UDC 004.652.4, DOI:10.2298/CSIS101020035S

Advanced Indexing Technique for Temporal Data

Bela Stantic, Rodney Topor, Justin Terry, and Abdul Sattar

Institute for Integrated and Intelligent Systems
Griffith University,
Queensland, Australia
{b.stantic, r.topor, j.terry, a.sattar}@griffith.edu.au

Abstract. The need for efficient access and management of time dependent data in modern database applications is well recognised and researched. Existing access methods are mostly derived from the family of spatial R-tree indexing techniques. These techniques are particularly not suitable to handle data involving open ended intervals, which are common in temporal databases. This is due to overlapping between nodes and huge dead space found in the database. In this study, we describe a detailed investigation of a new approach called "Triangular Decomposition Tree" (TD-Tree). The underlying idea for the TD-Tree is to manage temporal intervals by virtual index structures relying on geometric interpretations of intervals, and a space partition method that results in an unbalanced binary tree. We demonstrate that the unbalanced binary tree can be efficiently manipulated using a virtual index. We also show that the single query algorithm can be applied uniformly to different query types without the need of dedicated query transformations. In addition to the advantages related to the usage of a single query algorithm for different query types and better space complexity, the empirical performance of the TD-tree has been found to be superior to its best known competitors.

Keywords: Temporal Databases, Access Methods, Performance Evaluation.

1. Introduction

While being ever changing, time is an important aspect of all real world phenomena. Each event bears a time attached to it, sometimes in more than one form. Time marks the starting and ending of an event and establishes the validity of data. Facts, i.e. data, valid today may have had no meaning in the past and may hold no identity in the future. Some data, on the other hand may hold historical significance or may continue to be valid up to a predefined point in time. This relationship with time adds a temporal identity to most data and, in this light, it would be hard to identify applications that do not require or would not benefit from database support for time-varying data. Most current database systems represent a single state of data and this is most commonly assumed to be its current state. Any modifications normally result in the overwriting of the data with the old data being discarded. Although commercial databases offer some capabilities to keep track of old and discarded data (e.g. the Oracle

TimeSeries cartridge, Oracle “Flash-Back”, and the Informix TimeSeries DataBlade), this is solely for the purpose of database recovery and not to retain the previous state of the data.

In the last two decades, research on temporal databases has advanced in various aspects and reported many important results, however, many challenges still remain [4], [18]. In most of the previous studies, core concepts have been established, but it is yet to be shown how they can be applied for efficiently managing time dependent data. Because temporal databases are in general append-only and usually very large in size, an efficient access method is even more important than for conventional databases. Numerous access structures were proposed, however, they have usually lacked practical credibility. Moreover most of these structures cannot handle *now-relative* data adequately. This means that these structures assume that the starting and ending points of intervals are known explicitly when recorded into the database. Obviously, this is unrealistic and this particular constraint makes these structures unsuitable for practical temporal applications.

Many multidimensional access structures have been proposed and some of them have been recommended for handling temporal data [12]. The effectiveness of the majority of these index structures has been theoretically evaluated [17]. We classify existing access methods for temporal data into four groups. The first group contains methods which represents extensions of data partitioning spatial indexing structures such as the Segment R-tree [9], 4R-tree [3], or a number of partially persistent methods [12]. In the second group, we have identified modifications of regular B^+-tree access structures, such as the Fully Persistent B^+-tree [13] and the Snapshot index [23]. The third group includes techniques based on incremental structures, such as, the Time Index [5], Time Index$^+$ [24], and the Monotonic B^+-tree [16]. Finally, in fourth group are methods which are employing the existing B^+-tree access structure by mapping of one dimensional ranges to one dimensional points, such as, MAP21 [14], mapping strategy that linearize the data like Interval Space Transformation method (IST) [7] or managing the intervals by two relational indexes the RI-tree [11], [6].

Data partitioning access methods, such as spatial indexes, use a spatial containment hierarchy that clusters data into bounding regions at the leaf level. The nearby internal nodes are then clustered into bounding regions of the parent node forming a hierarchical directory structure. These regions may not represent the entire data space and could overlap. Overlapping is a problem for data partitioning access methods because even for a simple point query it may need to examine multiple paths. When open ended *now-relative* intervals (where the ending point of the temporal interval follows the current time) are represented with widely used maximum timestamp approach a significant overlapping between nodes and dead space causes very poor performance of the index [19], [21].

We intend to propose an access method for temporal data that relies on the exploitation of the relational database systems built-in functionalities; to utilises the native Data Definition Language (DDL), Data Manipulation Lan-

guage (DML); and to use PL/SQL procedure environment within the SQL standard.

