Indexing and Retrieval of Historical Aggregate Information about Moving Objects*

Dimitris Papadias[†], Yufei Tao[†], Jun Zhang[†], Nikos Mamoulis[§], Qiongmao Shen[†], and Jimeng Sun[†]

[†]Department of Computer Science
Hong Kong University of Science and Technology
{dimitris, taoyf, zhangjun, qmshen,jimeng}@cs.ust.hk

[§]Department of Comp. Science and Inf. Systems
University of Hong Kong
nikos@csis.hku.hk

Abstract

Spatio-temporal databases store information about the positions of individual objects over time. In many applications however, such as traffic supervision or mobile communication systems, only summarized data, like the average number of cars in an area for a specific period, or phones serviced by a cell each day, is required. Although this information can be obtained from operational databases, its computation is expensive, rendering online processing inapplicable. A vital solution is the construction of a spatiotemporal data warehouse. In this paper, we describe a framework for supporting OLAP operations over spatiotemporal data. We argue that the spatial and temporal dimensions should be modeled as a combined dimension on the data cube and present data structures, which integrate spatiotemporal indexing with pre-aggregation. While the well-known materialization techniques require a-priori knowledge of the grouping hierarchy, we develop methods that utilize the proposed structures for efficient execution of ad-hoc group-bys. Our techniques can be used for both static and dynamic dimensions.

1 Introduction

The motivation of this work is that many (if not most) current applications require summarized spatio-temporal data, rather than information about the locations of individual points in time. As an example, traffic supervision systems need the number of cars in an area of interest, rather than their ids. Similarly mobile phone companies use the number of users serviced by individual cells in order to identify trends and prevent potential network congestion. Other spatio-temporal applications are by default based on arithmetic data rather than object locations. As an example consider a pollution monitoring system. The readings from several sensors are fed into a database which arranges them in regions of similar or identical values. These regions should then be indexed for the efficient processing of queries such as "find the areas near the center with the highest pollution levels yesterday".

The potentially huge amount of data involved in the above applications calls for pre-aggregation of results. In direct analogy with relational databases, efficient OLAP operations require materialization of summarized data. The motivation is even more urgent for spatio-temporal databases due to several reasons. First, in some

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

* This work was supported by grants HKUST 6081/01E and HKUST 6070/00E from Hong Kong RGC.

cases, data about individual objects should not be stored due to legal issues. For instance, keeping the locations of mobile phone users through history may violate their privacy. Second, the actual data may not be important as in the traffic supervision system discussed. Third, although the actual data may be highly volatile and involve extreme space requirements, the summarized data are less voluminous and may remain rather constant for long intervals, thus requiring considerably less space for storage. In other words, although the number of moving cars (or mobile users) in some city area during the peak hours is high, the aggregated data may not change significantly since the number of cars (users) entering is similar to that exiting the area. This is especially true if only approximate information is kept, i.e., instead of the precise number we store values to denote ranges such as high, medium and low traffic.

Throughout the paper we assume that the spatial dimension at the finest granularity consists of a set of regions (e.g., road segments in traffic supervision systems, areas covered by cells in mobile communication systems etc.). The raw data provide the set of objects that fall in each region every timestamp (e.g., cars in a road segment, users serviced by a cell). Queries ask for aggregate data over regions that satisfy some spatio-temporal condition. A fact that differentiates spatio-temporal, from traditional OLAP is the lack of predefined hierarchies (e.g., product types). These hierarchies are taken into account during the design of the system so that queries of the form "find the average sales for all products grouped-by product type" can be efficiently processed. An analogy in the spatio-temporal domain would be "find the average traffic in all areas in a 1km range around each hospital".

The problem is that the positions and the ranges of spatio-temporal query windows usually do not conform to pre-defined hierarchies, and are not known in advance. Another query, for instance, could involve fire emergencies, in which case the areas of interest would be around fire departments (police stations and so on). In the above example, although the hierarchies are ad-hoc, the spatial dimension is fixed, i.e., there is a static set of road segments. In other applications, the spatial dimensions may be volatile, i.e., the regions at the finest granularity may evolve in time. For instance, the area covered by a cell may change according to weather conditions, extra capacity allocated etc. This dynamic behavior complicates the development of spatio-temporal data warehouses.

