Materialized Views for Count Aggregates of Spatial Data

Anan Yaagoub[1], Xudong Liu[1], Goce Trajcevski[1,*], Egemen Tanin[2], and Peter Scheuermann[1]

[1] Dept. of Electrical Engineering and Computer Science
Northwestern University
Evanston, USA
{anany, xudong, goce, peters}@eecs.northwestern.edu.com
[2] Dept. of Computing and Information Systems
University of Melbourne
Melbourne, Australia
etanin@unimelb.edu.au

Abstract. We address the problem of efficient processing of *count aggregate* queries for spatial objects in OLAP systems. One of the main issues affecting the efficient spatial analysis is the, so called, *distinct counting problem*. The core of the problem is due to the fact that spatial objects such as lakes, rivers, etc... – and their representations – have *extents*. We investigate the trade-offs that arise when (semi) materialized views of the count aggregate are maintained in a hierarchical index and propose two data structures that are based on the Quadtree indexes: Fully Materialize Views (FMV) and Partially Materialized Views (PMV). Each aims at achieving a balance between the: (1) benefits in terms of response time for range queries; (2) overheads in terms of extra space and update costs. Our experiments on real datasets (Minnesota lakes) demonstrate that the proposed approaches are beneficial for the first aspect achieving up to five times speed-up, while incurring relatively minor overheads with respect to the second one, when compared to the naïve approach.

1 Introduction

A variety of application domains such as Geographic Information Systems (GIS), multimedia and agricultural planning, require management of large amounts of multi-dimensional data [16]. Given the size of the datasets (e.g., Terrabytes of satellite images daily), often there is a need to extract summary data, without actually accessing all of the individual data items in the whole dataset.

In traditional databases, an aggregate function [10] applied to a set of tuples returns a single numeric value summarizing some property of the tuples – e.g., `COUNT`, `SUM`, `MAX`, `MIN`, `AVG` in the SQL-92 standard, with `EVERY`, `SOME` and `ANY` added in the SQL:1999 standard [7]. Using the `GROUP-BY` construct, one can obtain several aggregate values as an answer to a single query. A typical example

* Research supported by NSF – CNS 0910952.

T. Morzy, T. Haerder, and R. Wrembel (Eds.): ADBIS 2012, LNCS 7503, pp. 427–440, 2012.

would be to generate the average salary grouped by a department_ID of a given enterprize. An n-dimensional generalization of the `GROUP-BY` clause was provided by the `CUBE` operator [5], enabling queries that aggregate data across several attributes and at different levels of granularity, for the purpose of efficient Online Analytical Processing (OLAP).

Spatial databases manage objects with spatial extent – e.g., points, lines, polygons, volumes [17]. An important source of problems for storage management and query processing in this setting, in addition to the large quantities of data, stems from the fact that it is not straightforward to define spatial ordering that will capture the semantics of the extents in multiple dimensions [16]. The problem is further accentuated when aggregate queries need to be processed [11], along with warehouse and cubes construction [8, 12, 20]. When it comes to OLAP and spatial data cubes, aggregate queries typically involve a spatial selection predicate specified as a multidimensional (hyper) rectangle and, often, one is interested in a summarized information regarding the objects that overlap partially or completely with the query predicate [11].

The motivation for this work is the observation that due to the spatial extent, when data is grouped in several layers of granularity, the *distinct count problem* [1, 9, 15] may occur when processing spatial range query.

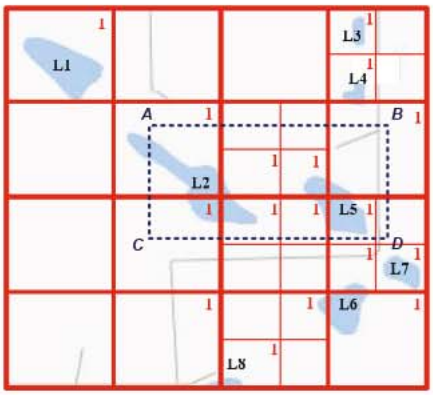


Fig. 1. Distinct Count Problem in OLAP Hierarchy

An illustration is provided in Fig. 1, showing a snapshot of eight lakes (labelled L1 – L8) from Polk County, Minnesota[1]. Assume that the region has been recursively subdivided along each dimension until the subset of (the extent of) each lake can be contained within a single cell, in a manner similar to MX-CIF Quadtree-based subdivision [16]. The top-right corner of each cell shows the value of the count ("1" in this case – although one could extend the threshold of

[1] The figure actually shows a zoomed portion of the Northwest part from `http //minnesotalakes net/LakePages_RR/cable-polk bmp`

the data items to be kept in a leaf-node [6]). Now, consider the following range aggregate query:

Q1: *'Retrieve the number of all the lakes that either intersect or are contained within the rectangular region ABCD'.*

Attempting to answer it based on the count of the individual items in the cells intersected by the rectangle ABCD and adding the count-values, would return **8** as the answer. However, in reality, the answer is 2: lakes L2 and L5 are the only ones that intersect ABCD, but portions of their extents have been counted multiple times due to their participation in different cells.

