Indexing Spatio-Temporal Data Warehouses

Dimitris Papadias, Yufei Tao, Panos Kalnis, and Jun Zhang
Department of Computer Science
Hong Kong University of Science and Technology
Clear Water Bay, Hong Kong
{dimitris, taoyf, kalnis, zhangjun}@cs.ust.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

Spatio-temporal databases have received considerable attention during the past few years due to the accumulation of large amounts of multi-dimensional data evolving in time, and the emergence of novel applications such as traffic supervision, and mobile communication systems. Research has focused on modeling, indexing and query processing issues for problems involving historical information retrieval [GBE$^+$00], motion and trajectory preservation [PJT00], future location estimation [SJLL00] etc. All these approaches assume that object locations are individually stored, and queries ask for objects that satisfy some spatio-temporal condition (e.g., mobile users inside a query window during a time interval, or the first car expected to arrive at a destination etc.).

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 objects 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 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 deal with static spatial dimensions and focus on queries that ask for 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. As a second step we extend our techniques to handle volatile regions; alternatives are proposed and their performance is evaluated experimentally. Our approach does not aim at simply indexing, but rather replacing the data cube for spatio-temporal data warehouses.

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 related work in the context of spatio-temporal access methods, traditional, temporal and spatial OLAP. Section 3 proposes indexing techniques for spatio-temporal OLAP applicable in the presence of static regions. Section 4 discusses structures to capture volatile regions. Section 5 contains an extensive experimental evaluation and section 6 concludes the paper with a discussion on future work.

2. Related Work

This section overviews background on access methods employed in the rest of the paper. It also describes previous work on the broad concept of storing additional information in index structures in order to accelerate aggregation queries. Furthermore, we discuss traditional OLAP techniques and their extensions to spatial and temporal data.

2.1 Spatio-temporal indexing – Aggregate trees

Numerous indexes have been proposed for indexing spatial and temporal databases. Among spatial access methods, the most popular one is the R-tree [G84] and its variations, notably the R*-tree [BKSS90]. The R-tree can be thought of as an extension of B^+-trees in multi-dimensional space. Figure 2.1 shows four regions and the corresponding R-tree assuming node capacity of 2. Regions R_1 and R_2 are grouped together in node R_5, and regions R_3 and R_4 in node R_6. Each R-tree entry r has the form <r.MBR, r.pointer> where r.MBR is the minimum bounding rectangle of r, and r.pointer points to the lower level node corresponding to r. For leaf entries r.pointer points to the actual record of the entry.

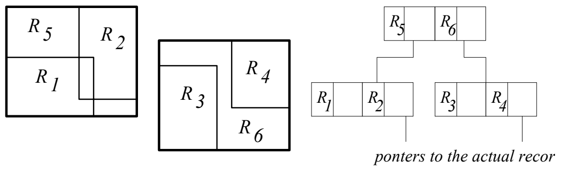


Figure 2.1: Simple R-tree example

R-trees were developed for static points and regions. Dynamic data can be indexed by multiple R-trees, each capturing a different version of the data. This, however, may lead to significant redundancy, if updates alter only a small part of the existing tree. *Historical R-trees* (HR-trees) [NS98] decrease the level of redundancy by allowing consecutive R-trees to share common branches. Assume that at timestamp 5, region R_1 is modified (due to movement, enlargement etc.) to a new version R'_1. This update propagates to the upper level of the tree, meaning that the extent of the father entry R_5 also changes to R'_5. The corresponding HR-tree structure is shown in Figure 2.2. The first logical R-tree covers timestamps 1-4, and the second one timestamp 5 and onwards. Both trees share the child node of R_6, because the contents of this node were not affected by the update.