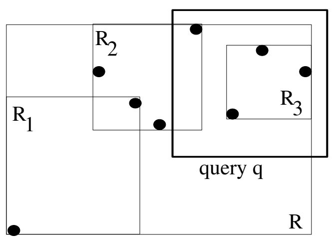
This paper addresses these problems by proposing several indexing solutions. First, we describe spatial trees suitable for the retrieval of aggregate information at a single timestamp. Then, we deal with static spatial dimensions focusing on queries that ask for historical aggregated data in a query window over a continuous time interval. An example would be "give me the number of cars in the city center during the last hour". For such queries we develop multi-tree indexes that combine the spatial and temporal dimensions. In contrast with traditional OLAP solutions, we use the index structure to define hierarchies and we store pre-aggregated data in internal nodes. Finally, we extend our techniques to volatile regions that change over time.

Depending on the type of queries posed, a spatio-temporal OLAP system should capture different types of summarized data. Since our focus is on indexing, we assume some simple aggregate functions like count, or average. In more complex situations we could also store additional measures including the source and the destination of data, direction of movement and so on. Such information will enable analysts to identify certain motion and traffic patterns which cannot be easily found by using the raw data. The proposed methods can be modified for this case. The rest of the paper is organized as follows. Section 2 describes aggregate spatial access methods, while Section 3 proposes indexing techniques for spatio-temporal data, applicable in the presence of static regions. Section 4 discusses structures for volatile regions and Section 5 concludes the paper with a discussion on future work.

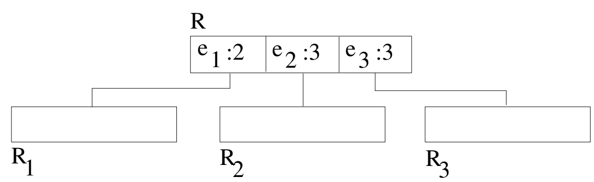
2 Spatial Aggregate Structures

A *window aggregate query* (WA for short) returns summarized information about objects that fall inside the query window, for example the number of cars in a road segment, the average number of mobile phone users per city block etc. An obvious approach to answer such queries is to first retrieve the actual objects by performing traditional window queries, and then compute the aggregate function. This, however, entails a lot of unnecessary effort, compromising performance. A solution for the problem is to store aggregate information in the nodes of specialized index structures.

The aggregate R-tree [8] improves the original R-tree [4, 3] towards aggregate processing by storing, in each intermediate entry, summarized data about objects residing in the subtree. In case of the *count* function, for example, each entry stores the number of objects in its subtree (the extension to any non-holistic functions is straightforward). Figure 1a shows a simple example where 8 points are clustered into 3 leaf nodes R_1, R_2, R_3, which are further grouped into a root node R. The solid rectangles refer to the MBR of the nodes. The corresponding R-tree with intermediate aggregate numbers is shown in Figure 1b. Entry $e_1 : 2$, for instance, means that 2 points are in the subtree of e_1 (i.e., node R_1). Notice that each point is counted only once, e.g., the point which lies inside the MBRs of both R_1 and R_2 is added to the aggregate result of the node where it belongs (e_1). The WA query represented by the bold rectangle in Figure 1a is processed in the following manner. First the root R is retrieved and each entry inside is compared with the query rectangle q. One of the 3 following conditions holds: (i) the (MBR of the) entry does not intersect q (e.g., entry e_1) and its sub-tree is not explored further; (ii) the entry partially intersects q (e.g., entry e_2) and we retrieve its child node to continue the search; (iii) the entry is contained in q (e.g., entry e_3), in which case, it suffices to add the aggregate number of the entry (e.g., 3 stored with e_3) without accessing its subtree. As a result, only two node visits (R and R_2) are necessary. Notice that conventional R-trees would require 3 node visits.