To alleviate such problems, we propose two approaches that augment the internal nodes of the index – a Quadtree variant [6,16] – with data containing correct values of the distinct count for the children in each subtree, along with the objects' unique identifiers. More specifically, the first approach maintains a Fully Materialized View (FMV) containing detailed description at every level of the subtree. The second approach maintains a Partially Materialized View (PMV(k)) at every k-th level – where k is a user defined parameter. Clearly, non-zero values of k have a potential of introducing an error in the answer to queries similar to the example query **Q1** – unless further descent along the tree is undertaken.

We evaluated the benefits of the proposed approaches in terms of response time for spatial queries of different size, as well as the error-impact of PMV(k) over the real dataset of 11,683 lakes from Minnesota[2]. Our experiments demonstrate that significant speedup can be achieved when answering range queries with our proposed methodologies – up to five times when compared to the naïve method relying on the "standard" PMR Quadtree. On the other hand, the time and space overheads for constructing the FMV and PMV(k) extensions, are less than 20% higher than the requirements of the naïve method in the worst case.

In the rest of this paper, Section 2 presents the background and Section 3 gives the details of our proposed data structures and algorithms. The details of our experimental observations are given in Section 4. Section 5 compares our work with the related literature and Section 6 summarizes the paper and outlines directions for future work.

2 Preliminaries

We now summarize some background information and terminology that we build upon in the rest of this paper. We note that there are comprehensive surveys describing various multidimensional indexing structures [3,16], as well as their impact/role in spatial warehousing and OLAP [12,20] and spatio-temporal aggregates computation [11]. Hence, in the rest of this section we focus on describing the Quadtree [16]: a multidimensional data structure that recursively partitions the d-dimensional space into disjoint regions of equal-sized cells (blocks). Any existing block can be further subdivided into 2^d equal-sized blocks by splitting

[2] Publicly available at: `http://deli.dnr.state.mn.us/data_search.html`

each dimension in half[3]. Figure 1 shows an example of recursive cells splitting in 2-dimensional space – in which case every node of the respective Quadtree is either a leaf node or has four children. Typically, the children of a given node are labelled in accordance with the compas-directions: Northwest (NW), Northeast (NE), Southwest (SW), Southeast (SE).

Many variants of the Quadtree have been proposed in the literature (cf. [16]), differing in various aspects like, for example, implementation of accessing the actual data items (storing the records directly with the leaf nodes vs. encoding them at leaf nodes and providing a linear data structure to index the records [4]). In this work, we implemented a version that was motivated by the PMR Quadtree [6] and MX-CIF Quadtree [16]. Both structures are targeting the indexing of polygons, except, the MX-CIF Quadtree optimizes the maintenance vs. retrieval trade-off for rectangles, whereas the PMR Quadtree is more generic, and applicable to objects of different spatial types (points, lines, polygons). Due to their extent, (portions of) the non-point object may be stored in more than one block in the leaf nodes – essentially "clipping" the object. The PMR Quadtrees can employ an external parameter – *splitting threshold* δ [6] – which specifies the maximum number of objects that can be stored in a leaf node before it needs to be split.

3 Quadtrees with Partially Materialized Aggregate Views

We now present the details of our methodologies for efficient processing of count aggregate queries over spatial data. We assume that the spatial objects are simple polygons (convex or concave, but without holes). We note that overlapping polygons can be split in three disjoint part: one for each of the set-differences, and one for the common intersections [15] – although it is not a requirement in our implementation.

The insertion of a new 2D polygon in an existing Quadtree $\mathcal{Q}$ follows the typical approach of traversing the tree in a top-down manner, visiting each node that intersects the new polygonal object. The object is added to any intersecting leaf node not exceeding the threshold for maximal number of polygons each leaf may point at – δ (cf. [6]). Let P denote a polygon to be inserted and $R(N_i)$ denote the region corresponding to the cell covered by the node N_i in $\mathcal{Q}$, and let $C(N_i)$ (respectively, $S(N_i)$) denote the set of all the children (respectively, siblings) of N_i. For a leaf node N_l, let $I(N_l)$ denote the set of polygons currently pointed to from N_l (note that $I(N_l) \leq \delta$). Algorithm 1 specifies the procedure for inserting a spatial object[4].