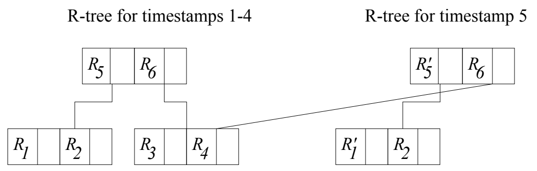


Figure 2.2: HR-tree example

Aggregate R-trees (aR-trees) [JL98, PKZT01] augment the tree nodes with summarized data about the sub-branch under them in order to answer aggregate queries. Consider, for instance, a dataset of points indexed by an R-tree. Each intermediate entry *r* in the R-tree, in addition to *r*.MBR and *r*.pointer, stores information about the value of an aggregate function (e.g., the number of points indexed by the node). The same concept is applied in [LM01] for a variety of high-dimensional indexes in order to calculate approximate answers. By traversing the index, a rough approximation is obtained from the values at the higher levels, which is progressively refined as the search continues towards the leaves.

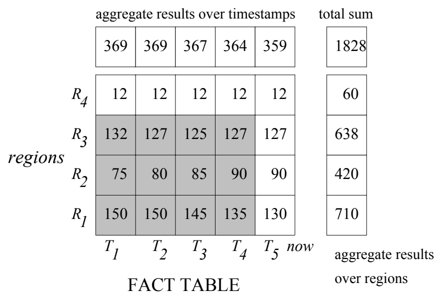
Similar structures have also been developed in the context of main memory temporal databases: Kline and Snodgrass [KS95] propose the *aggregation tree*, for computing aggregations over temporal data. The tree indexes *constant intervals*, i.e. the maximum continuous intervals where the value of the aggregation function is constant. The nodes store the difference of the aggregation value between the current and the next tree level; therefore, the aggregation function is computed by accumulating all the values in the path from the root to the leaves. The tree is built on-the-fly and the cost for computing the aggregation function for every constant interval is $O(n^2)$, where *n* is the number of tuples in the base relation. A similar structure, called *PA-tree* [KKK99], solves the problem in *O(nlogn)*. *SB-trees* [YW01] extend these techniques for secondary memory.

Our indexing methods are also based on aggregate trees. However, unlike previous approaches that focus on single spatial or temporal indexes, spatio-temporal data warehouses require some elaborate integration of spatial and temporal structures. The incorporation of volatile regions further complicates the indexing problem and the associated query mechanisms.

2.2 Data warehouses and OLAP

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 2.3 illustrates a simple case; observe that although the set of 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 [K96] 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, …, D_n store the details of the dimensions. A tuple in *F* has the form $<D_i[].key, M[]>$ where $D_i[].key$ is the set of foreign keys to the dimension tables and *M[]* is the set of measures.



		aggregate results over timestamps					total sum
		369	369	367	364	359	1828
	R_4	12	12	12	12	12	60
regions	R_3	132	127	125	127	127	638
	R_2	75	80	85	90	90	420
	R_1	150	150	145	135	130	710
		T_1	T_2	T_3	T_4	T_5 now	aggregate results over regions

FACT TABLE

Figure 2.3: 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 2.3 depicts these groupings assuming that the aggregation function is *sum*. The fact table, together with all possible combinations of group-bys, compose the *data cube* [GBLP96]. 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*.

Notice that a detailed group-by query can be used to answer more abstract aggregations. In our example, the total number of objects in all regions for all timestamps (i.e. 1828) can be computed either from the fact table, or by summing the projected results on the *time* or the *region* axis. If one of these projections is materialized, the cost of computing the total sum is lower, since less data are accessed. Ideally the whole data cube should be materialized, to enable efficient query processing. However, there are $O(2^n)$ group-by combinations for a data warehouse with *n* dimensional attributes and materializing all possible results may be prohibitive in practice. Therefore, several techniques have been proposed for the view selection problem in OLAP applications [SDN98, GM99]. Observe that all these methods benefit only queries on a predefined hierarchy. A query involving the shaded part of the data in Figure 2.3 would still need to access the fact table, even if the entire data cube were materialized.

Han et al [HSK98, SHK00] extend the above techniques to spatial data warehouses. They consider a generalization of the star-schema in which the cube dimensions can be both spatial and non-spatial and the measures can be regions in the space, in addition to numerical data. They focus on the spatial measures and propose a method for selecting spatial objects for materialization. The idea is similar to the algorithm of [HRU96], the main difference being the finer granularity of the selected objects.