(a) Clustering of points



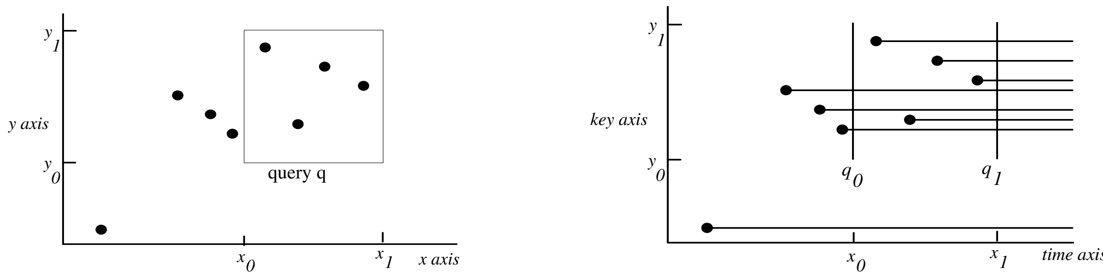
(b) The corresponding aR-tree

Figure 1: An aR-tree example

In summary, the improvement of the aR-tree over the conventional R-tree is that we do not need to visit the nodes (whose MBRs are) inside the query window, but only those nodes that intersect the edges of the window. The cost savings obviously increase with the size of the query window, an important fact because OLAP queries often involve large ranges. Notice, however, that despite the improvement of the aR-tree, query performance is still sensitive to the window size since, the larger the window, the higher the number of node MBRs that are expected to intersect its sides. Another structure, the aP-tree [10], overcomes this problem (i.e., the cost is independent of the query extent) by transforming points to intervals in the key-time plane as follows: the y-coordinate of the point can be thought of as a key value, while the x-coordinate represents the starting time of the interval. The ending time of all intervals is the current time (lying on the right boundary of the time axis). Figure 2a shows the points used in the example of Figure 1a, and Figure 2b illustrates the resulting intervals.

The original query is also transformed since the goal now is to retrieve the number of intervals that intersect the vertical line segment q_1 but not q_0. The intervals are stored using a variation multi-version B-trees [1] enhanced with aggregate information in intermediate entries. Query processing can be reduced to the vertical line segment intersection problem optimally solved by the multi-version B-tree, except that here we are interested in the aggregate number, instead of the concrete ids, of the qualifying objects. This fact differentiates query processing since we can avoid the retrieval of the actual objects intersecting q_1 and q_0 and the expensive computation of their set difference. The evaluation of [10] suggests that the aP- is faster than aR-tree at the expense of space consumption, which is $O(nlogn)$ (n is the number of records) as opposed to $O(n)$ for the aR-tree.

The *window-interval aggregate* query (WIA for short) is the natural extension of WA queries in the spatio-temporal domain. In particular, a WIA query (q_s,q_t) retrieves historical summarized information about objects



(a) Original points and query

(b) Transformed intervals and queries

Figure 2: Transformation of the problem

that fall inside the query window q_s during interval q_t. The next section discusses structures that can efficiently process such queries.

3 Indexing Static Spatial Dimensions

The most common conceptual model for data warehouses is the multidimensional data view. In this model, there is a set of numerical *measures* which are the items of analysis, for example *number* of *objects* (cars or mobile phone users). A measure depends on a set of dimensions, *Region* and *Time*, for instance. Thus, a measure is a value in the multidimensional space which is defined by the dimensions. Each dimension is described by a domain of attributes (e.g. days). The set of attributes may be related via a hierarchy of relationships, a common example of which is the temporal hierarchy (day, month, year). Figure 3 illustrates a simple case; observe that although the regions are 2-dimensional, they are mapped as one dimension in the warehouse. Region R_1 contains 150 objects during the first two timestamps and this number gradually decreases. The star schema [6] is a common way to map multi-dimensional data onto a relational database. A main table (called *fact table*) F, stores the multidimensional array of measures, while auxiliary tables $D_1, D_2, \ldots, D_n$ store the details of the dimensions. A tuple in F has the form $\langle D_i[].key, M[]\rangle$ where $D_i[].key$ is the set of foreign keys to the dimension tables and $M[]$ is the set of measures.