A number of access methods for temporal data that utilise the relational database systems built-in functionalities have been proposed, including: MAP21 [14], Time Index [5], Interval B-tree [2], those based on interval space transformation [7] and RI-tree [11]. We observe that the proposed access methods that rely on relational database systems built-in functionalities, such as Time Index, Interval B-tree and those based on interval space transformation IST, have either space complexity problem or are generally tailored to be efficient only for specific query types. In [11], it has been shown that the RI-tree is superior to the Window-List [15], Oracle Tile Index (T-Index) and IST-technique. In the work [10], it was extended and an algorithm for general interval relationships has been presented, but there is still a need to tailor query transformation to the specific query types. It is our intention to propose an efficient access method for temporal data with logarithmic access time and guaranteed minimum space complexity that can answer a wide range of query types with the same query algorithm.

In this paper we present and investigate the "Triangular Decomposition Tree" (TD-tree) access method to index and query temporal data. In contrast to previously proposed access methods for temporal data, this method can efficiently answer a wide range of query types, including point queries, intersection queries, and all nontrivial interval relationships queries, using a single algorithm, without dedicated query transformations.

The TD-tree is a space partitioning access method. The basic idea is to manage the temporal intervals by a virtual index structure that relies on a two-dimensional representation of intervals [20], and a triangular decomposition method. The resulting binary tree stores a bounded number of intervals at each leaf and, hence, may be unbalanced. As data is only stored in leaves, traversing the tree avoids disk accesses, and the tree depth does not affect the performance. Using the interval representation, any query type can be reduced to a spatial problem of finding those (triangular) leaves that intersect with the spatial query region. TD-trees can be implemented on top of a standard relational DBMS.

The efficiency of the TD-tree is due to the virtual internal structure, so there is no need for physical disk I/O's, query algorithms that ensures pruning, and efficient clustering of interval data. On top of the advantages related to the usage of a single query algorithm for different query types and better space complexity, the empirical performance of the TD-tree is demonstrated to be superior to its best known competitors.

The remainder of this paper is organised as follows: In the next Section, we briefly describe several temporal access methods of interest for this discussion and highlight their advantages and disadvantages. In Section 3, we define the mapping strategy and determine regions of interest. Section 4 describes the structure of TD-trees, and the key insertion and query algorithms. Section 5 contains the results and analysis of the empirical study conducted to demon-

strate the practical relevance and efficiency of the TD-tree. Finally, in Section 6, we present our conclusions.

2. Related work

In the literature, two time lines of interest have been mentioned, transaction time and valid time. The valid time line represents when a fact is valid in the modelled world and the transaction time line represents when a transaction was performed. A bitemporal database is a combination of valid and transaction time databases [4]. Because temporal databases are in general append only, they are usually very large in size, thus efficient access method is even more important in temporal databases than in conventional databases. A number of index structures for temporal data are described in the literature [17]. The existing temporal access structures, as highlighted in section 1, fit in one of four groups. We will focus on indexing structures from group four, which can be utilised by exploiting the structures and functionality of commercial RDBMSs and rely on the relational paradigm. We briefly discuss typical representatives from group three and four and highlight their advantages and disadvantages.

The Time Index [5] is an index structure for valid time intervals. It is a set of linearly ordered indexing points that is maintained by a B^+-tree. To overcomes the deficiencies of the Time Index, related to the space requirement which is O(n^2) for storing n intervals, the Time Index$^+$ has been proposed [24]. Time Index$^+$ relies on efficient storage model for partitioning logical buckets and on employing a new method to handle object versions with long and very long time intervals. The Time Index$^+$ requires less storage than the Time Index but but still requires significantly more space, even more than the spatial index methods. The disadvantage of this approach is the space required for the index, as for each point in time a bucket of pointers refers to the associated set of valid intervals. Since an interval may be registered with several points in time, This is a problem, particularly for data with many long living tuples.

The Interval B-tree (IB-tree) [2] overcomes the problems related to the extensive space usage of the Time index. It represents an implementation of the Edelsbrunner's interval tree using an augmented B^+-tree rather than a binary tree. The main memory model of the interval tree is transformed into an efficient secondary storage structure that preserves the optimal space and time complexity. The disadvantage of this approach is the complex three-fold model, which requires a dedicated structure for each level. This makes the IB-tree less attractive from the view point of time complexity.

The access method (ISP) [7] is based on interval space transformation. Since the data space may grow dynamically at the upper bound, this method is well suited for appending intervals. It indexes lists on different orders, start time, end time or duration. This access method is highly specialized with respect to the suggested mapping and can not efficiently answer more complex queries such as intersection query or point query.

The Hierarchical Triangular Mesh (HTM) is method [22] suited for indexing the sphere and especially for astronomy data. It subdivides the half surface of a sphere into four spherical triangles of similar, but not identical, shapes and sizes. Every triangle is further subdivided into four smaller triangles. Division forms a balanced tree, which is then indexed with the Quad-tree. The HTM is highly dedicated for the data that have an inherent location on the celestial sphere. The HTM has been mentioned as it has triangles as a region as our method and to highlight the differences.