Mendelzon and Vaisman [MV00] concentrate on temporal data warehouses, where, unlike the traditional approach, the dimensions are allowed to change over time (i.e. *schema versioning* or *schema evolution*). They propose a temporal multidimensional model and a query language to support it, while in [HMV99] they present an incremental update method for data cubes under dimension updates. Our structure also supports schema evolution by allowing the finest granularity objects of the spatial dimension to change over time.

In the following sections, we will present indexing techniques for spatio-temporal data warehouses, which accelerate various types of aggregation queries (e.g., such as the one that corresponds to the shaded part of Figure 2.3) without assuming pre-defined hierarchies. To the best of our knowledge, there is no published work on supporting spatio-temporal OLAP operations.

3. Indexing Static Spatial Dimensions

We first consider fixed spatial dimensions (e.g., a static set of road segments) and the goal is to keep historical aggregated information, i.e., statistical results about each region during the time evolution. For simplicity, in the following discussion we assume that there exists only one measure (e.g., number of objects) although additional measures (any non-holistic function) can be easily incorporated. The typical query has the form: "find the total number of objects in the regions intersecting some window q_s during a time interval q_t", where the total number is interpreted as the sum of objects in each timestamp. Notice that the interpretation of the query is not important, e.g., by dividing the total number with the interval length we could get the average number of objects per timestamp in the region.

3.1 Applications of existing techniques

Following a traditional OLAP approach we could create a data cube, where one dimension corresponds to time, the other to space, and keep the measure values in the cells of this two-dimensional table (as in Figure 2.3). 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 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*.

Even if there exists an additional spatial index on the regions, the simultaneous employment of both indexes will not provide significant benefit. Assume that first a window query (q_s) is performed on the spatial index to provide a set of ids for regions that qualify the spatial condition. Aggregate information about these regions, still needs to be retrieved from the columns corresponding to q_t (again, we find these columns through the B-tree index). However, since the storage method does not preserve spatial proximity, the data from the spatially qualifying regions is expected to be scattered in many pages, and the spatial index will only have some effect on queries with very high spatial selectivity. Furthermore, pre-materializing selected results is essentially meaningless for general queries, since the query windows q_s and q_t are usually ad-hoc and do not correspond to well-defined hierarchies.

An alternative approach, which achieves simultaneous indexing on both spatial and temporal dimensions, can be obtained by the generalization of the aR-tree (discussed in Section 2.1) to 3-dimensional space. In particular, each entry of the *aggregate 3DR-tree* (a3DR-tree) has the form <*r*.MBR, *r*.pointer, *r*.lifespan, *r*.aggr[]>, 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 2.3, 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 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. Next, we present a novel multi-tree structure which does not suffer from these problems.

3.2 The aggregate RB-tree

The *aggregate R- B-tree* (aRB-tree) 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 <*r*.MBR, *r*.pointer, *r*.btree, *r*.aggr[]> where *MBR* and *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 <*b*.time, *b*.pointer, *b*.aggr[]> 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 3.1 illustrates the aRB-tree for the regions of Figure 2.1 using the data of the cube in Figure 2.3. For instance, the number 1130 stored with the R-tree entry R_5, denotes that the total number of objects in R_5 is 1130. The first leaf entry of the B-tree for R_5 (1, 225) denotes that the number of objects in R_5 at timestamp 1 is 225. Similarly the first entry of the top node (1, 685) denotes that the number of objects during the interval [1,3] is 685. 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 2.3, i.e., answer queries involving only temporal conditions.

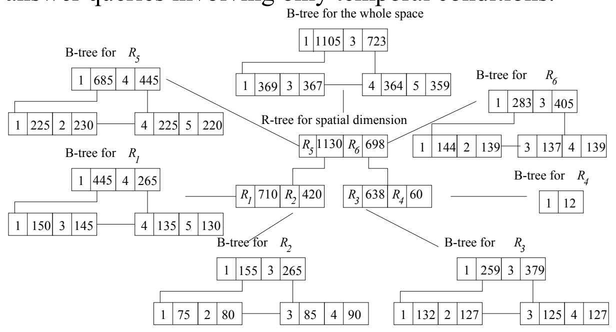


Figure 3.1: Example of aRB-tree

The aRB-tree facilitates the processing of aggregate 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 3.2 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. This corresponds to the sum of aggregate data in the gray cells of Figure 2.3.

