Techniques for Indexing and Querying Temporal Observations for a Collection of Objects*

(CS-TR-4503 & UMIACS-TR-2003-72)

Qingmin Shi and Joseph JaJa
Institute for Advanced Computer Studies
Department of Electrical and Computer Engineering
University of Maryland, College Park, MD 20742, USA
{*qshi,joseph@umiacs.umd.edu*}

Abstract

We consider the problem of dynamically indexing temporal observations about a collection of objects, each observation consisting of a key identifying the object, a list of attribute values and a timestamp indicating the time at which these values were recorded. We make no assumptions about the rates at which these observations are collected, nor do we assume that the various objects have about the same number of observations. We develop indexing structures that are almost linear in the total number of observations available at any given time instant, and that support dynamic insertions in polylogarithmic time. Moreover, these structures allow the quick handling of queries to identify objects whose attribute values fall within a certain range at every time instance of a specified time interval. Provably good bounds are established.

1 Introduction

Consider the scenario in which temporal observations about a large collection of objects are being collected asynchronously. Each observation record consists of a key identifying the object, a list of the values of a number of attributes, and a timestamp indicating the time at which these particular values were recorded. We make no assumptions about the rates at which these observations are collected, nor do we assume that various objects will have the same number of observations. In fact, we even allow the collection of objects to vary with time, with possibly new objects inserted into our collection. The only assumption we make is that the timestamp of a new observation record for a given object has to be larger than the timestamp of the object's observations that are already stored in our data structure. This assumption seems to be quite natural for the types of applications we have in mind.

*Supported in part by the National Science Foundation through the National Partnership for Advanced Computational Infrastructure (NPACI), DoD-MD Procurement under contract MDA90402C0428, and NASA under the ESIP Program NCC5300.

We are interested in storing these observations into an indexing structure that will enable the quick discovery of temporal patterns. For our purposes, the patterns of interest can be defined as the values of certain attributes remaining within certain bounds, changing according to a given pattern, or satisfying certain statistical distributions, for every observation with a timestamp falling within a given time interval. We will focus in this paper on detecting the objects whose temporal patterns are characterized by a set of value ranges. More specifically, we want to dynamically maintain an indexing structure so as to quickly identify objects whose attributes consistently fall within a set of ranges during a given time period.

We next introduce the problem more formally and give a few possible applications that will fit under this framework.

1.1 Problem Definition

Consider a set S of n objects $\{O_1, O_2, \ldots, O_n\}$, each identified by a key o_i and characterized by a set of d attributes $\{v_{i,1}(t), v_{i,2}(t), \ldots, v_{i,d}(t)\}$ whose values change over time t. Observations about each object are collected at discrete time instances. Let m_i be the number of observations about object O_i, say collected at time instances $t_i^1 < t_i^2 < \cdots < t_i^{m_i}$. We denote the observations of O_i at t_i^j as a vector $\mathbf{v}_i^j = [v_{i,1}^j, v_{i,2}^j, \ldots, v_{i,d}^j]$, where $v_{i,l}^j = v_{i,l}(t_i^j)$ for $l = 1, \ldots, d$. The total number of observations for all the objects in S is $m = \sum_{i=1,\ldots,n} m_i$. We denote the number of distinct time instances among $\{t_i^j | 1 \le i \le n, 1 \le j \le m_i\}$ as m'. Note that $m' \le m$.

We are interested in developing dynamic data structures to index all the observations so that the following type of queries, called *temporal range queries*, can be handled quickly:

> Given two vectors $\mathbf{a} = [a_1, a_2, \ldots, a_d]$ and $\mathbf{b} = [b_1, b_2, \ldots, b_d]$, and a time interval $[t_\mathrm{s}, t_\mathrm{e}]$, determine the set Q of objects such that $O_i \in Q$ if and only if the following two conditions are satisfied:
>
> - $\exists j$ such that $t_i^j \in \{t_i^l | l = 1, \ldots, m_i\}$ and $t_\mathrm{s} \le t_i^j \le t_\mathrm{e}$, i.e., there is at least one observation of O_i recorded between t_s and t_e.
> - $\forall j$ such that $t_i^j \in \{t_i^l | l = 1, \ldots, m_i\}$ and $t_\mathrm{s} \le t_i^j \le t_\mathrm{e}$, $a_k \le v_{i,k}^j \le b_k$ for all $1 \le k \le d$.

We will call each such object a *proper* object with respect to the query.

We allow new observations to be incorporated into the existing data structure. Whenever it happens, we assume that the timestamp associated with a new observation is larger than that of any previous observation of the same object. However, we allow the timestamp of a new observation to be smaller than the timestamps of observations related to other objects.

We note that the condition that at least one observation exists in the time interval $[t_\mathrm{s}, t_\mathrm{e}]$ can be relaxed as follows. We construct a set of intervals $\{(t_i^j, t_i^{j+1}) | j = 0, \ldots, m_i\}$ for each object O_i, where $t_i^0 = -\infty$ and $t_i^{m_i+1} = +\infty$. Reporting objects with no observation in $[t_\mathrm{s}, t_\mathrm{e}]$ is equivalent to reporting intervals that contain $[t_\mathrm{s}, t_\mathrm{e}]$, which can easily be performed using a priority search tree [17] in $O(\log n + f)$ time (note that only one interval will be reported for each such object). The priority search tree can handle insertions of new observations as well. The complexity of maintaining the correct set of intervals in the priority search tree for each insertion of a new observation is $O(\log n)$ time.

The complexity of our algorithms will be measured by the storage cost of the data structure, the time spent on answering a temporal range query, and the time it takes to incorporate a new observation into our data structure. We will represent these costs as functions of n, m and d, where d is typically considered to be a constant.

The problem described here is more general than the one discussed in our previous paper [23], in which we require that the observations of these objects are collected in a *synchronized* fashion. That is, the observations of the objects are all collected at the same time instances. In addition, only the static case was addressed in that paper.

1.2 Sample Applications

Many applications seem to involve the general problem described above. A typical scenario consists of a large distributed network of sensors asynchronously collecting some type of measurements, and sending these measurements to a central location for storage, real-time access, and mining. Two examples are provided next.

- *Environmental monitoring.* A large number of sensors are distributed over specific geographic areas, each working independently and collecting measurements about various environmental factors (such as temperature, humidity, and wind speed, etc). These measurements, each coupled with a key identifying the sensor (and hence the geographic area) and a timestamp are sent to a central server. Users will query the central server to discover spatio-temporal environmental patterns based on the information collected thus far, and try to relate them to different physical phenomena.

- *Marine traffic control.* We have a number of vessels, each reporting its position (and possibly some other information) to a traffic center on a regular basis. The staff at the traffic center may want to identify the vessels whose trajectories lied in a certain region during a time interval.

1.3 Previous Related Work

A special case of our problem is the well-studied d-dimensional orthogonal range search problem. In spite of the existence of an extensive literature, only a limited number of special orthogonal range search problems admit linear space data structures with polylogarithmic query time solutions. These special problems include the three-sided 2-D range queries [17] and the 3-D dominance queries [6, 15]. Otherwise, all fast query algorithms require non-linear space, sometimes coupled with matching lower bounds under certain computational models [4, 5, 12]. Note that we cannot treat our problem as an orthogonal range search by simply treating the time snapshots as just an extra time dimension added to the d dimensions corresponding to the attributes. This is the case since the observations collected at difference time instances for the same object cannot not be treated as independent of each other. However another version of our problem, in which there exists some observation in $[t_s, t_e]$ which satisfies the required bounds, can be reduced to the so called *generalized intersection problem* addressed in [11, 10].

A related class of problems that have been studied in the literature, especially the database literature, deals with time series of data by appending a timestamp (or time interval) to each piece of data separately, thus treating each *record*, rather than each object, as an individual entity. As far as we can tell, none of these techniques seem to be suitable to address the general problem defined in this paper. Examples of such techniques include those based on *persistent* data structures [8], such as the Multiversion B-tree [14], and Multiversion Access Methods [26], and the Overlapping B^+-trees and its extensions [16, 18, 24, 25]. Even though these techniques work well for queries that involve only a single time instance, they do not capture temporal information about individual objects, nor do they seem to be able to efficiently handle long time intervals (the query time of these methods typically depends on the length of the time interval, which is undesirable for our general problem since the temporal range query could cover a very long time period characterized only by the two parameters t_s and t_e). See [22] for a recent survey about these techniques.