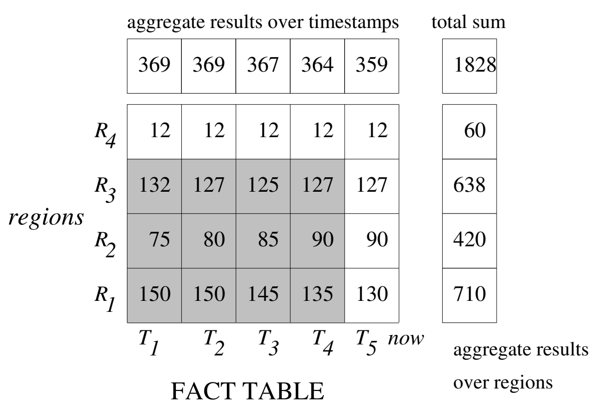


Figure 3: A data cube example

OLAP operations ask for a set of tuples in F, or for aggregations on groupings of tuples. Assuming that there is no hierarchy in the dimensions of the previous example, we identify four possible groupings: i) Group-by Region and Time, which is identical to F, ii-iii) group-by Region (Time), which corresponds to the projection of F on the *region* (*time*) -axis, and iv) the aggregation over all values of F which is the projection on the origin. Figure 3 depicts these groupings assuming that the aggregation function is $count$. The fact table, together with all possible combinations of group-bys, compose the *data cube* [5]. Although all groupings can be derived from F, in order to accelerate query processing some results may be pre-computed and stored as *materialized views*.

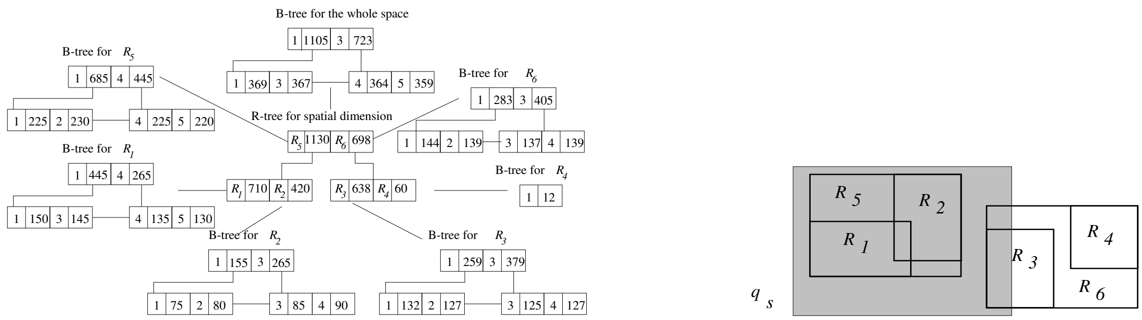
Since, the spatial dimension has no one-dimensional order we store the table in the secondary memory ordered by time and build a B-tree index to locate the blocks containing information about each timestamp. The processing of a typical WIA query employs the B-tree index to retrieve the blocks (i.e., table columns) containing information about q_t and then all regions are scanned sequentially. The aggregate data of those qualifying q_s is accumulated in the result. In the sequel, we refer to this approach as *column scanning*. An alternative approach, which achieves simultaneous indexing on both spatial and temporal dimensions, can be obtained by the generalization of the aR-tree to 3-dimensional space.[1] In particular, each entry r of the *aggregate 3DR-tree* (a3DR-tree) has the form $\langle r.MBR, r.pointer, r.lifespan, r.aggr[]\rangle$, i.e., for each region it keeps the aggregate value and the interval during which this value is valid. Whenever the aggregate information about a region changes, a new entry is created. Using the example of Figure 3, four entries are required for R_1: one for timestamps 1 and 2 where the aggregate value remains 150, and three more entries for the other timestamps where the aggregate value changes. Although the a3DR-tree integrates spatial and temporal dimensions in the same structure (and is, therefore, expected to be more efficient than column scanning for WIA queries that involve both conditions), it has the following drawbacks: (i) it wastes space by storing the MBR each time there is an aggregate change (e.g., the MBR of R_1 is stored four times), and (ii) the large size of the structure and the small fanout of the nodes compromises query efficiency.

In order to overcome these problems, we present a novel multi-tree structure, the *aggregate R- B-tree* (aRB-tree), which is based on the following concept: the regions that constitute the spatial hierarchy are stored only once and indexed by an R-tree. For each entry of the R-tree (including intermediate level entries), there is a pointer to a B-tree which stores historical aggregated data about the entry. In particular, each R-tree entry r has the form $\langle r.MBR, r.pointer, r.btree, r.aggr[]\rangle$ where $r.MBR$ and $r.pointer$ have their usual meaning; $r.aggr[]$ keeps summarized data about r accumulated over all timestamps (e.g., the total number of objects in r throughout history), and $r.btree$ is a pointer to the B-tree which keeps historical data about r. Each B-tree entry b, has the form $\langle b.time, b.pointer, b.aggr[]\rangle$ where $b.aggr[]$ is the aggregated data for $b.time$. If the value of $b.aggr[]$ does not change in consecutive timestamps, it is not replicated.