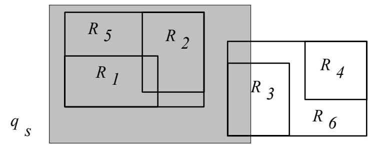


Figure 3.2: Query example

aRB-trees can also answer group-by queries, as, for instance, "for every district of the city, find the average traffic during peak hours". The grouping attribute (i.e. district), defines a set of query windows, for which we want the aggregated value. The union of all grouping windows does not necessarily cover all the space, and the windows may intersect with each other. A grouping aggregation query can be processed by employing the previous algorithm for every query window (e.g., indexed nested loops). In this case, the nodes of the aRB-tree that overlap multiple windows will be accessed more than once. A more efficient approach is to extend spatial join algorithms that join an R-tree-indexed data set with a non-indexed one (i.e., the query windows).

In summary, the aRB-tree is more than just an index, since it replaces the data cube. 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 (discussed in section 2.1). For example in Figure 4.1, the two R-trees share node C, because the extents of regions R_3 and R_4 did not change. Each node[1] 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.

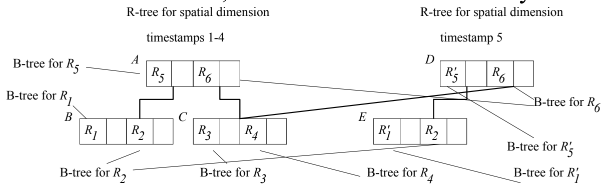


Figure 4.1: Example of aHRB-tree

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 4.2 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 4.1 are visited. In the first tree, since R_5 is inside 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[2].

[1] Notice that this is different for a3DR-trees where lifespans are kept for each entry. 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.

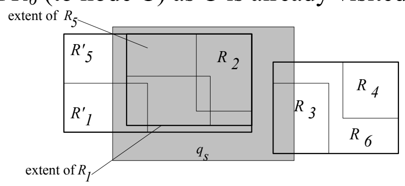


Figure 4.2: Query example for aHRB-trees

In order to avoid multiple visits to the same R-tree node via different roots, we use positive and negative pointers to distinguish exclusive and shared nodes. In Figure 4.1, for instance, when the root D of the new tree is created, its pointers to nodes B and C are negative. Then, as new nodes are created in the current tree (e.g., E), the negative pointers (e.g., to B) are replaced with positive ones (to E), while pointers to shared nodes (e.g., to C) remain negative. A query can now be answered as follows: the R-tree associated with the earliest timestamp is searched first and all (positive and negative) qualifying pointers are followed. Next the trees associated with the other timestamps are searched in chronological order by following only positive pointers.

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 4.1 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 in section 4.3.

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 <r.MBR, r.lifespan, r.pointer, r.btree, r.aggr[]>, where r.MBR, r.pointer, r.btree are defined as in aRB-trees; r.aggr[] stores data over r.lifespan[3].

[2] 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.

[3] 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.

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. Searching starts from the root of the 3DR-tree; for each entry r one of the following conditions may hold:

1. The entry is covered by both (q_s and q_t) query extents. In this case all required aggregate data are stored in *r*.aggr[].
2. The entry's spatial extent is covered by q_s, and its temporal extent partially overlaps q_t. The B-tree pointed by *r*.btree is accessed to retrieve aggregate information for q_t.
3. The entry's spatial extent partially overlaps q_s, and its temporal extent overlaps (or is inside) q_t. In this case *r*.pointer is followed to the next R-tree level and the same process is applied recursively.
4. Otherwise, the node pointed by the entry and its B-tree are not visited.