The Relational Interval Tree (RI-tree) is an access method for general closed interval data, it can be created for any relational or object-relational table containing intervals [11]. Analytical and experimental evaluation of the RI-tree shows that the performance of this method is superior to the other approaches. This is achieved by introducing a virtual primary structure. Although the structure is space-oriented, the storage of intervals is object-driven so no storage space is wasted for empty regions in the data space. In [10] work was extended and an algorithm for general interval relationships has been presented but still there is a need for tailored query transformation to the specific query types. It is our intention to propose an efficient access method for temporal data that can answer wide range of query types with the same algorithm and that does not require tailored query transformation for different query types.

3. Representation of intervals and interval relationships

We assume a discrete, totally ordered time model with epochs in the range $[0..\lambda]$, for some (large) $\lambda > 0$. It is straightforward to map absolute timestamps into such a range of natural numbers, as every Unix system, for example, does. We consider only semi-open intervals $[i_s,\ i_e)$, where $0 \leq i_s\ <\ i_e \leq \lambda$. Each such interval can then be represented as a point (i_s, i_e) in two-dimensional space as shown in Fig. 1. Here, the first coordinate represents the start, S, of the interval and the second coordinate represents the end, E, of the interval. Fig. 1 shows a set of intervals A, B, C and D and their representation in two-dimensional space .

For point data there are only a few distinct query types, *e.g.*, point queries and range queries, but for interval data there are many different query types, *e.g.*. In particular, Allen described 13 distinct interval algebra (IA) relationships that may hold between pairs of intervals [1].

Each of the 13 IA relationships may now be represented as a region, line or point in our two-dimensional space, as shown in Fig. 2. When we study Allen's relationships with indexing and query evaluation in mind, we observe that they fall into two distinct groups.

Relationships between two intervals such as 'start', 'start-by', 'finish', 'finish-by', 'before', 'after', 'meet'and 'meet-by': m_i can be queried efficiently by one dimensional index structures such as B^+-tree. This is because the problem is reduced to a simple comparison of two points, start or end. However to efficiently answer queries with relationships between two intervals that require the

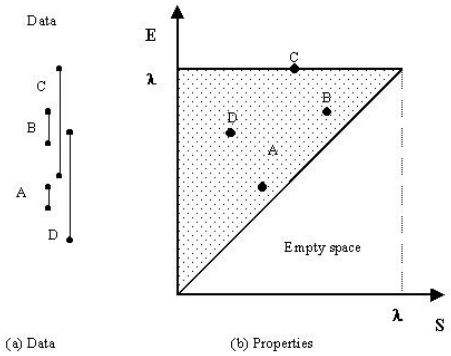

Fig. 1. Interval representation in two-dimensional space

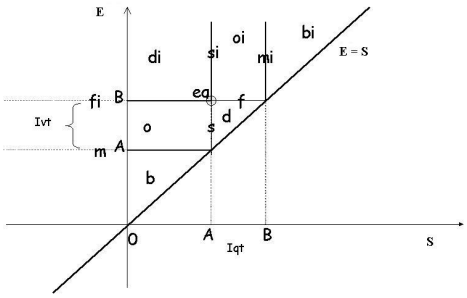

Fig. 2. Allen's 13 IA relationships between two intervals

comparison of both starting and ending points of both intervals such as: 'overlap', 'overlap-by', 'during', 'contain' and 'equal' require special access method

From now on, we focus on the problem of efficiently answering queries about relationships in the second group. We also study queries about the more general 'intersects' relationship and its special case the 'membership' relationship (or 'point' query). The basic query types we consider are queries from group two plus intersection and point query.

If the universe of intervals is $U = \{\, [u_s, u_e] \mid 0 \leq u_s < u_e \leq \lambda\}$. Due to the definition of intervals that they are semi-open u_s must be only less than u_e and can not be equal. Then Fig. 3 (a) shows a set of intervals A, B, C, D, a query point at T_0, and a query interval $I_{qt} = (T_1, T_2)$. The result of each query type above is then a two-dimensional rectangle, or point, as defined below.

– Equality Query EMQ - checks if the database contains an interval which equals the query interval:
$EMQ([i_s,\ i_e]) = \{\ [r_s,\ r_e]\ \mid r_s = i_s \wedge r_e = i_e\ \}$ is a point in two-dimensional space;