Figure 4a illustrates an aRB-tree using the data of the cube in Figure 3. For instance, the number 710 stored with the R-tree entry R_1, denotes that the total number of objects in R_1 is 710. The first leaf entry of the B-tree for R_1 (1, 150) denotes that the number of objects in R_1 at timestamp 1 is 150. Similarly the first entry of the top node (1, 445) denotes that the number of objects during the interval [1,3] is 445. The same information is also kept for the intermediate entries of the R-tree (i.e., R_5 and R_6). The topmost B-tree corresponds to the root of the R-tree and stores information about the whole space. Its role is similar to that of the extra row in Figure 3, i.e., answer queries involving only temporal conditions.

The aRB-tree facilitates the processing of WIA queries, by eliminating the need to visit nodes which are totally enclosed by the query. As an example, consider that a user is looking for all objects in some region overlapping the (shaded) query window q_s of Figure 4b during the time interval [1,3]. Search starts from the root of the R tree. Entry R_5 is totally contained inside the query window and the corresponding B-tree is retrieved. The top node of this B-tree has the entries (1, 685), (4, 445) meaning that the aggregated data correspond to the intervals [1,3], [4,5]. Therefore, the next level of the B-tree does not need to be accessed and the contribution of R_5 to the query result is 685. The second root entry of the R-tree, R_6, partially overlaps the query window so the corresponding node is visited. Inside this node only entry R_3, intersects q_s, and its B-tree is retrieved. The first entry of the top node suggests that the contribution of R_3, for the interval [1,2] is 259. In order to complete the result we will have to descend the second entry and retrieve the aggregate value of R_3 for timestamp 3 (i.e., 125). The final result (i.e., total number of objects in these regions in the interval [1,3]) is the sum 685+259+125.

[1] For the following discussion we assume aR-trees as the spatial aggregate structure because the aP-tree cannot be easily generalized to more than two dimensions.



(a) aRB-tree (b) example query

Figure 4: Example of aRB-tree

This corresponds to the sum of aggregate data in the gray cells of Figure 3.

If the aggregate data is not very dynamic, the size of structure is expected to be smaller than the data cube because it does not replicate information that remains constant for adjacent intervals. Even in the worst case that the aggregate data of all regions change each timestamp, the size of aRB-trees is about double that of the cube since the leafs (needed also for the cube) consume at least half of the space. Furthermore, aRB-trees are beneficial regardless of the selectivity, since: (i) if the query window (q_s, q_t) is large, many nodes in the intermediate levels of the aRB-tree will be contained in (q_s, q_t) so the pre-calculated results are used and visits to the lower tree levels are avoided; (ii) If (q_s, q_t) is small, the aRB-tree behaves as a spatio-temporal index. This is also the case for queries that ask for aggregated results at the finest granularity. Next, we extend these concepts for volatile regions.

4 Indexing Dynamic Spatial Dimensions

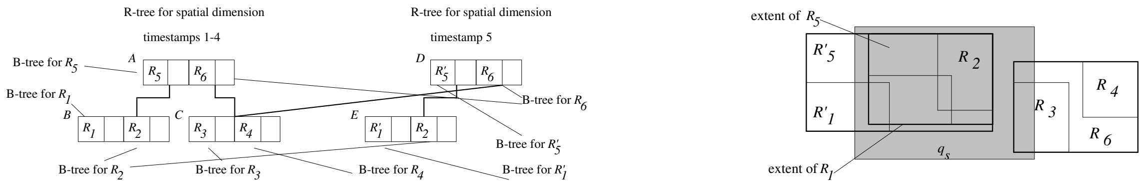
In this section we consider that the finest granularity regions in the spatial dimension, can change their extents over time and/or new regions may appear/disappear. Obviously, when the leaf-level regions change, the spatial tree structure is altered as well. We propose two solutions to this problem by employing alternative multi-tree indexes.