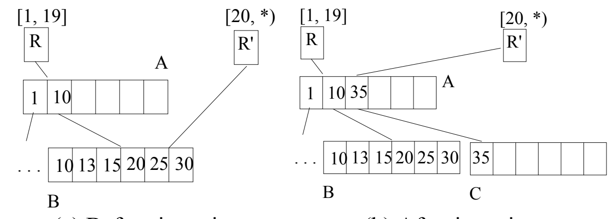
Although both aHRB- and a3DRB- trees are aimed at volatile regions they have two important differences: (i) a3DRB-trees maintain a large 3DR-tree for the whole history, while aHRB-trees maintain several small trees, each responsible for a relatively short interval. This fact has implications on their 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.

aHRB- and a3DRB- trees permit temporally disjoint entries to share B-trees in order to save space. Next we discuss the B-file, a B-tree-based organization that permits this sharing without compromising query performance.

4.3 Management of B-trees

In real life situations, the rate of changes may differ significantly for different objects. For instance, some objects may update their extents so often, that there exist very few aggregate values for each version. Keeping a separate B-tree for every version of such objects would seriously compromise space utilization. This problem led to the development of the *B-File* (BF), which is a space-efficient storage scheme for multiple B-trees. A BF possesses the following properties: (i) the B-trees stored in a BF index disjoint sets of keys (i.e., there can be at most one alive B-tree at any timestamp) (ii) since deletions never happen, all the nodes (except, possibly, for the last node of each level) are full, and (iv) the search algorithms and their cost are the same as those for conventional B-trees.

Figure 4.5a illustrates an example BF, which stores the B-trees of two regions R and R' (for simplicity, in each entry we show only the timestamps and not the aggregate values). The lifespan of R is [1,19], while that of R' is [20,*) (R' is currently alive). The B-tree of R consists of two levels while, up to timestamp 30, the B-tree of R' has only one level. Note that the root pointers of R and R' point to nodes at different levels. The insertion of 35 (in the B-tree of R') causes node B to overflow, and a new node C is created (Figure 4.5b). An entry 35, pointing to node C, is inserted into A, which becomes the root of the B-tree of R'.



(a) Before insertion (b) After insertion

Figure 4.5: A B-file example

If the live B-tree dies (e.g., R' ceases to exist), the corresponding BF becomes *vacant* and may be used for any B-tree created at later timestamps. Whenever a new B-tree needs to be created, we first search for vacant BFs. If such a BF does not exist, a new one is initiated. In practice, the creation of new BFs is infrequent because, when an object changes its position or extent, the new entry (in the HR- or 3DR-trees) can use the vacant BF of the previous version. Due to space constraints we will not describe in detail the insertion algorithms, but it is easy to verify that query processing with the BF is as efficient as independent B-trees. On the other hand, the BF can achieve significant space savings, especially for highly dynamic datasets.

5. Experimental Evaluation

In this section, we evaluate the performance of the proposed methods through experimental evaluation. Each dataset contains aggregate data of 10,000 uniformly distributed regions (density 0.2) collected over (unless specifically stated) 1,000 timestamps. At each timestamp, the aggregate data of A_{agi} of the regions is modified, where A_{agi} is a dataset parameter called the *aggregate agility*. Selected values for A_{agi} are 4%, 8%, 16%, 32%, and 64%. For example, A_{agi} = 4% denotes that 400 regions update their aggregate data per timestamp. The region agility R_{agi} corresponds to the percentage of regions that issue position changes per timestamp. For datasets of static regions R_{agi} = 0%, while for volatile regions, R_{agi} is fixed to 0.01% (to capture the fact that the spatial dimension changes much slower than the aggregate data). With these parameters, the number of records in a dataset ranges from 500,000 to 6,500,000.