Algorithm 1 follows the philosophy of insertion in PMR Quadtree with threshold δ, however, the resulting tree will have similar problems with count aggregate as the ones illustrated in Fig. 1. Namely, given a query region Q_R – processing of this count aggregate would require descending to every leaf node N_{leaf} the cell

[3] This, essentially, makes the Quadtree a *trie* – which need not be the case and implementations with varying cell-sizes exist [16].

[4] Naturally, we assume that the algorithm is invoked with $N_i = root$.

Algorithm 1. INSERT: Insertion of a region in a Quadtree node

Input: $(\mathcal{Q}, N_i, P)$

```
1: if (P ⊂ R(N_i) AND N_i is leaf) then
2:    if |I(N_i)| < δ then
3:       Insert P at N_i
4:    else
5:       Split N_i
6:       for all N_ij ∈ C(N_i) do
7:          for all P_k ∈ I(N_i) do
8:             INSERT(P_k, N_ij)
9:          end for
10:         INSERT(P, N_ij)
11:      end for
12:   end if
13: else if (P ⊂ R(N_i) AND N_i is not a leaf) then
14:   for all N_ij ∈ C(N_i) do
15:      INSERT(P, N_ij)
16:   end for
17: else if (P ∩ R(N_i) ≠ ∅) then
18:   for all N_j ∈ (S(N_i) ∪ {N_i}) do
19:      INSERT(P, N_j)
20:   end for
21: end if
```

of which has non-empty intersection with Q_R, in order to avoid double-counting. We note that although any Quadtree variant splits the cell of a given node in into disjoint regions, the problem stems from the fact that the object with spatial extent may spread over more than one cell. To speed up the processing of the count aggregate for spatial range queries, we propose two variations of the (PMR) Quadtree:

- *Fully Materialized Views* (FMV) in all internal nodes.
- *Partially Materialized Views* at every k-th level (FMV(k)).

The intuition behind the proposed structure(s) is illustrated in Fig. 2, which shows parts of the overall tree. The main observation is that in addition to the "regular" Quadtree that can be used to index the dataset used in Fig. 1, each internal node is augmented with a structure $A(N_i) = (CA, L(CA))$, containing two kinds of values:

- CA = The value of the count aggregate for the objects (polygons) intersecting $R(N_i)$ (also indexed by all the children of N_i).
- A list $L(CA)$ containing all the unique object identifiers (oID) for the objects that are intersecting $R(N_i)$ (also indexed by the children nodes of N_i).

We note that, in practice the organization of the children in the nodes follows an order induced by a Z-curve (a.k.a. Morton code) because, while the leaf-nodes may have the oIDs, the actual objects are indexed based on a linear order, e.g., using a B-tree (cf. [6]). However, for clarity of the illustration we used a simple

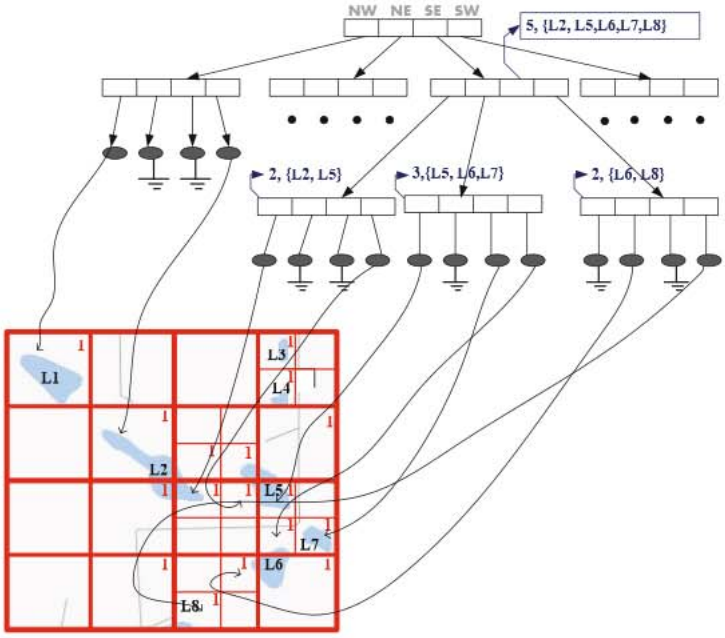


Fig. 2. Fully Materialized Views in Internal Nodes

counter-clockwise ordering for the children nodes. Having $A(N_i)$ as part of the internal nodes will require adaptation of the insertion algorithm with steps that will properly update both $N_i.CA$ and $N_i.L(CA)$ for a given node N_i. The crucial observation is that, due to the spatial extent, an insertion of a single new object in the tree may require modifications of the respective $A(N_i)$'s along several internal nodes/paths. The corresponding insertion procedure is presented in Alg. 2. As can be seen, the main difference in the insertion algorithm for FMV variant is in updating the structure that augments the nodes of the tree (cf. lines 4-5, 8-9, 15-16, 21-22). Clearly, augmenting the node structure in the FMV tree will impose a space overhead that will increase the one of the regular region based Quadtree by a factor of $(O(\log_4 N))$ – and we provide quantitative observations in Sec. 4.