Some work does explicitly address queries that involve time intervals, especially in indexing moving objects. However, they all deal with the "or" queries, queries that report an object if its values fall in the query ranges at some time within the query interval, which is quite different from our problem. In the case where the objects are assumed to be moving along a straight line and at constant speed, which implies that the positions of the objects need not be explicitly stored, solutions with provable bounds exist (See for example [1, 13, 21]). In other cases, where the trajectories of objects are recorded as sequences of line segments, practical algorithms have only been proposed with no guaranteed bounds, such as in [3, 20].

We start by addressing the special case when there is only one attribute for each object. We deal in Section 2 with the static case, where the m observations about the different objects are given as input for preprocessing, and propose three solutions with different space-time trade-offs. In Section 3, we show how these solutions can be made dynamic so that new observations can be efficiently incorporated into the existing structures. We generalize these techniques in Section 4 for an arbitrary number of attributes whenever we have a predefined time hierarchy, and in Section 5 we briefly mention how to extend our techniques to handle queries that also involve key ranges.

2 One-Sided Temporal Range Queries: The Static Case

One of our goals in designing the indexing structure and query algorithm is to make sure that no proper object will be missed and the query complexity is proportional to the number of such objects. To achieve this goal, we first transform the query on objects to a query on identifying specific observations. Our approach is based on enhancing each observation with additional information such that for each proper object, exactly one of its observations will be reported.

2.1 Preliminaries

Let v_i^j denote the observation of object O_i at time instance t_i^j. Given a query represented by the triple (t_s, t_e, a), we aim at identifying the objects that have at least one observation during the time interval $[t_s, t_e]$ and whose observations within that time interval are all greater than or equal to a. We call this type of queries *one-sided temporal range queries*.

We will give three solutions to this problem, each providing a tradeoff between the storage cost and the query time. To develop these algorithms, we reformulate our problem to make use of a number of known techniques borrowed from computational geometry.

We start by making the following straightforward observation.

Observation 1. *An object O_i is proper with respect to the query (t_s, t_e, a) if and only if $\min\{v_i^j | t_s \le t_i^j \le t_e\} \ge a$.*

Note that we define $\min\{v_i^j | t_s \le t_i^j \le t_e\} = -\infty$ whenever no j exists such that $t_s \le t_i^j \le t_e$. We define the *dominant interval* $I_i^j = (s_i^j, e_i^j)$ of observation v_i^j as the longest time interval during which v_i^j is the smallest observation of O_i. More specifically, let $v_i^{j_1}$ be the latest observation such that $j_1 < j$ and $v_i^{j_1} \le v_i^j$ and $v_i^{j_2}$ be the earliest observation such that $j_2 > j$ and $v_i^{j_2} < v_i^j$. Then $s_i^j = j_1$ and $e_i^j = j_2$. If j_1 does not exist, then $s_i^j = -\infty$. Similarly, $e_i^j = +\infty$ if j_2 does not exist. Note that I_i^j is an open interval, meaning that it does not include the time instances s_i^j and e_i^j. We thus transform an observation v_i^j into a 5-tuple (or tuple for short) $(v_i^j, t_i^j, s_i^j, e_i^j, o_i)$. Figure 1 shows an object with eight observations, taken at time instances $1, 2, \ldots, 8$. For example, the dominant of interval of the 4th instance is $(-\infty, 6)$ and that of the 5th instance is $(4, 6)$. The following lemma shows that there exists a unique representative tuple for each proper object.

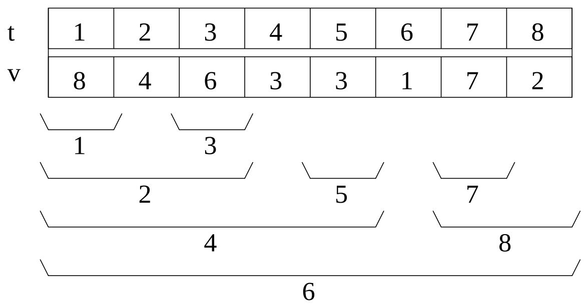

Figure 1: Dominant intervals for a time-series of observations corresponding to an object.

Lemma 1. *An object O_i is proper with respect to the query (t_s, t_e, a) if and only if there exists a unique tuple $(v_i^j, t_i^j, s_i^j, e_i^j, o_i)$ such that $(s_i^j, e_i^j) \supset [t_s, t_e]$, $t_i^j \in [t_s, t_e]$, and $v_i^j \ge a$.*

Proof. By definition an object O_i is proper if during the time interval $[t_s, t_e]$ no observation is smaller than a. Let $v_i^j = \min\{v_i^l | t_s \le t_i^l \le t_e\}$ (it always exists for a proper object), where j is the smallest such index if multiple minima exist. It is obvious that the tuple $(v_i^j, t_i^j, s_i^j, e_i^j, o_i)$ satisfies the three conditions stated in the lemma. On the other hand, if O_i is not proper, then either there is no observation of O_i in $[t_s, t_e]$, or the value of at least one such observation is less than a. In the latter case, no interval (s_i^l, e_i^l) with $t_i^l \in [t_s, t_e]$ and $v_i^l \ge a$ will be able to cover $[t_s, t_e]$. The uniqueness of this tuple is due to the fact that the dominant intervals are maximal. $\square$

Lemma 1 reduces the problem of determining the set of proper objects to finding for each such object one tuple that satisfies the three stated conditions. In the next sections, we show that such tuples can be efficiently identified using techniques from computational geometry.

2.2 An $O(m \log m)$-Space $O(\log n \log m + f)$-Time Solution

The indexing structures we propose in this and the next sections both follow the strategy of first singling out those tuples whose corresponding observations are collected during the time interval $[t_s, t_e]$ and then filtering them using the remaining two conditions. We call the data structure proposed in this section the *fast temporal range tree (FTR-tree)* because it is the fastest among the three solutions proposed; and the one discussed in Section 2.3, which uses less space but requires more query time, is called the *compact temporal range tree (CTR-tree)*.

Let $(t_1, t_2, \ldots, t_{m'})$ be the sorted list of all the distinct time instances. The skeleton of the FTR-tree is a balanced binary tree T built on this list. Each node u is associated with a set $S(u)$ of up to n tuples (n is the number of objects). If u is the kth leaf starting from the left, then $S(u) = \{(v_i^j, t_i^j, s_i^j, e_i^j, o_i) | t_i^j = t_k\}$. If u is an internal node with two children v and w, we decide for each object O_i which tuple to be added to $S(u)$ by examining the tuples corresponding to O_i in v and w. If $S(v)$ and $S(w)$ do not contain any such tuple, then no tuple for O_i will be added to $S(u)$. If only one of them do, then that tuple is included in $S(u)$. If both of them do, then the tuple with the longest dominant interval is chosen. Note that in this case the longer interval always contains the shorter one. Figure 2 illustrates how the tuples associated with each node are collected for an example consisting of two objects and a total of 16 observations. In this example, each node is associated with up to 2 tuples, the one above the horizontal line corresponds to object O_1 and the one below it corresponds to object O_2. We omit the values of t_i^j and o_i for each tuple.

t	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
O_1	8		4	6			3	5	1				7		2	
O_2		5			2	4				1	7	3		6		8

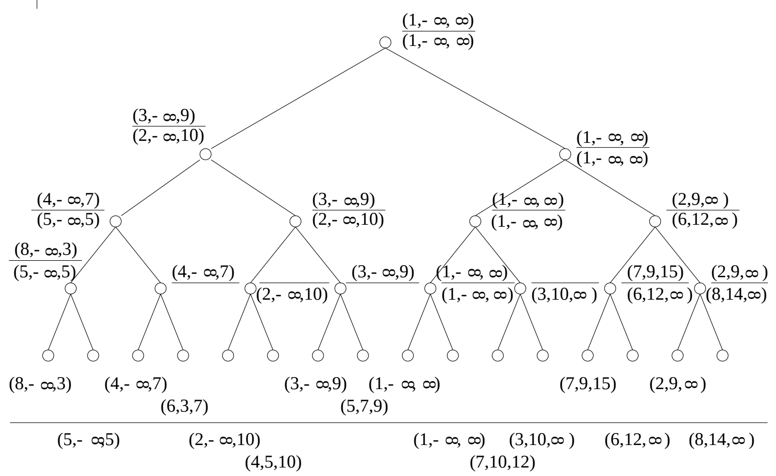

Figure 2: The observations of two objects and the corresponding FTR-tree.