4.1 The aggregate Historical RB-tree

A simple approach to deal with volatile regions is to create a new R-tree every time there is a change. Assume that at timestamp 5, region R_1 is modified to R_1' and this update alters the father entry R_5 to R_5'. Then, a new R-tree is created at timestamp 5, while the first one dies. In order to avoid replicating the objects that were not affected by the update, we propose the *aggregate Historical R-B-tree* (aHRB-tree), which combines the concepts of aRB-trees and HR-trees [7]. For example in Figure 5a, the two R-trees share node C, because the extents of regions R_3 and R_4 did not change. Each node[2] in the HR-tree, stores a lifespan, which indicates its valid period in history. The lifespans of nodes A and B are [1,4], while that of C is [1,*), where * means that the node is valid until the current time. The form of the entries is the same as in aRB-trees except that $r.aggr[]$, keeps aggregated information about the entry during the lifespan of the node that contains it, instead of the whole history.

Assume that the current time is after timestamp 5, and a query asks for objects in some region overlapping the query window q_s of Figure 5b during the time interval [1,5]. The figure illustrates the old and the new versions after the update at timestamp 5. Both R-trees of Figure 5a are visited. In the first tree, since R_5 is inside

[2] *Historical R-trees* (HR-trees) [7] decrease the level of redundancy by allowing consecutive R-trees to share common branches. Although traditional HR-trees do not store lifespans, we need this information in order to record the validity period of aggregate data in the R-tree nodes and avoid visiting the B-trees.



(a) aHRB-tree (b) example query

Figure 5: Example of aHRB-tree

q_s its child node B is not accessed. Furthermore, as the lifespan of R_5 (i.e., [1,4]) is entirely within the query interval, we retrieve the aggregate data in R_5 without visiting its associated B-tree. On the other hand, node C is accessed (R_6 partially overlaps q_s) and we retrieve the aggregate value of R_3 (for interval [1,5]) from its R-tree entry. Searching the subsequent R-trees is similar, except that shared nodes are not accessed. Continuing the above example, node E is reached and the B-trees of R'_1 and R_2 are searched, while we do not follow the pointer of R_6 (to node C) as C is already visited.[3]

Notice that independently of the query length (q_t), in the worst case the algorithm will visit the B-trees of two R-trees. These are the R-trees at the two ends of q_t. The lifespans of nodes in the trees for intermediate timestamps of q_t are entirely contained in q_t, so the relevant aggregate data stored with the R-tree entries are used directly. Furthermore, although in Figure 5a we show a separate B-tree for each HR-tree entry, the B-trees of various entries may be stored together in a space efficient storage scheme, described [9].

4.2 The aggregate 3DRB-tree

In HR-trees, a node (e.g., B) will be duplicated even if only one of its entries (e.g., R_1) changes. This introduces data redundancy and increases the size of aHRB-trees. The a3DRB-tree (*aggregate 3-dimensional R-B-tree*) avoids this problem, by combining B-trees with 3DR-trees. Every version of a region is modeled as a 3D box, so that the projection on the temporal axis corresponds to a time interval when the spatial extents of the region are fixed; different versions/regions are stored as distinct entries in the 3DR-tree. In particular, a 3DR-tree entry has the form $\langle r.MBR, r.lifespan, r.pointer, r.btree, r.aggr[]\rangle$, where $r.MBR$, $r.pointer$, $r.btree$ are defined as in aRB-trees; $r.aggr[]$ stores data over $r.lifespan$.[4] A typical query involving both spatial and temporal aspects ("find the total number of objects in the regions intersecting some window q_s during a time interval q_t") is also modeled as a 3D box.

Although both aHRB- and a3DRB- trees are aimed at volatile regions they have two important differences: (i) a3DRB-trees maintain a large 3DR-tree for the whole history, while aHRB-trees maintain several small trees, each responsible for a relatively short interval. This fact has implications on their query performance. (ii) The aHRB-tree is an *on-line* structure, while the a3DRB-tree is *off-line*, meaning that the lifespans of its entries should be known before the structure is created; otherwise, we have to store unbounded boxes inside the 3DR-tree, which affects query performance severely.