Given Alg. 2, the implementation of the PMV(k) – which is, partially materialized views at every k-th level is fairly straightforward. Namely, let $Depth(N_i)$ denote the depth of the given node N_i – which is, the length of the path from the root to N_i. Then, each of the (pairs of) lines 4-5, 8-9, 15-16 and 21-22 in Alg. 2 needs to be embedded in the body of the IF-clause, where the condition of the IF-clause will be to check whether $Depth(N_i) \equiv 0 \ (mod\ k)$. For a given object ID (oID), let P_{oID} denote the polygonal boundary of the spatial shape corresponding to oID. To determine the value of the count aggregate for a given range Q_R, we first select the children of the root that intersect Q_R ($N_j \in C(Root)$ and $R(N_j) \cap Q_R \neq \emptyset$). Although each of them has the numeric count of the objects readily available, simply adding all such $N_j.CA$ values could yield an incorrect answer, for a two-fold reason:

Algorithm 2. FMV-INSERT: Insertion of a region in a FMV Quadtree node

Input: $(\mathcal{FMV} - \mathcal{Q}, N_i, P)$

```
1: if (P ⊂ R(N_i) AND N_i is leaf) then
2:    if N_i is empty then
3:       Insert P at N_i
4:       N_i.CA = 1
5:       N_i.L(CA) = N_i.L(CA) ∪ {P}
6:    else
7:       Split N_i
8:       N_i.CA = N_i.CA + 1
9:       N_i.L(CA) = N_i.L(CA) ∪ {P}
10:      for all N_ij ∈ C(N_i) do
11:         FMV-INSERT(P, N_ij)
12:      end for
13:   end if
14: else if (P ⊂ R(N_i) AND N_i is not a leaf) then
15:   N_i.CA = N_i.CA + 1
16:   N_i.L(CA) = N_i.L(CA) ∪ {P}
17:   for all N_ij ∈ C(N_i) do
18:      FMV-INSERT(P, N_ij)
19:   end for
20: else if (P ∩ R(N_i) ≠ ∅) then
21:   N_i.CA = N_i.CA + 1
22:   N_i.L(CA) = N_i.L(CA) ∪ {P}
23:   for all N_j ∈ (S(N_i) ∪ {N_i}) do
24:      FMV-INSERT(P, N_j)
25:   end for
26: end if
```

(1) As demonstrated in the example shown in Fig. 1, due to its span, portions of a same object (same *oID*) may span over multiple cells in the subtrees. Thus, adding the $N_j.CA$ may cause a multiple-count of a given *oID*.
(2) Although a particular *oID* may be included in the $N_j.L(CA)$, it may still be outside the intersection region $R(N_j) \cap Q_R$.

It may be tempting to directly check for intersection of Q_R with all the P_{oID} for all the children of the root, that intersect Q_R, this may impose an overhead in terms of invoking the intersection test procedure. Namely, we can safely prune the *grandchildren* of the root (i.e., the children of $N_j \in C(Root)$) that do not intersect Q_R. However, this is already a hint towards recursive descending down the tree which, in certain pathological cases – e.g., too many small regions and/or narrow-band query region – could defeat the purpose of having the materialized views in the internal nodes. Hence, we need a stopping criterion that will strike a balance between recursive descending and the intersection-tests. One possibility that we adopted is as follows: if the area of the intersection of Q_R with the cell of a given node N_p is larger than a certain fraction Θ of that cell's area – i.e., $Area(Q_R \cap R(N_p)) \geq \Theta \cdot Area(R(N_p))$ – the pruning stops and we check every object in $N_p.L(CA)$ for intersection with Q_R. The ideas are formalized in Alg. 3.