Given a query (t_s, t_e, a), we can easily find the set of at most $2(\log m' - 1)$ *allocation nodes* in T that correspond to the interval $[t_s, t_e]$. An allocation node is a node whose corresponding time interval is fully contained in $[t_s, t_e]$ and that of whose parent is not. For each allocation node v, we know that all the $O(n)$ tuples in $S(v)$ correspond to observations taken during the time interval $[t_s, t_e]$. Therefore we only need to report those tuples in $S(v)$ that satisfy

$(s_i^j, e_i^j) \supset [t_{\mathrm{s}}, t_{\mathrm{e}}]$, and $v_i^j \geq a$. Lemma 1 guarantees that exactly one such tuple will be reported for each proper object whenever $t_i^j \in [t_s, t_e]$. No further search on v's descendants is necessary.

One final note is that, even though an object is stored multiple times in the form of its representative tuples, it will be reported at most once. This can be seen as follows. If an object is reported, then only one of its m tuples satisfies the conditions derived from the query. Even though a tuple may be stored in up to $\log m' + 1$ nodes, these nodes form a suffix of the path from the root to its corresponding leaf node and, as a result, only the allocation node will be considered.

For each allocation node v, looking for tuples $(v_i^j, t_i^j, s_i^j, e_i^j, o_i)$ that satisfy $(s_i^j, e_i^j) \supset [t_{\mathrm{s}}, t_{\mathrm{e}}]$ and $v_i^j \geq a$ is equivalent to a three dimensional dominance reporting problem, which can be solved in $O(\log n(v) + f(v))$ time and $O(n(v))$ space using the data structure of Makris and Tsakalidis [15], which we call the *dominance tree*, where $n(v)$ is the number of tuples stored in v and $f(v)$ is the number of tuples reported.

The storage cost of this data structure can be estimated as follows. First, since any tuple can appear at most once at each level of the tree T, the total number of tuples stored in T is $O(m \log m)$. Second, we have at most $2m - 1$ nodes in the tree T and each node stores at most n tuples. Hence the total number of tuples is $O(mn)$. Since the dominance tree associated with each node in T is linear in the number of tuples stored there, the overall storage cost is $O(m \min(\log m, n))$. As to the search complexity, finding the allocation nodes takes $O(\log m)$ time, and $O(\log n + f(v))$ is spent at each such node v with $f(v)$ tuples corresponding to proper objects. We thus have the following algorithm.

Theorem 1. *Using $O(m \min(\log m, n))$ space, any one-sided temporal range query involving n objects with a total number of m observations can be handled in $O(\log n \log m + f)$ time, where f is the number of objects satisfying the query.*

2.3 An $O(m)$-Space $O(\log m(\log n + f))$-Time Solution

The solution in the previous section requires non-linear space because a tuple could be stored at multiple levels of the primary tree T. For example, if the first two observations of object O_i are taken at time t_1 and t_9, then the tuple associated with v_i^1 is stored at least in the leftmost node at each of the three bottom levels of T. Indeed, it is easy to construct an example where each of the m observations will be replicated $O(\log(m/n))$ times.

To reduce the storage cost, we have to remove these duplicates. Consider an arbitrary observation v_i^j stored in an node u of T. We stipulate that v_i^j be removed from u if there is no observation of O_i stored in the sibling of u. We illustrate this new structure using the same example shown in Figure 3, with the tuples removed according to this new rule shown in gray color.

Lemma 2. *The modified data structure uses $O(m)$ space.*

Proof. Since the auxiliary data structure associated with each node of T is linear in the number of tuples stored there, we only need to show that the total number of tuples in T is $O(m)$. To accomplish this, it suffices to demonstrate that the total number of tuples corresponding to each object O_i is $O(m_i)$. This becomes obvious if we view the primary tree

t	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
O_1	8		4	6			3	5	1				7		2	
O_2		5			2	4				1	7	3		6		8

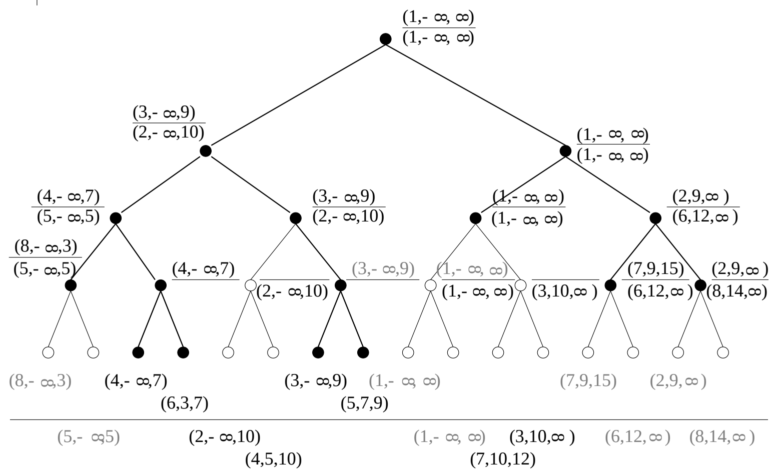

Figure 3: The observations of two objects and the corresponding CTR-tree.

nodes that contain a tuple of O_i as the nodes of another tree T_i. The children of a node u in T_i is its *nearest descendants* in T with regard to T_i. A node v in T is called a nearest descendant of u with regard to T_i if v is in T_i and there is no node v' in T_i such that v' is a descendant of u and an ancestor of v in T. It is easy to realize that T_i is a full binary tree, and each leaf of T_i contain a distinct tuple of O_i. Hence the number of tuples stored in T_i is $O(m_i)$. $\square$

A negative effect of this reduction in storage cost is that it is no longer sufficient to only search the allocation nodes corresponding to the time interval specified by the query, since a tuple that would previously be stored in an allocation node v may now only appear at some ancestors of v. To ensure the correctness of our algorithm, we search not only the allocation nodes, but also the nodes on the path from the root to them. Although no proper object will be missed in this process, some tuples that do not satisfy the conditions stated in Lemma 1 may be mistakenly reported. Consider a tuple $(v_i^j, t_i^j, s_i^j, e_i^j, o_i)$ found in an ancestor of an allocation node which satisfies the conditions $(s_i^j, e_i^j) \supset [t_\mathrm{s}, t_\mathrm{e}]$ and $v_i^j \geq a$. Its timestamp t_i^j could be outside $[t_\mathrm{s}, t_\mathrm{e}]$. Fortunately, the corresponding object O_i is still proper since the observation v_i^j is smaller than or equal to any observation during $[t_s, t_e]$. An object may be reported at most $O(\log m)$ times, once at each level of T.

The search time depends on the number of nodes visited, as $O(\log n)$ time is taken at each of them. The following lemma completes our complexity analysis.

Lemma 3. *The total number of nodes on the paths from the root to the allocation nodes is* $O(\log m)$.

Proof. Consider the embedded tree T' that consists of all the nodes on the paths from the root of T to the allocation nodes. Let Π_l (resp. Π_r) be the set of leftmost (resp. rightmost) nodes at each level of T'. It is easy to see that each internal node of T' which is not on Π_l nor on Π_r has two children, that each internal node on Π_l has a right child, and that each internal node on Π_r has a left child. For each internal node v of T' on Π_l that does not have a left child, we add one, and we add right children to those internal nodes on Π_r which do not have one. By doing so, we turn T' into a full binary tree; and we have added at most two leaf nodes during the process. Clearly, the number of internal nodes of the full binary tree T' is $O(\log m)$ as the number of its leaf nodes is $O(\log m)$. $\square$

Theorem 2. *Using $O(m)$ space, any one-sided temporal range query involving n objects with a total of m observations can be answered in $O(\log m \log n + f \log m)$ time, where f is the number of proper objects.*

2.4 An $O(m)$-Space $O(\log^3 m + f)$-Time Solution

In this section, we give a linear space solution that reports each proper object exactly once. We call it *linear temporal range tree* (*LTR-tree*). In designing the LTR-tree, we apply twice the interval tree techniques of Edelsbrunner [9] and use dominance trees to handle the queries.