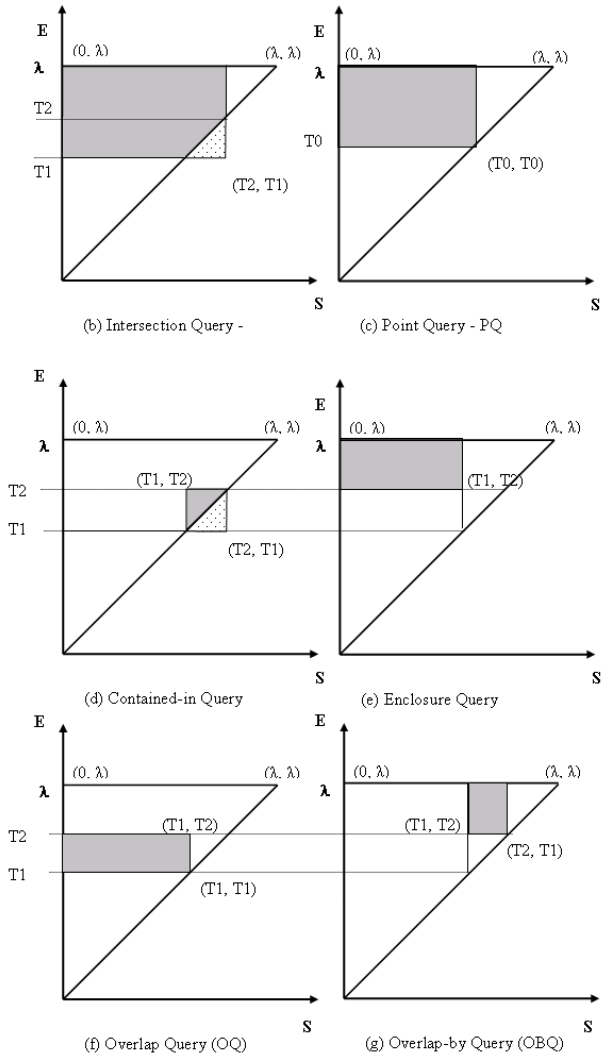

Fig. 3. Query regions

- Intersection Query (IQ), Fig. 3 (b) - finds all intervals that intersect the query interval:
  $IQ([i_s,\ i_e]) = \{\ [r_s,\ r_e]\ \mid\ (r_s < i_e) \wedge (i_s < r_e)\ \}$ is a rectangle $[(0, \lambda), (i_e, i_s)]$, in our example $i_s = T_1$ and $i_e = T_2$);
- Point Query (PQ), Fig. 3 (c) - also called timeslice query is a special case of intersection query it finds all intervals that contain the query point: $PQ(p) = \{\, [r_s, r_e] \mid r_s < T_0 < r_e \,\}$ is a special case of IQ and results in the rectangle $[(0, \lambda), (p, p)]$, in our example $p = T_0$;

- Contained-in Query (CQ), Fig. 3 (d) - finds all intervals that are contained in the query interval:
  $CQ([i_s, i_e])] = \{\, [r_s, r_e] \mid (i_s < r_s < i_e) \wedge (i_s < r_e < \ i_e) \,\}$ can be simplified to $\{\, [r_s, r_s] \mid (i_s < r_s < r_e < i_e)\}$, maps to the rectangle $[(i_s, i_e), (i_e, i_s)]$;
- Enclosure Query (EQ), Fig. 3 (e) - finds all intervals that contain the query interval.: $EQ([i_s, i_e]) = \{\, [r_s, r_e] \mid (r_s < i_s < r_e) \wedge (r_s < i_e < r_e) \,\}$ can be simplified to $\{[r_s, r_e]|(r_s < i_s < i_e < r_e)\}$, as $r_s < r_e$ and $i_s < i_e$, and results in the query box $[(0, \lambda), (i_s, i_e)]$, ,
- Overlap Query (OQ), Fig. 3 (f): $OQ([i_s, i_e]) = \{\, [r_s, r_e] \mid (i_s < r_s) \wedge (r_s < i_e < r_e) \,\}$, maps to the rectangle $[(0, i_e), (i_s, i_s)]$;
- Overlap by Query (OBQ), Fig. 3 (g): $OBQ([i_s, i_e]) = \{\, [r_s,\ r_e] \ \mid \ (r_s \ < \ i_s \ < \ r_e) \ \wedge (r_s < i_e) \,\}$, is rectangle $[(i_s, \lambda), (i_e,\ i_e)]$;

The point of this analysis is that the evaluation of every query type can now be reduced to the spatial problem of finding all data intervals that belong to the rectangle associated with that query. In particular, this means that every query type can be evaluated by a common algorithm, which is what we now study. Note, to form a rectangular query region for particular query type, the query region can extend under the line $E = S$, as for example for intersection query Figure 3 (b) and containment query Figure 3 (d). Because $0 \leq i_s < i_e \leq \lambda$ no intervals will be registered under the line $E = S$ so extending query region under the line $E = S$ to form rectangular query region will not affect the answer.

4. The Triangular Decomposition Tree (TD-tree)

The structure of our indexing method is based on the observation, that all data and query intervals of interest represented in two dimensional space lie in the isosceles, right-angle triangle with vertices at (0,0), $(0, \lambda)$ and (λ, λ), which lies above the line $E = S$. We call this triangle the *basic triangle* Figure 1. This is due to nature of interval space transformation and fact that $i_s < i_e$.