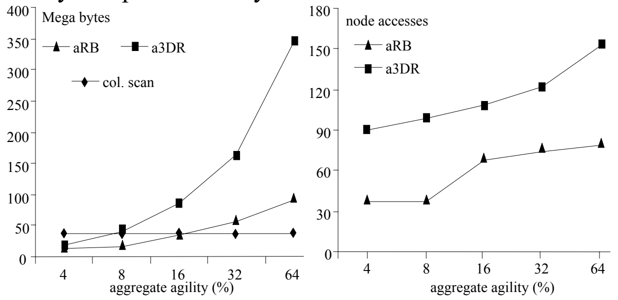
Performance of various methods is measured by the number of nodes accessed during the processing of *workloads*, each consisting of 500 queries. Every query involves two parameters that affect performance: (i) the side of the query window (q_s), represented as the percentage of its length over the spatial universe, and (ii) interval length (q_t). These parameters have identical values for all queries in the same workload. The selected values for q_s and q_t are 1%, 3%, 5%, 7%, 9% of the universe and 1, 25, 50, 75, 100 timestamps (up to 10% of the history length) respectively. In the sequel, we refer to a workload as $WRKLDq_s,q_t$ to indicate its parameters. Queries are generated so that (i) the position of each query window is distributed uniformly in the spatial universe; (ii) each interval distributes uniformly in [1,1000]. The node size is fixed to 1024 bytes in all cases. With this size, the fanouts of aggregate R-trees, B-trees, HR-trees are 36, 82, 36, and 36 respectively. The fanouts of the 3DR-trees used in a3DR-trees and a3DRB-trees are 31 and 28 respectively. R-tree implementations are based on R*-trees [BKSS90], and B-tree implementations on B+-trees.

5.1 Static regions

For static regions, we compare the performance of aRB-trees to the two applications of existing techniques described in section 3.1, namely, the a3DR-tree and data cube (column scanning). In addition to the 2D array with aggregate data, the data cube approach maintains a 1D array for region extents. Each column of the 2D array consists of 40 blocks, while the 1D array contains 160 blocks. Columns are linked through a doubly linked list to support sequential scans and are indexed by a B-tree so that the first entry of each column (i.e., timestamp) can be easily located. Notice that the aggregate information of each region should be stored for each timestamp even if it does not change. There is no obvious way to avoid replicating aggregate information, since storage is according to columns in order to preserve temporal proximity.

For the first experiment we employ static datasets with aggregate agilities ranging from 4% to 64%. Figure 5.1(a) shows the space requirements of the three methods, as a function of A_{agi}. For low agilities, both a3DR-tree and aRB-tree consume less space than the cube because they avoid the replication of aggregate data that do not change. On the other hand, since the size of the cube is constant, above A_{agi} =16% it becomes the most economical method. The a3DR-tree increases linearly with A_{agi} and for high values it involves huge space requirements. The behavior of aRB-tree is much better, since the increase of A_{agi} affects only the sizes of the B-trees where entries are much smaller. As predicted, even for A_{agi} =64%, the aRB-tree requires less than twice the size of the data cube.

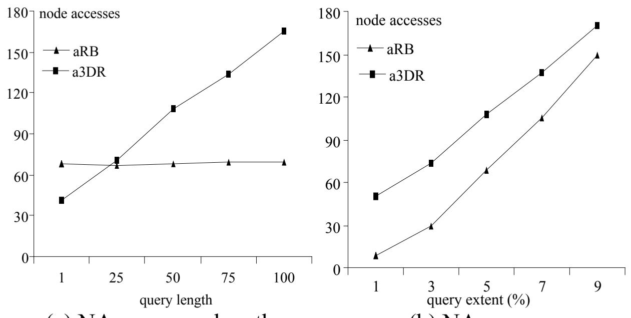
Figure 5.1b shows the average number of node accesses (NA) in answering workloads $WRKLD_{5\%,50}$ as a function of A_{agi}. The aRB-tree outperforms the a3DR-tree significantly. Notice that, the cost of aRB-trees remains stable up to A_{agi} = 8%, at which point there is a step in the query cost. This is caused by the increment of the height of B-trees, as each of them indexes more entries when the agility increases. The performance of column scanning is omitted because its cost is prohibitively high (more than 2,000 node accesses for scanning 50 columns). This is expected because the spatial condition is not taken into account during query processing. Therefore, in the sequel we omit the naïve cube implementation from our experiments since it is always outperformed by the other methods.