Rewriting the conditions stated in Lemma 1, we have $t_\mathrm{s} \in (s_i^j, t_i^j]$, $t_\mathrm{e} \in [t_i^j, e_i^j)$, and $v_i^j \geq a$. Handling such a query can be viewed as a geometrical retrieval problem. Each tuple $(v_i^j, t_i^j, s_i^j, e_i^j, o_i)$ can be viewed as a rectangular plate in a three-dimensional space whose edges are parallel to the x- and y-axes, and whose projections to these two axes are $(s_i^j, t_i^j]$ and $[t_i^j, e_i^j)$ respectively; and the query can be viewed as finding the plates that are intersected by a ray perpendicular to the x-y plane shooting in the direction of positive z-axis from the point $(t_\mathrm{s}, t_\mathrm{e}, a)$. Figure 4 illustrates such a geometrical interpretation of the query.

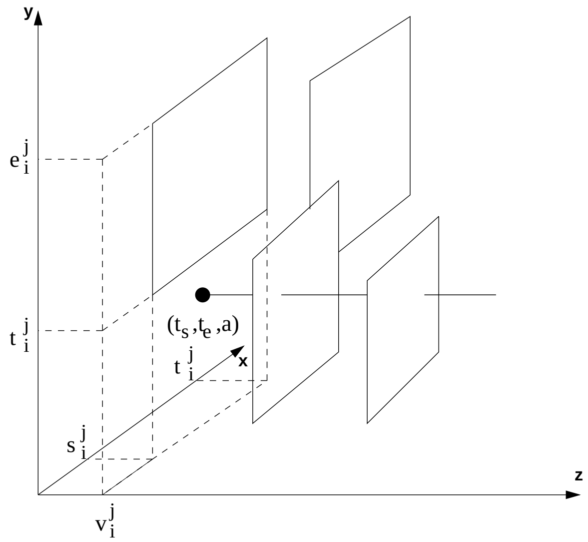

Figure 4: Illustration of the plate intersection problem.

We consider the projections of these plates to the x-y plane, which are rectangles. The primary structure of the LTR-tree is a balanced binary tree built by recursively partitioning these rectangles according to the x-coordinates of their vertical edges. We choose a vertical line $x = x(r)$ such that half of the distinct verticals of these rectangles are to the left of it and the other half to the right, and store the value $x(r)$ at the root node r of the primary interval tree T. This vertical line partitions the set of rectangles into three groups: those whose corresponding horizontal edges are intersected by the partition line: $\{(v_i^j, t_i^j, s_i^j, e_i^j, o_i) | s_i^j < x(r) \le t_i^j\}$; those rectangles that are completely to its left: $\{((v_i^j, t_i^j, s_i^j, e_i^j, o_i) | t_i^j < x(r)\}$; and those completely to its right: $\{(v_i^j, t_i^j, s_i^j, e_i^j, o_i) | s_i^j \ge x(r)\}$. We associate the tuples that correspond to the first group of rectangles with the root node and recursively construct its left and right subtrees for the tuples corresponding the latter two sets of rectangles respectively. See Figure 5 for an example, in which the rectangles A, B, and E are associated with the node r; D and G with the subtree rooted at node u; and C and F with the subtree rooted at node v.

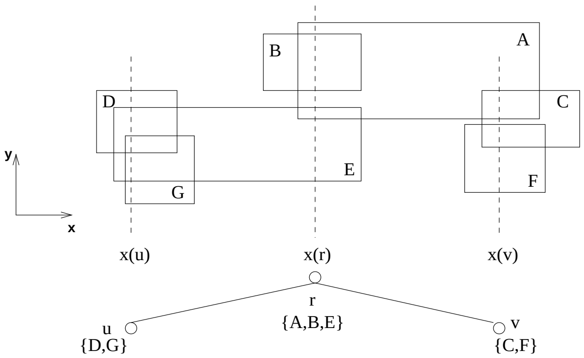

Figure 5: An LTR-tree.

Each node v of T is now associated with a set $S(v)$ of tuples. Similar to the construction of the primary interval tree, we build a secondary interval tree $T(v)$ on $S(v)$, this time based on the y-coordinates of the horizontal edges of their corresponding rectangles. By doing so, we further distribute the tuples in $S(v)$ to the nodes of $T(v)$. To be exact, a node α of $T(v)$ with a partition line $y = y(\alpha)$ contains tuples that satisfy $t_i^j \le y(\alpha) < e_i^j$; its left subtree contains tuples that satisfy $e_i^j \le y(\alpha)$; and its right subtree contains tuples that satisfy $t_i^j > y(\alpha)$.

For the tuples associated with each node μ of $T(v)$, we construct four versions $T_0(\mu)$, $T_1(\mu)$, $T_2(\mu)$, and $T_3(\mu)$ of the dominance tree, for the following point dominance queries respectively: 1. $(t_s > s_i^j, t_e \ge t_i^j, v_i^j \ge a)$; 2. $(t_s > s_i^j, t_e < e_i^j, v_i^j \ge a)$; 3. $(t_s \le t_i^j, t_e \ge t_i^j, v_i^j \ge a)$; 4. $(t_s \le t_i^j, t_e < e_i^j, v_i^j \ge a)$.

To analyze the storage cost of this data structure, we first notice that a tuple is stored in exactly one secondary tree. In this secondary tree, it is stored in at most 4 dominance trees. Since a dominance tree is linear in the number of tuples stored there, the total size of all the dominance trees is $O(m)$; for the same reason, the overall size of the secondary tree is also $O(m)$; and finally, the primary tree is of size $O(m)$.

To answer a query (t_s, t_e, a), we start from the root of the primary tree r. We first access the secondary tree $T(r)$ to report tuples stored at r. Then we check if $t_s \le x(r)$. If this is the case, we recursively access the subtree rooted at r's left child; otherwise, we recursively access the subtree rooted at r's right child. When accessing a secondary tree $T(u)$, we start from its root α. We first compare t_e with $y(\alpha)$. Depending on whether $t_s \le x(u)$ and whether $t_e < y(\alpha)$, we access one of the four dominance trees associated with α. More specifically, we access $T_0(\alpha)$ if $t_s \le x(u)$ and $t_e < y(\alpha)$; $T_1(\alpha)$ if $t_s \le x(u)$ and $t_e \ge y(\alpha)$; $T_2(\alpha)$ if $t_s \ge x(u)$ and $t_e < y(\alpha)$; and $T_3(\alpha)$ if $t_s \ge x(u)$ and $t_e \ge y(\alpha)$. After the points associated with α are reported, we recursively access its left child if $t_e \leq y(\alpha)$ or its right child otherwise.

To demonstrate the correctness of the query algorithm, let's follow one search path during the handling of query (t_s, t_e, a). At a node v being visited, suppose $t_s \leq x(v)$; then all the tuples stored in $T(v)$ satisfy $t_s \leq t_i^j$. At a node α of $T(v)$ being visited, suppose $t_e \geq y(\alpha)$. Then all the tuples stored in $T(v)$ satisfy $t_e \geq t_i^j$. Therefore, we only need to filter these tuples using the conditions ($t_s > s_i^j$, $t_e < e_i^j$, and $v_i^j \geq a$), which is exactly what $T_1(\alpha)$ is designed to do. It is easy to verify that the tuples stored in $T_0(\alpha)$, $T_2(\alpha)$, and $T_3(\alpha)$ cannot satisfy the query. Also due to the fact that $t_e \geq y(\alpha)$, no tuples stored in the α's left subtree will be reported; so we only need to recursively access its right subtree. Similarly, when the access to the node v is finished, we only need to recursively access its left subtree.

Now we analyze the complexity of the query algorithm. To answer a query, we need to access $O(\log m)$ primary tree nodes, one at each level. For each such node v, $O(\log m)$ nodes in $T(v)$ need to be processed. And finally, for each node α in $T(v)$ visited, $O(\log m + f(\alpha))$ time is spend to access one of its associated dominance trees, where $f(\alpha)$ is the number of tuples reported. Note that each tuple that satisfies query will be reported exactly once.