Given that our region of interest is a triangle, our main proposal is to recursively decompose the basic triangle into two smaller triangles. This triangular decomposition of the basic triangle forms a tree which we call a *TD-tree*. This tree is not balanced in general. Data intervals (points in two-dimensional space) are stored in the database in blocks associated with the leaves of the TD-tree. Figure 4 shows the second and third level of a unbalanced triangular decomposition. Arrows point to the “apex”, the right-angled vertex of the triangle, of each triangle,

In such a triangular decomposition, each triangle is uniquely identified by its *apex* position (s, e) , and its *direction* d, the direction of the arrow from the midpoint of the triangle’s hypotenuse to the apex. Note that there are eight possible directions, corresponding to the eight points of the compass, all of which are shown in Fig. 5.

Given a *Parent* (P) triangle in this decomposition, its apex and direction uniquely determine the apex and direction of each of its two subtriangles. We

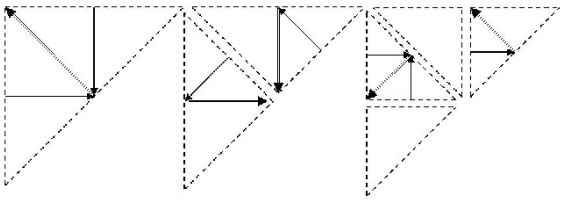

Fig. 4. Unbalanced triangular decompositions of the basic triangle.

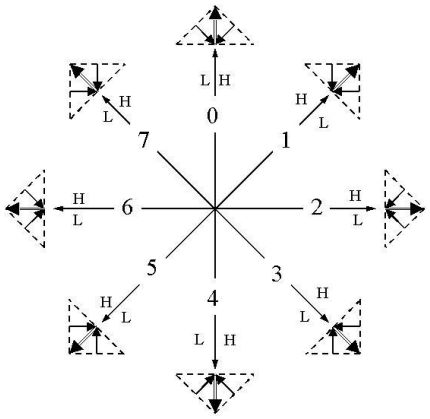

Fig. 5. Positions of low and high subtriangles

call these subtriangles *low* and *high Children* (C). Figure 5 shows, for each possible direction, which are the low and high subtriangles, and where the apexes of these two subtriangles are. Note that we number the possible directions 0 to 7 clockwise starting from direction “north”.

Input: ($P.d$: Parent direction)
Output: $L.d$: Left child direction, $R.d$: Right child direction
begin
  if $(1 <= P.d <= 4)$ **then**
    $L.d = (P.d + 5)\ mod\ 8$;
    $H.d = (P.d + 3)$;
  else
    $L.d = (P.d + 3)\ mod\ 8$;
    $H.d = (P.d + 5)\ mod\ 8$;
  end
end

Algorithm 1: Children apex directions

By observation of Fig. 5, we see that it is possible to define the apex position and direction of the subtriangles of a given triangle using the following two algorithms. Algorithm 1 computes the direction d of the lower (L) and higher (H) *Children* (C) subtriangles of a *Parent* triangle P with direction d.

Input: ($P.s$: Parent start, $P.e$: Parent end, d: direction), $length$ length
Output: $C.s$: Child start, $C.e$: Child end
begin
  if $(d = 0)$ **then**
    $C.s := P.s$; $C.e := P.e - length$
  else if $(d = 1)$ **then**
    $C.s := P.s - length/root(2)$; $C.e := P.e - length/\sqrt{2}$
  else if $(d = 2)$ **then**
    $C.s := P.s - length$; $C.e := P.e$
  else if $(d = 3)$ **then**
    $C.s := P.s - length/root(2)$; $C.e := P.e + length/\sqrt{2}$
  else if $(d = 4)$ **then**
    $C.s := P.s$; $C.e := P.e + length$
  else if $(d = 5)$ **then**
    $C.s := P.s + length/root(2)$; $C.e := P.e + length/\sqrt{2}$
  else if $(d = 6)$ **then**
    $C.s := P.s + length$; $C.e := P.e$
  else
    $C.s := P.s + length/root(2)$; $C.e := P.e - length/\sqrt{2}$
  end
end

Algorithm 2: Children apex position calculation

Algorithm 2 computes the position (s, e) of the apex of each subtriangle C of a parent triangle P at any level l. This is possible only knowing the position of the parent apex and its level.

From Fig 1 it is straight forward that the basic triangle apex is $(0, \lambda)$ and we accepted that the basic triangle has level 0. Without loss of generality, we may assume that $\lambda = 2^k$, for some $k > 0$. To find the children's apex position the adjustment length that has to be applied to the parent apex position as presented in Algorithm 2. Adjustment length depends only on level of partition l and k. It can be calculated as:

$$length = 2^k * (2^{1/2}/2)^{(l-1)} \tag{1}$$

Note that both child subtriangles of the parent triangle have the same apex position. Position of the child C apex (s, e) will be calculated depending to the direction d of the parent P apex using the Algorithm 2.
Note that the level of the subtriangles of a triangle are one more than the level of the triangle. Note also that the resulting tree need not be balanced. In an unbalanced tree, different leaves may be at different levels. The shape of a tree depends on the distribution and density of data intervals.

