree node) size to 1Kb. For top-k queries, the default parameter values are $k = 16$ and $c = 10000$ total group-by cells (e.g., 100×100 cells for a 2D dataset).

The first experiment compares our algorithm to the naive approach for various sizes of the base data, using the default values for the other parameters. Figure 4 shows the results. Our top-k OLAP algorithm incurs an order of magnitude fewer I/O accesses compared to the naive approach, due to its ability to prune early aR–sub-trees that do not contain query results. The performance gap grows with N, because the aR–tree node extents become smaller and there are higher chances for node MBRs to be contained in cells and not accessed.

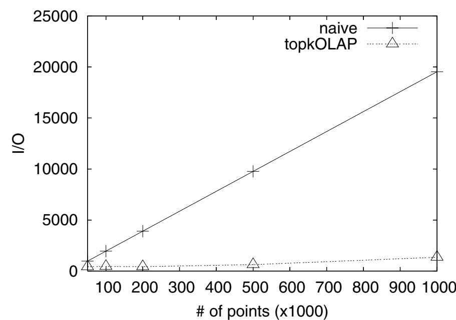


Fig. 4. Performance as a function of database size

Next, we evaluate the efficiency of our approach as a function of the skew on the measures (Figure 5). We used the default data generation parameters and varied the values of θ to $0, 0.5, 1, 1.5, 2$. As expected, the efficiency of our method increases with θ, because the top-k cells become more distinguishable from the majority of cells with low aggregate values. On the other hand, for uniform measures (recall that the points are also uniform), our algorithm becomes worse than the naive approach; it accesses all aR–tree pages (more than the data blocks due to the lower node utilization). In this case, the top-k results are indistinguishable from the remaining cells, since all cells have more or less the same aggregate value.

[1] collected and integrated from http://www.maproom.psu.edu/dcw/

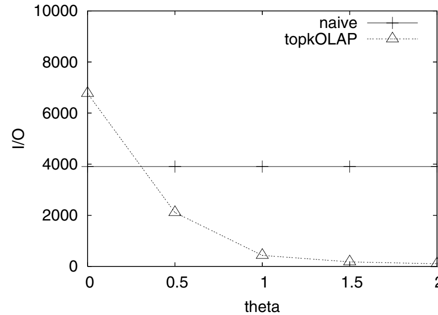


Fig. 5. Performance as a function of skew on the measure values

We also validated efficiency as a function of k; the number of cells with the highest values to be retrieved. As Figure 6 shows, the cost of our method increases with k, although not dramatically. The reason is that for large values of k, the k-th result becomes less indistinguishable from the average cells (or else, the k-th result becomes less different than the k+1-th).

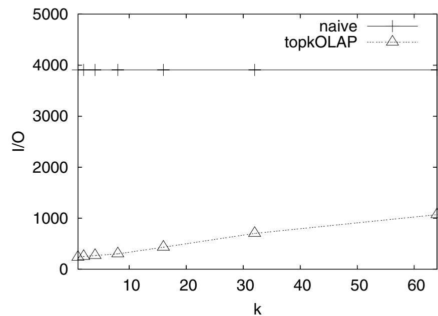
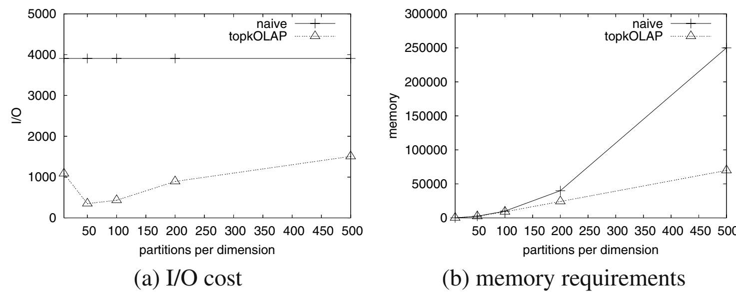


Fig. 6. Performance as a function of k

So far, we have assumed a fixed number of cells (100×100). We now evaluate the performance of our method when this value changes. Figure 7a shows the I/O cost of our method as a function of the number of partitions at each dimension. The dimensionality is fixed to $d = 2$, so the total number of cells is the square of the partitions per dimension. In general, the performance of our algorithm decreases with c, but not dramatically. For as few as $c = 10 \times 10$ cells, our method accesses many nodes because $k = 16$ is relatively high compared to the number of cells and there is no great difference between the k-th and k+1-th cell.