Theorem 3. *Using $O(m)$ space, any one-sided temporal range query involving n objects with a total of m observations can be answered in $O(\log^3 m + f)$ time, where f is the number of proper objects.*

3 One-Sided Temporal Range Queries: The Dynamic Case

In this section, we consider the problem of designing dynamic indexing structures that enable the quick handling of temporal range queries and at the same time can be efficiently updated when new observations are added. As stated before, we make the assumption that the timestamp of a new observation of an object O_i is larger than that of any existing observation of O_i. Note that adding a new object simply means adding the first observation of that object.

Since our solution will use the characterization given in Lemma 1, we need to examine the changes that will occur to the object's tuples when a new observation of that object is inserted. In Section 3.1, we show how to quickly determine the tuples that need to be updated and the one tuple to be inserted due to the introduction of a new observation. Since our algorithms will use the 3-D dominance query data structure, we introduce a data structure for the dynamic case in Section 3.2. This data structure is a crucial component of the dynamic versions of our temporal range trees, which will be presented in Sections 3.3 through 3.5.

We will describe in detail the "dynamization" of the FTR-tree. The techniques introduced can also be used to "dynamize" the other two temporal range trees. Therefore, for these two structures, we will only comment on the new issues they raise.

3.1 Creating and Updating Tuples

The addition of new observations may require that many of the existing tuples corresponding to the same object be updated to reflect the possible change of their dominant intervals. To facilitate the quick identification of such tuples, we maintain a *Cartesian tree* [27] C_i for each object O_i. A Cartesian tree for a sequence $(t_i^j, v_i^j), 1 \leq j \leq m_i$, is a binary tree with m_i nodes. The root stores the smallest value v_i^j over the time interval $[t_i^1, t_i^{m_i}]$, where j is the smallest such index if multiple minima exist. Its left child is the root of the Cartesian tree for observations $\{v_i^1, \ldots, v_i^{j-1}\}$; and its right child is the root of the Cartesian tree for observations $\{v_i^{j+1}, \ldots, v_i^{m_i}\}$. Note that a node may not have a left or right child. The Cartesian tree C_i can be built in $O(m_i \log m_i)$ time by inserting the observations in order of their timestamps, using an algorithm that we discuss later.

Let $v_i^{m_i+1}$ be the new observation of object O_i with a timestamp $t_i^{m_i+1}$, where $t_i^{m_i+1} > t_i^{m_i}$. Let Π_i be the rightmost path of the Cartesian tree C_i before the addition of $v_i^{m_i+1}$ and π_i be the prefix of Π_i such that each node on π_i, except the root, is the right child of its parent. To update C_i, we first find the pair of parent-child nodes u_{pred} and u_{succ} on π_i and the corresponding observations $v_i^{j_{\text{pred}}}$ and $v_i^{j_{\text{succ}}}$ such that $v_i^{j_{\text{pred}}} \leq v_i^{m_i+1} < v_i^{j_{\text{succ}}}$. Note that u_{pred} (u_{succ}) is null if $v_i^{m_i+1}$ is less than (greater than or equal to) all the observations associated with π_i. Since the values of the observations corresponding to the nodes on π_i are non-decreasing from the root, we can easily find this pair in $O(\log m_i)$ time using binary search.

Now consider the parent node u_{pred} and the child node u_{succ}. If u_{pred} is not null, then the new node u that corresponds to $v_i^{m_i+1}$ becomes its right child. If u_{succ} is not null, it becomes the left child of u. The node u becomes the root of the new C_i if u_{pred} is null. Figure 6 shows how the additions of two new observations $v_i^9 = 1$ and $v_i^{10} = 5$ for object O_i is handled. The path π_i can be maintained in an array whose size is h_i, where h_i is the length of π_i.

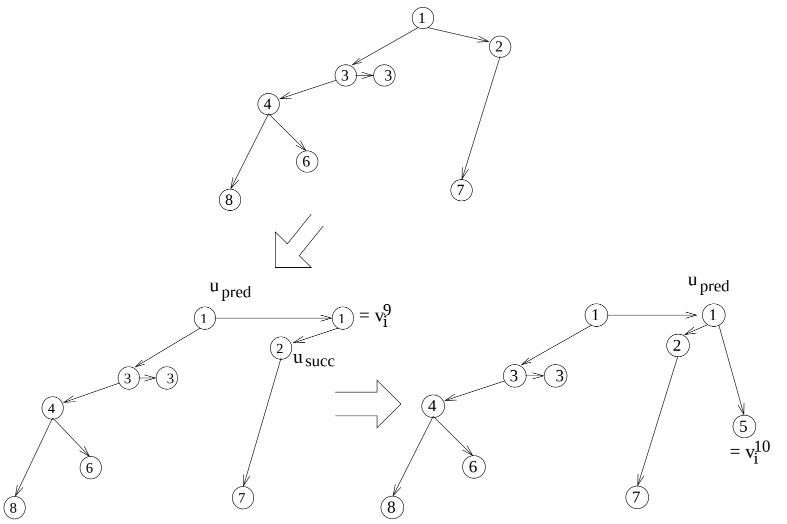

Figure 6: The addition of two new observations to an existing Cartesian tree.

The following lemma guarantees that on average, only a constant number of tuples need to be updated.

Lemma 4. *Let m_i be the number of observations of object O_i maintained in the current Cartesian tree C_i prior to the insertion of the new observation $v_i^{m_i+1}$ with a time stamp t_{m_i+1}. Then the g tuples that need to be updated after inserting $v_i^{m_i+1}$ can be identified in $O(\log m_i + g)$ time. Furthermore the amortized value of g over the next m_i insertions corresponding to O_i is at most 2.*

Proof. Consider the insertion of a new observation $v_i^{m_i+1}$ of O_i. It easy to realize that for any $1 \leq k \leq m_i$, if $v_i^k \leq v_i^{m_i+1}$, then its dominant interval (s_i^k, e_i^k) does not change. Furthermore, for any observation v_i^j stored in a descendent of a left child of a node v on π_i, we know that its dominant interval will be limited from the right by the timestamp of $v_i^{j'}$ which is stored in v and thus will not be affected by the insertion of the new instance. Therefore we only need to update the tuples whose corresponding observations are associated with the suffix of π_i, starting from u_{succ}. Indeed, for each such tuple, whose dominant interval is in the form $(s, +\infty)$, the updated dominant interval will be $(s, t_i^{m_i+1})$. We also need to add the tuple $(v_i^{m_i+1}, t_i^{m_i+1}, j_{\text{pred}}, +\infty, o_i)$ that corresponds to the (m_i+1)-th observation of O_i, or $(v_i^{m_i+1}, t_i^{m_i+1}, -\infty, +\infty, o_i)$ if u_{pred} is null.

Let h_i^j be the length of π_i in the version of C_i with j observations and l_i^j be the *height* of the node u_{prec}, i.e., the number of nodes on the path from the root to u_{prec}, including u_{prec}. It is clear from the previous analysis that $h_i^j - l_i^j$ existing tuples need to be updated. These tuples can be retrieved in $O(1)$ time each, provided that u_{succ} has been located, a task that can be done in $O(\log h_i^j)$ time. Furthermore, the length h_i^{j+1} of the new π_i after the addition is $l_i^j + 1$.

Now consider the m_i consecutive additions of new observations of object O_i, assuming that m_i observations have already been recorded. The number of tuples to be updated during these m_i insertions is given by:

$$\sum_{j=m_i}^{2m_i-1} h_i^j - l_i^j = \sum_{j=m_i}^{2m_i-1} h_i^j - (h_i^{j+1} - 1) = h_i^{m_i} - h_i^{2m_i} + m_i < 2m_i. \tag{1}$$

Therefore the amortized number of tuples to be modified per insertion over the next m_i insertions is less than 2. □

The following lemma shows that that the aggregate number of tuples that need to be updated over a sequence of inserting m observations is less than $2m$.