(a) Size vs. aggregate agility (b) NA vs. aggregate agility

Figure 5.1: Size and performance comparisons

The next set of experiments compares aRB-trees and a3DR-trees as a function of the query (spatial) extent and length. In Figure 5.2a, we fix q_s to 5% of the spatial axis and vary q_t from 1 to 100 timestamps; in Figure 5.2b, we fix q_t to 50 timestamps (i.e., 5% of the total history), and vary q_s from 1% to 9% of the axis. In both cases the aggregate agility is fixed to 16%. According to the first diagram, the performance of aRB-trees does not depend on the query length. This is not surprising because the number of B-trees accessed remains constant; furthermore, long intervals have a high chance to be answered by the top B-tree nodes. On the other hand, the cost of a3DR-trees grows linearly with the query length because the probability that the aggregate information of a node changes also increases. Thus, several versions of the same node (each with different aggregate values) may be retrieved. For short query intervals, the probability of retrieving different versions of the same node is small and a3DR- outperform aRB-trees because they avoid accessing the B-trees. In Figure 5.2b, a3DRB-trees are better in all cases of window extents. The difference is reduced for large query windows, because as the window increases, so does the number of B-trees that need to be accessed.



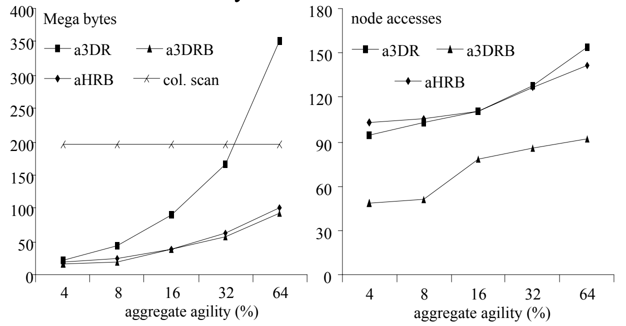
(a) NA vs. query length (b) NA vs. q_s

Figure 5.2: Effect of query parameters

In summary, although the cube implementation is the most space-efficient alternative for very high agilities, it 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.

5.2 Volatile regions

The methods applicable to volatile regions are column scanning, a3DR-, aHRB-, and a3DRB-trees. Figure 5.3a shows the sizes of the methods as a function of A_{agi} assuming region agility 0.01%. The cube approach requires more space than the static case, because the extents of the regions are now stored in a 2D array since they are no longer fixed. The a3DR-tree is still very space-consuming. The sizes of the a3DRB- and aHRB-tree are almost identical because the sizes of the B-Files in the two structures (the dominant factor in the overall size), are similar. The a3DRB-tree is slightly smaller due to the redundancy in HR-trees.

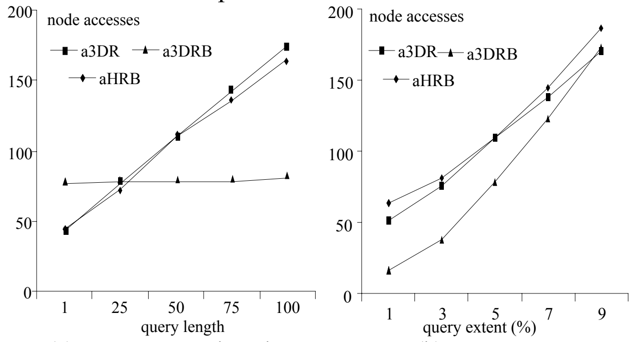


(a) Size vs. aggregate agility (b) NA vs. aggregate agility

Figure 5.3: Size and performance comparisons

Figure 5.3b compares the structures on query performance using $WRKLD_{5\%,50}$. The column scanning approach is again omitted due to its prohibitive cost. The relative performance of a3DRB- and a3DR-trees is similar to the case of static regions. The cost of aHRB-trees is comparable to that of a3DR-trees. Their high cost (compared to a3DRB-trees) is explained as follows. Since the region agility is 0.01% and there exist 10,000 objects in the dataset, on the average one object issues an extent change per timestamp. Thus, a new logical tree is created each timestamp. Although consecutive trees share a large number of nodes, a query in $WRKLD_{5\%,50}$ has to access at least the 50 roots that correspond to its q_t (duplicate visits to lower level nodes are avoided). Notice that the difference between aHRB- and a3DRB-trees is about 50 NA.