Algorithm 3. Get_Candidates: Determine the objects from a subtree which intersect a region

Input: (Tree $\mathcal{QFMV}$, Range Q_R, Node N_i)
Output: (A collection of oIDs from objects intersecting Q_R)

```
1: if (Q_R ∩ R(N_i) = ∅) then
2:    Candidates = ∅
3: else if (Q_R ⊃ R(N_i)) then
4:    Candidates = N_i.L(CA)
5: else if (Area(Q_R ∩ R(N_p)) ≥ Θ · Area(R(N_p))) then
6:    for all oID ∈ N_i.L(CA) do
7:       if (P_oID ∩ Q_R ≠ ∅) then
8:          Candidates = Candidates ∪ {oID}
9:       end if
10:   end for
11: else
12:   for all (N_j ∈ Child(N_i)) do
13:      Child_Candidates = Get_Candidate(QFMV, Q_R, N_j)
14:      Candidates = Candidates ∪ Child_Candidates
15:   end for
16: end if
17: return Candidates
```

Again, we note that Alg. 3 will be initially invoked with the root of the $\mathcal{QFMV}$ tree. Upon returning the list collection of the candidates, its cardinality is the desired value of the `COUNT DISTINCT` aggregate with respect to the range Q_R.

When we have a tree with partially materialized views at every k-th level from the root, – i.e., PMV(k) for a given k – Alg. 3 needs a slight modifications. Namely, the very fist line needs to be preceded with the test whether $Depth(N_i) \equiv 0 \ (mod\ k)$. If so, the "regular" Get_Candidates function is executed as outlined in Alg. 3 itself. Otherwise, assuming that $Depth(N_i) \equiv j \ (mod\ k)$, we will need a loop that will generate a collection of all the descendants of N_i at level $Depth(N_i) + (k - j)$, and iteratively execute Alg. 3 over the elements of the collection.

An observation of relevance for both FMV and PMV(k) variants is that the statements in lines 5, 9, 16 and 23 all include the union ($\cup$) operation. The meaning is the traditional one – which is, the union operation does not generate duplicates. In reality, this would require an implementation of an algorithm for duplicates elimination. In the current stage of our work, we relied on maintaining the L(CA) list sorted by the oID value with every new insertion. This enabled us to implement the union via straightforward merging of lists when executing Alg. 3. However, this may not be the most efficient way of updating the index from the perspective of bulk-insertion – which is something that we are planning to investigate in the future.

We conclude this section with a couple of final observations regarding Alg. 3. Firstly, we note that, in the special cases at which the query region completely

coincides with a particular cell of a given node N_i – we need not execute Alg. 3, since the value of the $N_i.CA$ can be immediately returned as an answer. Secondly, one may be tempted to simply add the values of the "CA" attribute of the intersecting nodes, without checking for duplicates and/or non-existing intersections. Clearly, this will incur an error in the results. However, having (partially) materialized views will decrease the severity of that error, as our experiments will illustrate in Sec. 4.

4 Experimental Evaluation

Fig. 3. Map of the Dataset

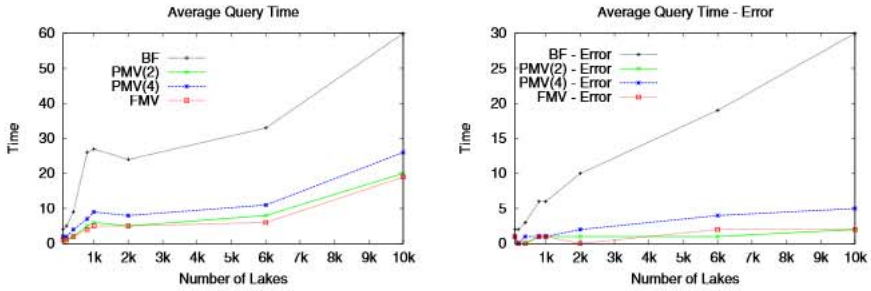
We now proceed with presenting the observations from the experimental evaluation of our proposed approaches. Our experiments were conducted on an Intel Core i7 CPU machine with 8GB memory, with 8 dual-core processors at 2.20GHz running 64-bit windows 7.0 operating system, and the implementations were done in Java.

The spatial dataset used in the experiments consists of 11,386 lakes (cf. Fig. 3) the boundaries of which are approximated with polygons. The size of the polygons varied from 100 to 4000 edges. The dataset was obtained from Minnesota Department of Natural Resources (http://deli.dnr.state.mn.us/), and the file size is 13.5 MB.

For the experiments we generated (sub)sets with 10 different cardinalities by randomly selecting objects from the original dataset. We note that even for different query shapes, the use of MBR to prune irrelevant subregions is likely so, to evaluate the benefits of the proposed approaches, we used rectangles with variable sizes in terms of the percentage of the total area of interest (5%, 10%, 20%, 25% and 50%). At the current stage, the threshold for the leaf nodes (δ) was set to 4, and the parameter Θ (cf. Alg. 3) was fixed to 50% (approximately 70% overlap on each dimension) – the more detailed investigation of its impact is left for a future work. We note that both the source code and the datasets are publicly available at:
http://www.sharpedgetech.com/Quadtree_Views

We compare the proposed approaches: FMV and PMV(k) for two values of the parameter k: k = 2 and k = 4, against the naïve approach – which is, the quadtree with no augmented inner nodes, labelled as "BF" (Brute-Force) in the corresponding figures. The values of the time in all of the subsequent graphs are in milliseconds.