Lemma 5. *Let m be the number of observations maintained in the current primary data structure corresponding to all the objects. Then the aggregate number of tuples that need to be updated over the insertions of the next k new observations is less than $m + k$.*

Proof. Let n' be the number of objects in the data structure after the k insertions, and let m_i and k_i, with $i = 1, \ldots n'$, be respectively the numbers of current and new observations corresponding to object O_i. Note that O_i may not have any observations in the current data structure, in which case $m_i = 0$.

Using the notation of the proof of Lemma 4 and similar to Equation 1, we can calculate the number of tuples to be updated during these m insertion as

$$\begin{aligned} & \textstyle\sum_{i=1}^{n'} \sum_{j=m_i}^{m_i+k_i-1} \left(h_i^j - l_i^j\right) \\ = & \textstyle\sum_{i=1}^{n'} \sum_{j=m_i}^{m_i+k_i-1} \left(h_i^j - h_i^{j+1} + 1\right) \\ = & \textstyle\sum_{i=1}^{n'} \left(h_i^{m_i} - h_i^{m_i+k_i} + k_i\right) \\ < & \textstyle\sum_{i=1}^{n'} (m_i + k_i) \\ = & m + k. \end{aligned} \tag{2}$$

□

Lemma 5 allows us to handle the insertions by first identifying the old tuples that need to be updated, followed by performing each of the updates, and finally adding the new tuple. Note that if we start from an empty data structure, then the amortized cost for including a new observation is one update operation and one insertion. On the other hand, if we start from a data structure that already contains m tuples, then the amortized cost for inserting a new observation over the next m insertions is at most two update operations and one insertion.

3.2 Dynamic Data Structures for 3-D Dominance Queries

Once we identify the tuples to be modified and the new tuple to be inserted, we need to update the corresponding data structures for handling dominance queries. In particular, we have to remove the points corresponding to old tuples and insert the points associated with the updated tuples. Although the dominance tree described before has very good performance in terms of space and query time, it does not appear to be suitable for the dynamic case. To make our data structure dynamic, we use a combination of the range tree and the priority search tree, a structure that we will refer to as the *dynamic dominance tree*, to solve the 3-D dominance query problem. We now elaborate on this data structure using the version of dominance query in which we are asked to find all the points $p = (p_{\mathrm{x}}, p_{\mathrm{y}}, p_{\mathrm{z}})$ that are dominated by a query point $q = (q_{\mathrm{x}}, q_{\mathrm{y}}, q_{\mathrm{z}})$, i.e., $p_x \leq q_x$, $p_y \leq q_y$, and $p_z \leq q_z$.

Given a set of n three dimensional points, we first build a *weight-balanced tree* [2] T of degree c on the z-coordinates sorted in increasing order, where c is a constant. A weight-balanced tree storing n points is a dynamic search tree of $O(\log n)$ height which supports insertion and search in $O(\log n)$ time. More importantly, if a node whose subtree has w leaf nodes is split, then for each new node created as a result of this split, at least $\Omega(w)$ insertions have to pass through it to make it split again.

For each internal node v, we build a priority search tree [17] that stores the set of points in the subtree of v projected onto the x-y plane. Recall that a priority search tree containing n elements requires $O(n)$ space and $O(n \log n)$ preprocessing time, and can handle search, insertion, and deletion operations in $O(\log n)$ time [17]. A dominance query can be answered by first identifying the $O(c \log n)$ allocation nodes in T that together correspond to the z-range $(-\infty, q_z]$, and then searching the corresponding priority search trees to answer the query $(p_{\mathrm{x}} \leq q_{\mathrm{x}}; p_{\mathrm{y}} \leq q_{\mathrm{y}})$. The query time is $O(\log^2 n + f)$ and the space required by the data structure in $O(n \log n)$.

To insert a point, we first perform a *virtual* insertion to handle any necessary node split in T. When a node is split, the priority search trees of the two newly created nodes are built from scratch. Since the total size of the priority search trees stored in a subtree rooted at a node v is asymptotically the same as the number of leaves in that subtree, the amortized cost of this split is $O(\log n)$. After that, the new point is inserted into T as well as into the priority search trees on the path from the root to its corresponding leaf node. This process takes $O(\log^2 n)$ time.

Deletion can be done using *global rebuilding* technique [19]. For each node on the path from the root to the leaf corresponding to the point being deleted, we remove this point from its associated priority search tree in an overall $O(\log^2 n)$ time. We do not delete the leaf node in the primary tree at this time. Instead, we wait until $n/2$ deletions have happened and then rebuild the entire data structure using $O(n \log^2 n)$ time.

Generalization of the above results to higher dimensions is straightforward. and can be summarized by the following lemma.

Lemma 6. *For any $d \geq 2$, using $O(n \log^{d-2} n)$ space and $O(n \log^{d-1} n)$ preprocessing, we can store n d-dimensional points in a data structure such that dominance queries can be answered in $O(\log^{d-1} n + f)$ time and updates can be performed in $O(\log^{d-1} n)$ amortized time.*

3.3 Dynamic FTR-tree

To make the structure in Section 2.2 dynamic, we replace the binary tree built on the time instances by a weight-balanced tree T of degree c. Each node is associated with a set of tuples, each representing an object. The dominant interval of a tuple associated with an internal node v contains the dominant intervals of all the tuples stored in the subtree of v representing the same object. With each node v of T, we store the dynamic dominance tree structure $T_{\mathrm{dom}}(v)$ built on the tuples stored at v, and a dynamic binary search tree, say a red-black tree [7], $T_{\mathrm{key}}(v)$ built on the keys associated with these tuples. It can be shown using similar arguments as in the static case that the size of our data structure will be $O(m \log n \min\{\log m, n\})$ (the extra $\log n$ factor is due to the dynamic dominance tree structure being used).

The query process is almost the same as in the static case. The only difference is that we now have up to $O(c \log m)$ allocation nodes, each of which takes $O(\log^2 n + f(v))$ time to search.

There are two major steps required to update our overall data structure. The first is to update the tuples that are no longer valid, and the second is to insert the new time stamp and the new tuple into the primary tree.

Consider the update step. Suppose that the tuple $(v_i^l, t_i^l, s_i^l, e_i^l, o_i)$ needs to be updated. Notice that the entry t_i^l of this tuple does not change. Therefore, there is no need to update the primary tree. Furthermore, we have the following lemma.

Lemma 7. *An updated tuple associated with a previous observation should be stored in the auxiliary tree structures $T_{\mathrm{dom}}(v)$ and $T_{\mathrm{key}}(v)$ of the new primary structure if and only if the old tuple is also stored there.*

Proof. This lemma is immediate once we realize that the node at which a tuple $(v_i^j, t_i^j, s_i^j, e_i^j, o_i)$ resides depends solely on its value v_i^j and timestamp t_i^j, which do not change when a new observation is inserted. □

Therefore, what we need to do is to go through each node on the path from the root to the leaf node corresponding to t_i^l. For each node v on this path, we search $T_{\mathrm{key}}(v)$ using o_i to find the old tuple and replace it with the new one. Then we remove the same old tuple from, and insert the new tuple into, $T_{\mathrm{dom}}(v)$. The whole process takes $O(\log m \log^2 n)$ time.

To add a new tuple $(v_i^{j+1}, t_i^{j+1}, s_i^{j+1}, +\infty, o_i)$, we first insert the new time instance into the primary tree T. This may cause up to $O(\log m)$ nodes to split, which can be handled in $O(\log m \log^2 n)$ amortized time following similar arguments as in Section 3.2. To insert the new tuple, we traverse the path from the leaf node corresponding to t_i^{j+1} up toward the root. At each node v visited, we search the representative tuple for O_i in $T_{\text{key}}(v)$ using o_i. If there is no such tuple, we insert the new tuple into both $T_{\text{key}}(v)$ and $T_{\text{dom}}(v)$. If one such tuple is found, we check if it needs to be replaced by the new tuple. If it does, then we remove the old tuple from and insert the new tuple into both $T_{\text{key}}(v)$ and $T_{\text{dom}}(v)$. Otherwise, we do not need to visit any of v's ancestors.