Because we can identify the apex and direction of every node of a TD-tree, starting from the basic triangle, using the two algorithms, *we do not need to store the internal tree nodes*. Thus, a TD-tree is a virtual tree. All we need to store is the value λ and a reference to the root node.

The actual data intervals, together with information about the intervals, are stored in a table indexed by a leaf identifier. The tree is organised so that at most b data intervals are stored with each leaf, for some integer blocking factor $b > 0$. A node identifier is a binary string, stored as a (binary) integer, constructed as follows. The identifier of the base triangle or tree root is 1. If a node has identifier ϕ, the lower and upper children of the node have identifiers $\phi 0$ and $\phi 1$ respectively. The length of the identifier is thus one greater than the depth of the node.

Information about leaf nodes themselves are stored in a separate directory, containing an identifier and number of records per leaf. The root node stores the blocking factor b and current maximum depth of the tree l.

4.1. Insertion algorithm

Insertion of data interval into a TD-tree is performed according to Algorithm 3. We first descend the tree from the root to the virtual leaf at maximum tree depth containing the interval. This is done arithmetically, without disk access, by repeatedly selecting the lower or upper child of each node depending on the value of the interval.

The leaf found is called "virtual" because that branch of the tree may have length less than the maximum depth. For example, the upper child of the root node in Fig. 6 labelled `'g'` is a leaf on a path of length 2, whereas the tree has maximum depth 7, as it can be seen in Table 1.

Input: (object_for_insertion: OBJ, Directory: D, blocking_factor: b,
max_population, max_depth)
begin
  Find maxregion at max_depth where OBJ would belong;
  target_region = region in D with longest number of bits in common
  left to right with the maxregion;
  Find target_region population;
  if *(*$target\_regionpopulation > max\_population$) **then**
    Increment the population in D of target_region;
    Update region with target_region;
  else
    perform Split;
  end
end

Algorithm 3: Insertion

Given the sample decomposition from Fig. 6, the directory would be as shown in Table 1. This table shows the label, identifier (in both binary and decimal) and identifier extension (in both binary and decimal) for each leaf of the

tree. The identifier extension is the unique extension of the leaf identifier with zeros so that it is of length l, where l is the maximum depth of the tree. Values for binary identifier and both binary and decimal identifier extension are not stored as they can be calculated from decimal identifier and max depth of the tree 'l'.

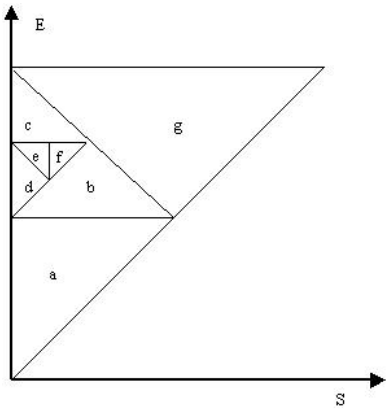

Fig. 6. Running example regular decomposition

Table 1. Directory for sample tree

Label	Identifier (binary)	Identifier (decimal)	Extension (binary)
a	100	4	1000000
b	1010	10	1010000
c	10111	23	1011100
d	101100	44	1011000
e	1011010	90	1011010
f	1011011	91	1011011
g	11	3	1100000

We attempt to insert the data interval into the actual leaf that is an ancestor of the virtual leaf found by the above traversal.

If the identifier of the virtual leaf w containing the interval is z, then the identifier of the actual leaf v that is ancestor of w is given by the longest identifier in the directory that is a prefix of z. For example, if the identifier of the virtual leaf containing the interval is 1010010, then the identifier of the actual leaf in which the interval should be stored is 1010.

Considering the sample from Figure 6 and Table 1, let for example an interval, according to the start and end points, would belong to region on max depth '1010010'. This max depth region '1010010' at the maximum depth doesn't does not exist so it is required to locate the actual region to store the interval in. That region is given by the longest identifier in the directory that is a prefix of max depth region '1010010' and in our case it is the region '1010'. Having found the region which to store the interval, we simply update leaf identifier in table with that identifier and in directory increment number of records that region holds by one.

To ensure efficient retrieval, we store at most b data intervals with each leaf. If a leaf already has that many intervals, we construct the two children of the leaf, replace the parent with the two children in the directory, distribute the current (and new) intervals between the two children as appropriate, and repeat this process recursively if all intervals go into the same child. If this operation increases the maximum tree depth, we record the new maximum depth. This split is performed according to Algorithm 4.