Our first set of experimental observations pertains to the response time for generating the answer to the count-aggregate. Figure 4(a) shows the average



(a) Average times – exact answer (b) Average times – with error

Fig. 4. Response time for range queries – averaged values

response time (averaged over the query polygons used) as a function of the size of the dataset. As can be observed, a speed up of over 4 times can be achieved with the FMV, and over 3 times when the materialized views are used at every other level of the tree (PMV(2)). The second part, Fig. 4(b) illustrates the results of similar settings, except now we tolerated errors in the answer – the value of the count-aggregate. Clearly, the obtained speed-up is even higher in this scenario (factor of up to 10 for FMV and up to 7 for PMV(2)).

Figure 5 illustrates the impact of the size of the query region Q_R on the response time. As can be seen, the speed up is significantly higher for large query regions – which conforms to the intuitive expectations. Note that for smaller query regions, the response time of PMV(2) is rather close to the one of FMV.

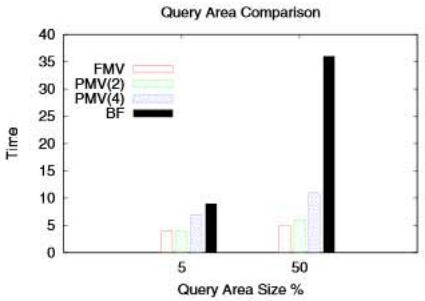
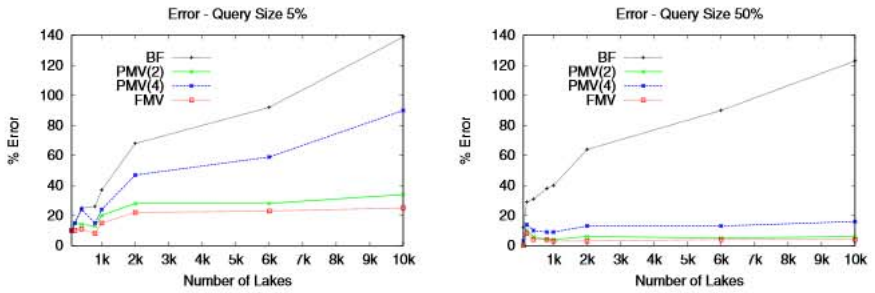


Fig. 5. Impact of the size of the range query

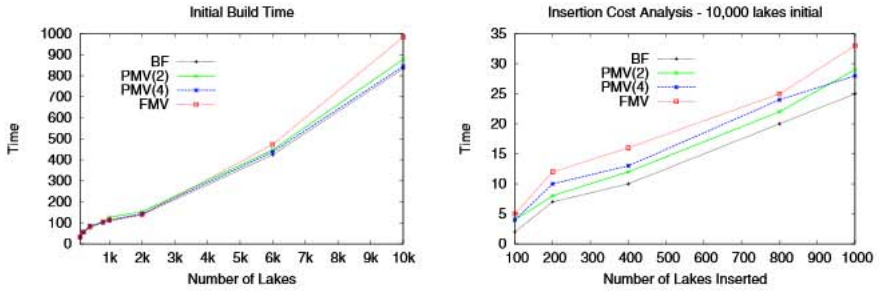
Fig. 6, illustrates the robustness of the proposed approaches in terms of accumulating errors for the count-aggregate value. The two sub-figures correspond to the cases of small-size (Fig. 6(a)) and large-size query regions (Fig. 6(b)). Apparently, the larger the query region, the less error is incurred in the calculation of the count-aggregate.

Our next set of experiments pertains to the overheads in terms of constructing the FMV and PMV(2), as well as PMV(4), when compared to the plain PMR-based Quadtree. As expected, the naïve approach is fastest – both when we consider construction of the index for the entire data set (cf. Fig. 7(a)) as well as "incremental" construction, shown in Fig. 7(a). For the incremental construction, we selected 10,000 objects from the dataset, constructed the respective structures, and then kept inserting batches of objects in sizes indicated on the



(a) Query region = 5% (b) Query Region = 50%

Fig. 6. Impact of the error tolerance on response time for small and large query regions



(a) Initial construction (b) Updates

Fig. 7. Index Maintanance

X-axis in Fig. 7(a). Note that while FMV exceeds the index build-up time by up to 20% for the complete dataset (25% in the "incremental" case), the performance of the PMV(2) is only 3-7% worse than the naïve method (up to 15% for the incremental case).