Theorem 4. *Any temporal range query involving n objects with a total number of m observations can be answered in $O(\log m \log^2 n + f)$ time using a data structure of size $O(m \log n \cdot \min\{\log m, n\})$. This data structure can be constructed in $O(m \log m \log^2 n)$ time and updated in $O(\log m \log^2 n)$ amortized time over the next m updates.*

3.4 Dynamic CTR-tree

Since a CTR-tree is derived from its corresponding FTR-tree, a dynamic CTR-tree is derived from the corresponding dynamic FTR-tree by removing the representative tuple p of object O_i from a node v if none of its siblings contains an observation of O_i. It is easy to show that the storage cost of the dynamic CTR-tree is still $O(m \log n)$ and the query time is still $O(\log m \log^2 n + f \log m)$.

Notice that Lemma 7 holds for CTR-trees as well. Therefore, updating an exiting tuple in a CTR-tree takes $O(\log m \log^2 n)$ time.

Now consider the process of adding a new tuple. If a node v splits into v' and v'' during the insertion of the new time instance, we rebuild, as we did for the FTR-tree, the subtrees rooted at v' and v'' completely. The amortized cost is $O(\log m \log^2 n)$. The only additional detail we need to examine carefully is whether the set of tuples associated with the parent u of v might change. Notice u does not have a representative tuple of object O_i if and only if either (i) no observation of O_i was taken during the time period associated with u; or (ii) the set of observations associated with the subtree rooted at u is the same as that associated with the subtree rooted at u's parent. These two conditions will not change as a result of the node-split, and hence the tuples associated with u will not change.

Finally, we comment on the insertion of the new tuple $p = (v_i^{j+1}, t_i^{j+1}, s_i^{j+1}, +\infty, o_i)$. As we did in the case of dynamic FTR-trees, we traverse the path Π from the root of T to the leaf node corresponding to t_i^{j+1}. At each node v, we first use $T_{\text{key}}(v)$ to identify the representative tuple of O_i in $S(v)$. If no such tuple is found, i.e. no tuple corresponding to O_i is stored in the subtree rooted at v, we simply insert the new tuple in $T_{\text{key}}(v)$ and $T_{\text{dom}}(v)$. If there is such a tuple, say $q = (v_i^l, t_i^l, s_i^l, +\infty, o_i)$, we check whether the dominant interval of p contains that of q and in the affirmative replace q with p in both $T_{\text{dom}}(v)$ and $T_{\text{key}}(v)$, and continue to visit the next node on Π. Unlike the FTR-tree, when the tuple q is replaced by p, and p and q belong to subtrees of different children of v, we need to insert q to the root of the subtree it belongs to. Hence, the process of inserting the new tuple takes $O(\log m \log^2 n)$ time.

Theorem 5. *Any temporal range query involving n objects with a total number of m observations can be answered in $O(\log m(\log^2 n + f))$ time using $O(m \log n)$ space. This data structure can be constructed in $O(m \log m \log^2 n)$ time and updated in $O(\log m \log^2 n)$ amortized time.*

3.5 Dynamic LTR-tree

To make the LTR-tree dynamic, we replace the primary and secondary binary search trees with weight-balanced trees of degree c. In either the primary tree or the secondary tree, a node is thus associated with $c-1$ partition lines; and a tuple is associated with a node if its corresponding rectangle intersects at least one of its partition lines.

Let $S(u)$ be the set of tuples associated with a primary tree node u, $S(u)$ is partitioned into $\Theta(c^2)$ subsets, each containing the tuples whose corresponding rectangles intersect a specific pair of vertical partition lines and is organized as a secondary tree $T_{g,h}(u)$ with $0 < g \leq h < c$. Similarly the set of tuples associated with each such secondary tree node α is indexed using $\Theta(c^2)$ dynamic dominance trees, four of different versions for each pair of horizontal partition lines.

It is easy to see that the overall storage cost of this data structure is $O(m \log m)$, the query time is $O(\log^4 m + f)$, and the preprocessing time is $O(m \log^2 m)$. Following the same arguments as in Section 3.3, and by using the properties of the weight balanced trees and the techniques of global rebuilding, it is not difficult to show that we can insert a new tuple in $O(\log^3 m)$ time and delete an old tuple in $O(\log^2 m)$ time, both amortized. Thus updating an LTR-tree takes $O(\log^3 m)$ time.

Theorem 6. *Any temporal range query involving n objects with a total number of m observations can be answered in $O(\log^4 m + f))$ time using a $O(m \log m)$ space data structure. This data structure can be constructed in $O(m \log^2 m)$ time and updated in $O(log^3 m)$ amortized time.*

4 Handling The General Temporal Queries

For the general problem, we assume that we have a predefined time hierarchy imposed on our time line, say starting at a fixed time instance t_0 until $t_{m+1} = +\infty$, such that all queries involve one of the time intervals defined in this hierarchy. This is indeed the case in many applications. In fact, the hierarchy "day→week→month→season→year" is widely used for applications such as OLAPs. We are interested in queries that will identify objects whose attributes fall within certain ranges at every time instance in one of the time intervals defined by the hierarchy. As a specific application, consider a set of probes located in a large number of geographic areas, each collecting a number of measures (say temperature, humidity, snowfall, wind speed, pressure, etc.) and sending the information to a server - they may arrive at different times but will have a timestamp indicating when the information was recorded. A typical query would be to determine the regions which, during the first week of February 2002, the temperature was higher than 30°C and the snowfall smaller than 2 inches during each day of that week.

Let us formally define our time hierarchy as a tree $T = (V, E)$. Each node v of T is associated with a time interval $I(v) = [t_s, t_e)$ at a certain level of this hierarchy. An internal node v has a set of children that correspond to the time intervals of a finer granularity. Except for the root, which is associated with the time interval $[t_0, +\infty)$, the time interval associated with any other internal node v is $I(v) = \cup_{u \in \mathbf{children}(v)} I(u)$. The leaves correspond to time intervals of the finest granularity in the hierarchy. For example, for the "day→week→month→season→year" hierarchy, the root corresponds to the entire history of the data set. Each child of the root represents a year and has four children, each corresponding to a different season of that year, etc. Let $I(v)$ be the time interval associated with node v, a query of type Q′ is defined as follows.

> Q′. Given two vectors $\mathbf{a} = [a_1, a_2, \ldots, a_d]$ and $\mathbf{b} = [b_1, b_2, \ldots, b_d]$, and a node $v \in V$, determine the set Q of objects such that $O_i \in Q$ if and only if the following two conditions are true:
>
> - There exist at least one observation taken at time t_i^j such that $t_i^j \in I(v)$.
> - For every observation $\mathbf{v}_i^j$ such that $t_i^j \in I(v)$, we have $a_k \leq v_{i,k}^j \leq b_k$ for $k = 1, 2, \ldots, d$.

We store at each node v a set $S(v)$ of $(2d+1)$-tuples: $S(v) = \{(\min_{i,1}^v, \max_{i,1}^v, \ldots, \min_{i,d}^v, \max_{i,d}^v, o_i) | \exists j, t_i^j \in I(v)\}$, where $\min_{i,l}^v$ and $\max_{i,l}^v$ are the minimum and maximum values of the lth attribute of O_i during the time interval $I(v)$. Note that if there is no observation for O_i during the time interval $I(v)$, then there is no tuple in $S(v)$ representing O_i. To be able to tell which objects are represented in v, we maintain a *red-black* tree $T_{\text{key}}(v)$ to index the tuples in $S(v)$ on the keys o_i.

By observation 1, we can answer a query of type Q′ by determining the $(2d-1)$-tuples at v which satisfy: $\max_{i,1}^v \leq b_l$ and $\min_{i,1}^v \geq a_l$, for all $l = 1, 2, \ldots, d$. Finding such tuples in $S(v)$ is equivalent to answering a (2d)-dimensional dominance query. By Lemma 5, there exists a data structure $T_{\text{dom}}(v)$ of size $O(n \log^{2d-2} n)$ such that the proper objects in $S(v)$ can be reported in $O(\log^{2d-1} n + f(v))$ time. The total number of tuples stored in T is $O(m)$, since each tuple is stored in a constant number of nodes, one at each level of the hierarchy (the number of hierarchy levels is assumed to be constant independent of the number of observations). Let $n(v)$ be the number of tuples stored in v. The overall size of the data structure is

$$O\left(\sum_{v \in V} n(v) \log^{2d-2} n(v) + m'\right) = O\left(\log^{2d-2} n \sum_{v \in V} n(v) + m'\right) = O\left(m \log^{2d-2} n + m'\right),$$

where m' is the number of leaves in T, which is typically much smaller than m. The construction of this data structure is straightforward. We first use $O(m)$ time to construct the set $S(v)$ for each node v. We then spend $O(n(v) \log n(v))$ to build $T_{\text{key}}(v)$ and $O(n(v) \log^{2d-1} n(v))$ time to build $T_{\text{dom}}(v)$. The overall preprocessing time is $O(m \log^{2d-1} n + m')$.