We also plot the memory requirements of the two algorithms in Figure 7b. The memory usage is measured in terms of cells for which lower bounds (or partial measures for the naive approach) are explicitly maintained in memory in the worst case. For

the naive algorithm this corresponds to the total number of cells, assuming that the memory is large enough to accommodate them. For our algorithm (see Section 4.2), the memory requirements are also dominated by the number of cells, which is expected to be much larger than the number of entries in the heap. The number of cells for which we have to keep information in memory is not constant; initially it increases, it reaches a *peak*, and then decreases. Here, we plot the peak number of cells. Observe that for small c our method has similar memory requirements to the naive approach. However, the memory requirements of our method increase almost linearly with the number of partitions per dimension, as opposed to the naive approach which requires $O(c)$ memory (i.e., quadratic to the number of partitions per dimension). This is a very important advantage of our approach, because memory savings are more important for large values of c, e.g., where c exceeds the available memory.



(a) I/O cost (b) memory requirements

Fig. 7. Performance varying the number of partitions

Figure 8 shows the performance of our algorithm, when varying the problem dimensionality d. The total number of cells is fixed to $c = 10000$, which implies that the partitions per dimension decrease with d. Note that for few dimensions (2 or 3) our method performs well, however, for higher dimensional values, it may access similar or more pages compared to the naive approach. This behavior can be attributed to two facts. First, as the dimensionality increases, the performance of multi-dimensional structures (like the aR–tree) deteriorates. The bounding hyper-rectangles become less tight with more empty space. In addition, their extents are larger and it becomes unlikely that they separate well the top-k cells from cells of small aggregate values. Second, as d increases, the extents of cells at each dimension become larger, thus more aR–tree nodes overlap the cells in the result, as well as other, irrelevant cells. Thus, our method is especially useful for low (2 or 3) dimensional data (like spatial OLAP data), or skewed high-dimensional data.

The next experiment evaluates the efficiency of our approach for top-k spatial OLAP queries. We used the spatial dataset and generated a measure for each point in it, according to their distance to the nearest of 10 random anchor points ($\theta = 1$). Figure 9a shows that our algorithm is very efficient compared to the naive approach for a wide range of spatial partitionings. In Figure 9b, we compare the two methods for 100×100 cells and different values of θ, when generating the measures. Our top-k OLAP algorithm is

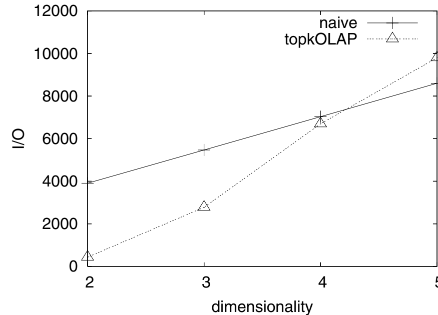
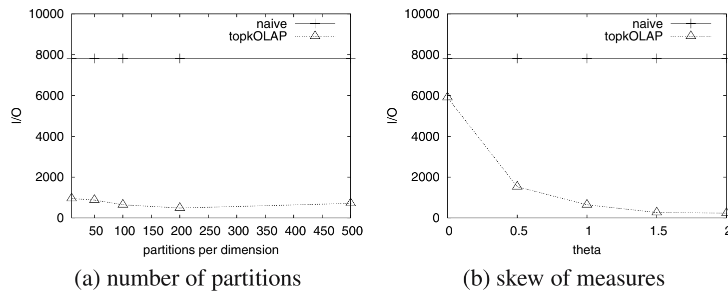


Fig. 8. Performance with respect to dimensionality

more efficient than the naive approach, even for uniform measures. Due to the spatial skew, many cells are empty and there is large difference in the aggregate values of cells, making our method very effective in pruning the search space. In addition, the data are very dense, compared to the uniform points in the synthetically generated datasets, and many nodes of the aR–tree need not be loaded as they are included in cells. Overall, our method is very efficient when either the data points or the measures are skewed, a realistic case especially in spatial data warehouses (since most real spatial data are skewed by nature).