The experimental evaluation of [9] for static spatial dimensions suggests that the cube implementation is unsuitable in practice due to extreme query cost. aRB-trees consume a fraction of the space required by a3DR-trees, while they outperform them in all cases except for very short query intervals. Furthermore, unlike a3DR-trees where all the data must be known a priori, aRB-trees are on-line structures. For dynamic dimensions, the a3DRB-tree has the best overall performance in terms of size and query cost. Since however, it is an off-line structure, aHRB-trees are the best alternative for applications requiring on-line indexing.

[3]To be specific, the B-trees should be visited only if node E remains alive after timestamp 5. Otherwise, the aggregate values of R'_1 and R_2 for timestamp 5 are stored in E.

[4]The 3DR-tree structure of a3DRB-trees is similar to the a3DR-tree, but now each version is generated by an extent (rather than aggregate) change. Thus, there is no redundancy since the storage of MBRs is required to capture the new extent.

5 Conclusions

Numerous real-life applications require fast access to summarized spatio-temporal information. Although data warehouses have been successfully employed in similar problems for relational data, traditional techniques have three basic impediments when applied directly in spatio-temporal applications: (i) no support for ad-hoc hierarchies, unknown at the design time (ii) lack of spatio-temporal indexing methods, and (iii) limited provision for dimension versioning and volatile regions.

Here, we provide a unified solution to these problems by developing spatio-temporal structures that integrate indexing with the pre-aggregation technique. The intuition is that, by keeping summarized information inside the index, aggregation queries with arbitrary groupings can be answered by the intermediate nodes, thus saving accesses to detailed data. We first consider static dimensions and describe the basic structure (aRB-tree). Subsequently, we present a generalization of aRB-trees, which supports dynamic dimensions (aHRB-tree). For the same case, we also develop a solution based on a 3-dimensional modeling of the problem (a3DRB-tree). Our approach does not aim at simply indexing, but rather replacing the data cube for spatio-temporal data warehouses.

We believe that spatio-temporal OLAP is a new and very promising area, both from the theoretical and practical point of view. Since this is an initial approach, we limited this work to simple numerical aggregations. In the future, we will focus on supporting spatio-temporal "measures" like the direction of movement. This will enable analysts to ask sophisticated queries in order to identify interesting numerical and spatial/temporal trends. The processing of such queries against the raw data is currently impractical considering the huge amounts of information involved in most spatio-temporal applications.

Another interesting area concerns the extension of the proposed techniques to different access methods. For instance, we could apply the R-tree insertion algorithms of [2] in order to obtain on-line structures based on 3DR-trees. Furthermore, the integration of multi-version data structures may provide on-line methods more efficient than aHRB-trees. The problem with such methods (and all methods maintaining multiple R-trees) is the avoidance of multiple visits to the same node via different ancestors. Although various techniques have been proposed in the context of spatio-temporal data structures, it is not clear how they can be applied within our framework.

References

[1] Becker, B., Gschwind, S., Ohler, T., Seeger, B., Widmayer, P. An Asymptotically Optimal Multi-version B-Tree. *VLDB Journal*, 5(4): 264-275, 1996.

[2] Bliujute, R., Jensen, C., Saltenis, S., Slivinskas, G. R-Tree Based Indexing of Now-Relative Bitemporal Data. *VLDB*, 1998.

[3] Beckmann, N., Kriegel, H., Schneider, R., Seeger, B. The R*-tree: an Efficient and Robust Access Method for Points and Rectangles. *SIGMOD Conference*, 1990.

[4] Guttman, A. R-trees: A Dynamic Index Structure for Spatial Searching. *SIGMOD Conference*, 1984.

[5] Gray, J., Bosworth, A., Layman, A., Pirahesh, H. Data Cube: a Relational Aggregation Operator Generalizing Group-by, Cross-tabs and Subtotals. *ICDE*, 1996.

[6] Kimball, R. The Data Warehouse Toolkit. John Wiley, 1996.

[7] Nascimento, M., Silva, J. Towards Historical R-trees. *ACM SAC*, 1998.

[8] Papadias, D., Kalnis, P., Zhang, J., Tao, Y. Efficient OLAP Operations in Spatial Data Warehouses. *SSTD*, 2001.

[9] Papadias, D., Tao, Y., Kalnis, P., Zhang, J. Indexing Spatio-Temporal Data Warehouses. *ICDE*, 2002.

[10] Tao, Y., Papadias, D., Zhang, J. Aggregate Processing of Planar Points. *EDBT*, 2002.