Input: (SR: Split Region, D: Directory, blocking_factor, max_pop, max_depth)
begin
  while *not both child regions population* $<$ *max_pop* **do**
    divide data of SR into children; current_depth = SR depth + 1;
    if *current_depth* $>$ *max_depth* **then**
      max_depth = max_depth + 1;
    end
    if *child region is POINT* **then**
      Exit;
    end
  end
end

Algorithm 4: Split

It is possible that all intervals in a region that has to be split are located within one newly created smaller region, which will cause a further split. Splits will be performed until intervals can be distributed between two child regions or the maximum split was reached (region represents a point). If maximum split was reached the population of the region is allowed to grow beyond blocking factor, which means that multiple blocks may associate with one region.

4.2. Query algorithm

Following the analysis of Section 3, we can assume that every query corresponds to a rectangular region of the two-dimensional interval space, defined by the top-left and bottom-right corners of this region. The task of the query evaluation is to find all data intervals that occur within this query region. The particular region chosen depends on whether we are performing an intersection query, an overlaps query, a contains query, and so on, but in each case the query evaluation algorithm is identical, an important property of our approach.

Query evaluation itself proceeds in two phases Algorithm 5. In the first phase we find those TD-tree nodes which are contained entirely within the query region and those TD-tree leaves which overlap (but are not contained in) the query region. This phase accesses the disk to retrieve nodes from the directory and to retrieve data intervals from overlapping leaves. In the second phase, we return the intervals in the first set of nodes, and scan the intervals in the second set of leaves for those that occur in the query region. This second phase requires no additional disk access.

The first phase may be implemented as follows. It takes as input the query region `Q` and the directory `D`. It returns the set of data intervals that occur within `Q`.

```
Input: (D: Directory, Q: Query region)
Output: A1: Containing leaves, A2: Overlapping leaves
begin
    Add all nonempty leaves in D to a LIST;
    let length L be 1;
    while LIST is not empty do
        let F be the first leaf in LIST;
        let R be the ancestor of F whose identifier consists
        of the first L digits of F's identifier;
        if R is contained within Q then
            add all leaves in LIST with the same prefix as R to A1 and
            remove them from LIST;
            set L to 1;
        else if R is disjoint from Q then
            remove all leaves in LIST with the same prefix as R from LIST;
            set L to 1;
        else if R equals F then
            add R to A2 and remove it from LIST;
        else
            increment L;
        end
    end
end
```

Algorithm 5: Query algorithm

This phase terminates with `A1` containing the set of nodes whose descendent leaves are contained entirely within `Q`, and `A2` containing the set of leaves which overlap `Q`. By testing whether the ancestor `R` of `F` is contained within `Q`, we can select all leaves under `R` in one operation. This property of our algorithm significantly reduces the number of disk accesses and improves its overall performance. To test whether a triangle is contained within a rectangle or whether a triangle intersects a rectangle are straightforward geometric operations based on the vertices of the two operations.

On Figure 7 and 8 virtual region '100' (dark grey) lies completely outside the region for point query at T_0, and for that reason all leaf regions that are children of virtual region '100' are excluded from the answer (10001, 100000, 100001,

10010, 100110, 100111). This bulk exclusion improves the query efficiency. On Figure 7 and 8 we show total exclusion and total inclusion with the dark and light grey respectively.