(a) number of partitions (b) skew of measures

Fig. 9. Performance for spatial OLAP queries

6 Conclusions

In this paper, we studied a new and important query type for on-line analytical processing; the top-k OLAP query. We proposed a branch-and-bound technique that operates on an aR–tree and computes the result of the query, by accessing only a part of the tree. We proposed an effective optimization that greatly reduces the memory requirements of our method, rendering it applicable even to queries with a huge number of candidate results (i.e., cells of the partitioned space). Experiments confirm the efficiency of our approach, compared to a conventional hash-based approach that does not utilize existing indexes.

References

1. S. Agarwal, R. Agrawal, P. Deshpande, A. Gupta, J. F. Naughton, R. Ramakrishnan, and S. Sarawagi. On the computation of multidimensional aggregates. In *Proc. of VLDB*, pages 506–521, 1996.
2. R. Agrawal, A. Gupta, and S. Sarawagi. Modeling multidimensional databases. In *Proc. of ICDE*, pages 232–243, 1997.
3. N. Beckmann, H. Kriegel, R. Schneider, and B. Seeger. The R*–tree: An efficient and robust access method for points and rectangles. In *Proc. of ACM SIGMOD*, pages 220–231, 1990.
4. K. S. Beyer and R. Ramakrishnan. Bottom-up computation of sparse and iceberg CUBEs. In *Proc. of ACM SIGMOD*, 1999.
5. M. Fang, N. Shivakumar, H. Garcia-Molina, R. Motwani, and J. D. Ullman. Computing iceberg queries efficiently. In *Proc. of VLDB*, 1998.
6. J. Han, N. Stefanovic, and K. Koperski. Selective materialization: An efficient method for spatial data cube construction. pages 144–158, 1998.
7. V. Harinarayan, A. Rajaraman, and J. D. Ullman. Implementing data cubes efficiently. In *Proc. of ACM SIGMOD*, pages 205–216, 1996.
8. R. Kimball. *The Data Warehouse Toolkit*. John Wiley, 1996.
9. Y. Kotidis and N. Roussopoulos. An alternative storage organization for ROLAP aggregate views based on cubetrees. In *Proc. of ACM SIGMOD*, pages 249–258, 1998.
10. I. Lazaridis and S. Mehrotra. Progressive approximate aggregate queries with a multi-resolution tree structure. In *Proc. of ACM SIGMOD*, 2001.
11. Z. X. Loh, T. W. Ling, C.-H. Ang, and S. Y. Lee. Analysis of pre-computed partition top method for range top-k queries in OLAP data cubes. In *Proc. of CIKM*, pages 60–67, 2002.
12. I. S. Mumick, D. Quass, and B. S. Mumick. Maintenance of data cubes and summary tables in a warehouse. In *Proc. of ACM SIGMOD*, pages 100–111, 1997.
13. Ondelette.com. Data Warehousing and OLAP: A research-oriented bibliography. In *http://www.ondelette.com/OLAP/dwbib.html*, 2005.
14. D. Papadias, P. Kalnis, J. Zhang, and Y. Tao. Efficient OLAP operations in spatial data warehouses. In *Proc. of SSTD*, 2001.
15. T. B. Pedersen and N. Tryfona. Pre-aggregation in spatial data warehouses. In *Proc. of SSTD*, pages 460–480, 2001.
16. M. Zhu, D. Papadias, J. Zhang, and D. Lee. Top-k spatial joins. *IEEE TKDE*, 17(4):567–579, 2005.