To study the effects of query parameters, we adopt the same approach as in the static case using the dataset with A_{agi}=16% and R_{agi}=0.01%. In Figure 5.4a, q_s is fixed to 5% and the query length varies. Like aRB-trees (and for the same reasons), the a3DRB-tree is not sensitive to the query length, while the costs of the other two structures grow linearly. The behavior of a3DR-trees is similar to the static case. aHRB-trees are also favored by short queries, because the number of logical trees that need to be accessed is small. In Figure 5.4b, q_t is fixed to the standard value 5% of the history length while q_s is varied. Similar to aRB-trees, the a3DRB-tree is more efficient than the a3DR-tree, but the difference decreases with the window size. The performance of aHRB-trees is comparable to that of a3DR-trees.



(a) NA vs. query length (b) NA vs. q_s

Figure 5.4: Effect of query parameters

In general, 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.

6. Conclusion

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). We demonstrate the applicability of our methods through a set of experiments that attempt to simulate realistic situations.

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 [BJSS98] 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 (e.g., [vBS96]), it is not clear how they can be applied within our framework.

Acknowledgments

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

References

[BJSS98] Bliujute, R., Jensen, C., Saltenis, S., Slivinskas, G. R-Tree Based Indexing of Now-Relative Bitemporal Data. *VLDB*, 1998.

[BKSS90] Beckmann, N., Kriegel, H., Schneider, R., Seeger, B. The R*-tree: an Efficient and Robust Access Method for Points and Rectangles. *SIGMOD*, 1990.

[G84] Guttman, A. R-trees: A Dynamic Index Structure for Spatial Searching. *SIGMOD*, 1984.

[GBE$^+$00] Güting, R., Böhlen, M., Erwig, M., Jensen, C., Lorentzos, N., Schneider, M., Vazirgiannis, M. A Foundation for Representing and Querying Moving Objects. To appear in *ACM TODS*.

[GBLP96] Gray, J., Bosworth, A., Layman, A., Pirahesh, H. Data Cube: a Relational Aggregation Operator Generalizing Group-by, Cross-tabs and Subtotals. *ICDE*, 1996.

[GM99] Gupta, H., Mumick, I. Selection of Views to Materialize Under a Maintenance-Time Constraint. *ICDT*, 1999.

[HRU96] Harinarayan, V., Rajaraman A., Ullman, J. Implementing Data Cubes Efficiently. *SIGMOD*, 1996.

[HSK98] Han, J., Stefanovic, N., Koperski, K. Selective Materialization: An Efficient Method for Spatial Data Cube Construction. *PAKDD*, 1998.

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

[K96] Kimball, R. *The Data Warehouse Toolkit*. John Wiley, 1996.

[KKK99] Kim, J., Kang, S., Kim, M. Effective Temporal Aggregation using Point-based Trees. *DEXA*, 1999.

[KS95] Kline, N., Snodgrass, R. Computing Temporal Aggregates. *ICDE*, 1995.

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

[MV00] Mendelzon, A., Vaisman, A. Temporal Queries in OLAP. *VLDB*, 2000.

[NS98] Nascimento, M., Silva, J. Towards Historical R-trees. *ACM SAC*, 1998.

[PJT00] Pfoser, D., Jensen, C., Theodoridis, Y. Novel Approaches to the Indexing of Moving Object Trajectories. *VLDB*, 2000.

[PKZT01] Papadias, D., Kalnis, P., Zhang, J., Tao, Y. Efficient OLAP Operations in Spatial Data Warehouses. *SSTD*, 2001.

[SDN98] Shukla, A., Deshpande, P., Naughton, J. Materialized View Selection for Multidimensional Datasets. *VLDB*, 1998.

[SHK00] Stefanovic, N., Han, J., Koperski, K. Object-Based Selective Materialization for Efficient Implementation of Spatial Data Cubes. *TKDE*, 12(6), 2000.

[SJLL00] Saltenis, S., Jensen, C., Leutenegger, S., Lopez, M. Indexing the Positions of Continuously Moving Objects. *SIGMOD*, 2000.

[vBS96] Van den Bercken, Seeger, B. Query Processing Techniques for Multiversion Access Methods. *VLDB* 1996.

[YW01] Yang, J., Widom, J. Incremental Computation of Temporal Aggregates. *ICDE*, 2001.