The last set of observations pertains to the trade-off in terms of space requirements. Table 1 presents the number of *oID* values stored for each of the methodologies used in our experiments, for a collection of a subsets from the dataset with different cardinalities, as shown in the first column. As expected, the naïve method has the smallest space-overhead. The number of *oID*s is always (progressively with the size) larger than the cardinality of the dataset – this is because objects (lakes) span over > 1 cell. The FMV needs to store anywhere between 4.5 and up to 6.3 times more *oID*s than the naïve one in the internal nodes. The partially materialized views require storage overhead in terms of *oID*s in the internal nodes which are within factors of 2 (PMV(4)) and 3 (PMV(2)). However, those are *oID*s which, assuming 32-bits size for integers, would yield $< 4Mb$ overhead for the worst-case of FMV tree (117,959 nodes) – which is fraction of the size of the dataset (13.5MB).

Table 1. Space-overheads for materialized views

Cardinality	Depth	#*oID*s BF	#*oID*'s FMV	#*oID*s PMV(2)	#*oID*s PMV(4)
100	10	112	716	393	237
200	11	258	1,561	841	545
1,000	12	1,404	8,165	4,655	2,898
2,000	13	3,236	17,631	10,290	6,493
10,000	18	21,875	102,577	62,388	40,969
All: 11,386	18	25,319	117,959	72,829	49,074

Summarizing our experimental observations, it appears that maintaining partially materialized views in the internal nodes of the indexing structure may be a viable option for spatial OLAP. While there are certainly overheads incurred, the processing time when calculating the exact value of the count aggregate gains significant speed-up. Taking the naïve approach and the FMV as two end-points of a spectrum, the choice of k for the PMV(k) method will clearly affect the possible trade-offs.

5 Related Work

Efficient aggregates' computations is a problem of great interest for many applications that deal with OLAP and warehousing of spatial (and spatio-temporal) data [2,9,12,13,17,20,21]. In a certain sense, they have provided a new domain for investigating efficient multidimensional indexing structures [16].

Several techniques have been introduced addressing the distinct count problem. For example, in [18] sketches were proposed as a viable option – however, the approach provides only approximate solution for the distinct count. Another approach for efficiently generating approximate solutions to spatio-temporal aggregates was presented in [2]. In this paper, we aimed at providing exact solution and investigated different trade-offs that arise towards that end. Euler Histograms (cf. [1]) are another formalism that has been used for aggregates computation in spatial and spatio-temporal settings [21] – however, a drawback is that they do not work for concave shapes.

An approach very similar in spirit to ours is based on the aR-Tree in [14]. Using R-trees as a basic indexing mechanism, the aR-Tree extends the internal nodes with numeric values corresponding to the value of the desired aggregate. The work identifies three categories of aggregates: distributive, algebraic and holistic, and demonstrates the nice properties of distributive values. The approaches are close to our "view" concept, and the reported experiments (using R* tree variant) used datasets consisting of polylines, without addressing the correctness of distinct count for polygons. The main difference of our work (aside from using Quadtree-based indexing) is that we were concerned with the issue of the materialized views in the internal nodes and the trade-offs that arise. The ap-Tree variant was proposed in [19], however, the work addressed efficient aggregates computations of points data.

6 Concluding Remarks

We addressed the problem of efficient computation of the distinct count aggregate value for spatial shapes in OLAP settings. Specifically, we proposed augmenting the (variants of the region) Quadtree [6,16] with materialized views in the internal node, containing data relevant for calculating the desired exact value. In addition to the "extreme" FMV approach of maintaining fully materialized views at every level, we proposed the PMV(k) approach which maintains views only at subset of the levels. We investigated and quantified the impacts of the views maintenance in terms of the space overheads and index construction/update times. Our experiments demonstrated that the proposed approaches yield significant speedups, with reasonable overheads in terms of index maintenance and space. Even if errors in the answer to count aggregate are to be tolerated, our approaches yield significantly higher precision.

Part of our future work is focusing on deriving better insights regarding the optimization of the values of k (cf. PMV(k)) and Θ parameters. Clearly, this will need to include properly formulated costs – however, in many applications the data may be disk-resident and we will need to incorporate the accessing and page-transfer in the cost. Another extension is to consider distribution of large datasets and/or index in cloud-based trajectory warehouse. Lastly – one would expect that the exact computation of certain aggregates (coupled with selection predicates) would benefit more from a particular index-structure than others – and, towards this, we plan to conduct a detailed comparative evaluation of indexing structures for spatial OLAP computations (e.g., [14,22]), along with efficient bulk-updates for each [6].