When a new observation $\mathbf{v}_i^{m_i+1}$ of object O_i is added, we first look for the leaf node u such that t_i^j is in its corresponding time interval. We distinguish between two cases as described below:

> Case 1. A leaf node u containing t_i^j already exists in our structure. We visit nodes on the path from u to the root. For each such node v, we first look for the representative tuple of O_i in $S(v)$ by searching $T_{\text{key}}(v)$, which takes $O(\log n(v))$ time. If such a tuple is found, we compare $v_{i,k}^{m_i+1}$ against $\min_{i,k}^v$ and $\max_{i,k}^v$ for each $1 \leq k \leq d$, and update the maximum or minimum value if necessary. If one of the maximum or minimum values is updated, then the old $(2d+1)$-tuple is removed from $T_{\text{dom}}(v)$ and the new tuple is inserted. Only $O(\log^{2d-1} n)$ time is needed to perform this task.

Case 2. No leaf node containing t_i^j exists. We need to add a sequence of leaf nodes to the right of the rightmost leaf of the existing tree, so that the time interval of last node u added covers the time instance $t_i^{m_i+1}$, and the new tuple $(v_{i,1}^{m_i+1}, \ldots, v_{i,d}^{m_i+1}, o_i)$ is inserted into the empty node u. If the newly added nodes are children of the rightmost node v one level higher in the hierarchy, the trees $T_{\text{key}}(w)$ and $T_{\text{dom}}(w)$ for each node w on the path from v to the root are updated using $v_i^{m_i+1}$ as described in Case 1. Otherwise, new nodes at this level need to be added. We repeat the same process until we reach the level just below the root. Since the root is associated with the entire history of the data set, the process of adding new nodes is guaranteed to end at this level. The complexity of adding the new observation in this case is $O(\log^{2d-1} n + \Delta m')$, where $\Delta m'$ is the number of new leaf nodes added.

We thus have the following theorem.

Theorem 7. *Any temporal range query of type Q' involving n objects with a total number of m observations, and involving a time hierarchy with m' predefined time intervals at the lowest level, can be answered in $O(\log^{2d-1} n + f)$ time using $O(m \log^{2d-2} n + m')$ space. The preprocessing takes $O(m \log^{2d-1} n + m')$ time. Any new observation can be added in $O(\log^{2d-1} n + \Delta m')$ amortized time, where $\Delta m'$ is the number of new time intervals added to the lowest level of the hierarchy.*

5 Adding Key Ranges to the Search

By increasing the storage by a factor of $O(\log n)$, we can extend all the previous data structures so that they can be used to answer queries that not only specify the time and value ranges, but also the key ranges. That is, only a subset of the proper objects O_i, those with keys between k_1 and k_2 satisfying the temporal range constraints will be reported. We use a dynamic balanced binary tree to index the tuples according to their keys. Each node of this tree is thus associated with a key range; and we attach one of the data structures described in the previous sections, containing only tuples within this key range. The query times are increased by a factor of $O(\log n)$.

References

[1] P. K. Agarwal, L. Arge, and J. Erickson. Indexing moving points. In *19th ACM Symposium on Principles of Database Systems*, pages 175–186, 2000.

[2] L. Arge and J. S. Vitter. Optimal dynamic interval management in external memory. In *37th Annual Symposium on Foundations of Computer Science*, pages 560–569, Burlington, Vermont, Oct. 1996.

[3] V. P. Chakka, A. Everspaugh, and J. M. Patel. Indexing large trajectory data sets with SETI. In *First Biennial Conference on Innovative Data Systems Research*, 2003.

[4] B. Chazelle. Filtering search: A new approach to query-answering. *SIAM Journal on Computing*, 15(3):703–724, Aug. 1986.

[5] B. Chazelle. Lower bounds for orthogonal range search I. The reporting case. *Journal of the ACM*, 37(2):200–212, 1990.

[6] B. Chazelle and H. Edelsbrunner. Linear space data structures for two types of range search. *Discrete Comput. Geom.*, 3:113–126, 1987.

[7] Cormen, Leiserson, and Rivest. *Introduction to Algorithms.* MIT Press, 1990.

[8] J. R. Driscoll, N. Sarnak, D. Sleattor, and R. E. Tarjan. Make data structures persistent. *J. of Compu. and Syst. Sci.*, 38:86–124, 1989.

[9] H. Edelsbrunner. A new approach to rectangle intersections, part I. *Int. J. Computer Mathematics*, 13:209–219, 1983.

[10] P. Gupta, R. Janardan, and M. Smid. Further results on generalized intersection searching problems: counting, reporting, and dynamization. *Journal of Algorithms*, 19:282–317, 1995.

[11] R. Janardan and M. Lopez. Generalized intersection searching problems. *International Journal of Computational Geometry & Applications*, 3(1):39–69, 1993.

[12] K. V. R. Kanth and A. K. Singh. Optimal dynamic range searching in non-replicating index structures. In *Proceedings of the 7th International Conference on Database Theory*, pages 257–276, Jerusalem, Israel, Jan. 1999.

[13] G. Kollios, D. Gunopulos, and V. J. Tsotras. On indexing mobile objects. In *Proceedings of the Eighteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 261–272, 1999.

[14] S. Lanka and E. Mays. Fully persistent B^+-trees. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 426–435, 1991.

[15] C. Makris and A. K. Tsakalidis. Algorithms for three-dimensional dominance searching in linear space. *Information Processing Letters*, 66(6):277–283, 1998.

[16] Y. Manolopoulos and G. Kapetanakis. Overlapping B^+-trees for temporal data. In *Proceedings of the 5th Jerusalem Conference on Information Technology*, pages 491–498, 1990.

[17] E. M. McCreight. Priority search trees. *SIAM Journal on Computing*, 14(2):257–276, May 1985.

[18] M. A. Nascimento and J. R. O. Silva. Towards historical R-trees. In *Proceedings of the ACM Symposium on Applied Computing*, pages 235–240, Feb. 1998.

[19] M. H. Overmars. *The design of dynamic data structures.* Springer-Verlag, LNCS 156, 1983.

[20] D. Pfoser, C. S. Jensen, and Y. Theodoridis. Novel approaches in query processing for moving object trajectories. In *Proceedings of 26th International Conference on Very Large Databases*, pages 395–406, Sept. 2000.

[21] S. Saltenis, C. S. Jensen, S. T. Leutenegger, and M. A. Lopez. Indexing the positions of continuously moving objects. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, pages 331–342, 2000.

[22] B. Salzberg and V. J. Tsotras. Comparison of access methods for time-evolving data. *ACM Computing Surveys*, 31(2):158–221, 1999.

[23] Q. Shi and J. JaJa. A new framework for addressing temporal range queries and some preliminary results. Technical Report CS-TR-4438, Institute of Advanced Computer Studies (UMIACS), University of Maryland, 2003.

[24] Y. Tao and D. Papadias. Efficient historical R-trees. In *Proceedings of the 13th International Conference on Scientific and Statistical Database Management*, pages 223–232, 2001.

[25] T. Tzouramanis, Y. Manolopoulos, and M. Vassilakopoulos. Overlapping Linear Quadtrees: A spatio-temporal access method. In *Proceedings of the 6th ACM Symposium on Advances in Geographic Information Systems (ACM-GIS)*, pages 1–7, Bethesda, MD, 1998.

[26] P. J. Varman and R. M. Verma. An efficient multiversion access structure. *IEEE Transactions on Knowledge and Data Engineering*, 9(3):391–409, 1997.

[27] J. Vuillemin. A unifying look at data structures. *Communications of the ACM*, 23(4):229–239, 1980.