References

1. Beigel, R., Tanin, E.: The Geometry of Browsing. In: Lucchesi, C.L., Moura, A.V. (eds.) LATIN 1998. LNCS, vol. 1380, pp. 331–340. Springer, Heidelberg (1998)
2. Braz, F., Orlando, S., Orsini, R., Raffaetà, A., Roncato, A., Silvestri, C.: Approximate aggregations in trajectory data warehouses. In: ICDE Workshops, pp. 536–545 (2007)
3. Gaede, V., Günther, O.: Multidimensional access methods. ACM Comput. Surv. 30(2), 170–231 (1998)
4. Gargantini, I.: An effective way to represent quadtrees. Commun. ACM 25(12), 905–910 (1982)
5. Gray, J., Chaudhuri, S., Bosworth, A., Layman, A., Reichart, D., Venkatrao, M., Pellow, F., Pirahesh, H.: Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub totals. Data Min. Knowl. Discov. 1(1), 29–53 (1997)
6. Hjaltason, G.R., Samet, H.: Speeding up construction of PMR quadtree-based spatial indexes. VLDB J. 11(2), 109–137 (2002)
7. ANSI/ISO international standard. Database language SQL (1999), `http://webstore.ansi.org`
8. Jensen, C.S., Pedersen, T.B., Thomsen, C.: Multidimensional Databases and Data Warehousing. Morgan & Claypool (2012)

9. Khatri, V., Ram, S., Snodgrass, R.T., O'Brien, G.M.: Supporting user-defined granularities in a spatiotemporal conceptual model. Ann. Math. Artif. Intell 36(1-2), 195–232 (2002)
10. Klug, A.C.: Equivalence of relational algebra and relational calculus query languages having aggregate functions. J. ACM 29(3), 699–717 (1982)
11. López, I.F.V., Snodgrass, R.T., Moon, B.: Spatiotemporal aggregate computation: a survey. IEEE Trans. Knowl. Data Eng. 17(2), 271–286 (2005)
12. Malinowski, E., Zimanyi, E.: Advanced Data Warehouse Design From Conventional to Spatial and Temporal Applications (Data-Centric Systems and Applications). Springer (2008)
13. Orlando, S., Orsini, R., Raffaetà, A., Roncato, A., Silvestri, C.: Spatio-temporal Aggregations in Trajectory Data Warehouses. In: Song, I.-Y., Eder, J., Nguyen, T.M. (eds.) DaWaK 2007. LNCS, vol. 4654, pp. 66–77. Springer, Heidelberg (2007)
14. Papadias, D., Kalnis, P., Zhang, J., Tao, Y.: Efficient OLAP Operations in Spatial Data Warehouses. In: Jensen, C.S., Schneider, M., Seeger, B., Tsotras, V.J. (eds.) SSTD 2001. LNCS, vol. 2121, pp. 443–459. Springer, Heidelberg (2001)
15. Pedersen, T.B., Tryfona, N.: Pre-aggregation in Spatial Data Warehouses. In: Jensen, C.S., Schneider, M., Seeger, B., Tsotras, V.J. (eds.) SSTD 2001. LNCS, vol. 2121, pp. 460–480. Springer, Heidelberg (2001)
16. Samet, H.: Foundations of Multidimensional and Metric Data Structures. Morgan Kaufmann (2006)
17. Shekhar, S., Chawla, S.: Spatial Databases: A Tour. Prentice Hall (2003)
18. Tao, Y., Kollios, G., Considine, J., Li, F., Papadias, D.: Spatio-temporal aggregation using sketches. In: ICDE, pp. 214–225 (2004)
19. Tao, Y., Papadias, D., Zhang, J.: Aggregate Processing of Planar Points. In: Jensen, C.S., Jeffery, K., Pokorný, J., Šaltenis, S., Bertino, E., Böhm, K., Jarke, M. (eds.) EDBT 2002. LNCS, vol. 2287, pp. 682–700. Springer, Heidelberg (2002)
20. Vaisman, A., Zimányi, E.: What Is Spatio-Temporal Data Warehousing? In: Pedersen, T.B., Mohania, M.K., Tjoa, A.M. (eds.) DaWaK 2009. LNCS, vol. 5691, pp. 9–23. Springer, Heidelberg (2009)
21. Xie, H., Tanin, E., Kulik, L.: Distributed histograms for processing aggregate data from moving objects. In: MDM, pp. 152–157 (2007)
22. Zhang, D., Tsotras, V.J., Gunopulos, D.: Efficient aggregation over objects with extent. In: PODS, pp. 121–132 (2002)