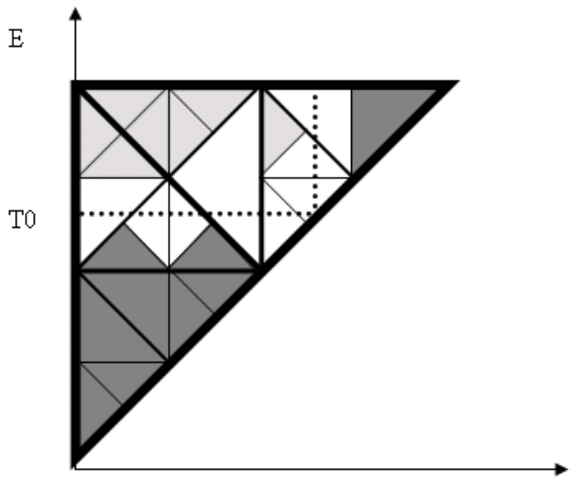

Fig. 7. Point query on transformed region

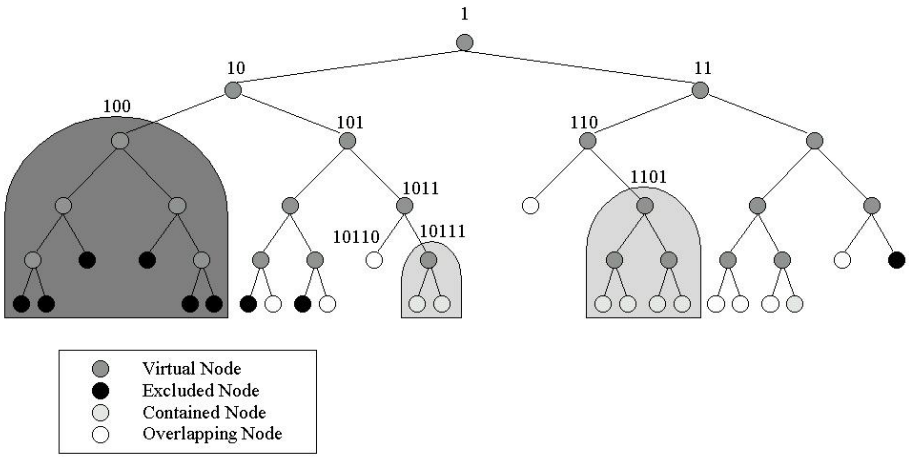

Fig. 8. Unbalanced Binary tree

In the second phase, it suffices to return all data intervals in all descendent leaves of nodes of O1 and to scan all data intervals in all leaves of O2, returning those intervals that occur within Q. This latter test is a simple arithmetic comparison. No additional disk accesses are required in this phase.

Deletion Algorithm removes regions from the directory that contain zero objects due to the decrement of population. Also, this algorithm merges two children into parent region if sum of population of both children falls under the one third of the blocking factor.

Input: D: Directory, object_for_deletion, blocking_factor)
begin
  delete_region = region where object_for_deletion belongs; Delete object_for_deletion; Decrement the population of delete_region; **if** *combined population of delete region and its sibling* $<$ *blocking_factor/3* **then**
    merge two children into parent regions;
  end
end

Algorithm 6: Deletion

Update can be seen as delete and insert and therefore is handled by *Deletion* and *Insert* algorithms.

It is straightforward to show that the query evaluation algorithm of the previous subsection returns all data intervals that occur within the query region and only these. In the interests of brevity we omit the details. It is perhaps more important to note the following complexity result.

Proposition 1: An intersection query on a TD-tree with blocking factor b and n data intervals with answer size of phase one of the query algorithm a having the directory size m, performs $O(m/b + log_b n + a/b)$ disk accesses.

Proof Because the TD-tree has no internal nodes, to find the answer size of phase one of the query algorithm *a* having the directory size *m* and block size *b*, O(m/b) I/O complexity is performed. Additionally, scanning the index organised table has I/O complexity of O(log_b n) and reporting the total of *a* results requires O(a/b) operations. Therefore, the I/O complexity for TD-tree is $O(m/b + log_b n + a/b)$.

Note that in worst case scenario, due to the secondary filtering of phase two of the query algorithm, a can be equal to n. Similarly to any query algorithm that has secondary filtering in worst case scenario I/O complexity can be basically $O(n/b)$. However, in our experiments we could not replicate such scenario.

In Table 2, we present the complexity cost for TD-tree and several other access methods of interest. a denotes the size of the point query while n represents the number of interval objects. For RI-tree h is a virtual backbone tree that corresponds to the current expansion and granularity of the data space but does not depend on n. Because there is no internal nodes in TD-tree and the size of the directory is significantly smaller than the data table, the complexity cost for TD-tree practically depends only on answer size. Our experimental results support this claim.

Table 2. Performance analysis of access methods of interest

$Access\ Method$	$Space\ Usage$	$Point\ Query$
$R-Trees$	$O(n/b)$	$O(n/b)$
$Time\ Index$	$O(n^2/b)$	$O(log_b n + a/b)$
$Bitemp.R-Tree$	$O(n/b)$	$O(log_b n + a/b)$
$RI-tree$	$O(n/b)$	$O(hlog_b n + a/b)$
$TD-tree$	$O(m/b)$	$O(m/b + log_b n + a/b)$

5. Experimental evaluation

To show the practical relevance of our approach, we performed an extensive experimental evaluation of the TD-tree and compared it to the RI-tree [11].

The RI-tree was chosen, since it provides the same practically important properties as our approach. It is easy to implement and integrate, it uses standard RDBMS methods which provides scalability, update-ability, concurrency control and space efficiency. Furthermore it has been proven [11] that the RI-tree is superior to the Window-List [15], Oracle Tile Index (T-Index) and IST-technique [7] so performance results of the TD-tree can be transferred to these indexing techniques. We could not compare our TD-tree with improved implementation of RI-tree [6] as it indexes Interval-and-Value tuples together while our method only index intervals.

All experimental results presented in this section are computed on eight 850MHZ CPU - SUN UltraSparc II processor machine, running Oracle 10.2.0 RDBMS, with a database block size of 8K and SGA (System Global Area) of 500MB. At the time of testing database server did not have any other significant load. We used Oracle built-in methods for statistics collection, analytic SQL functions and the PL/SQL procedural runtime environment.

5.1. Data sets

In order to simulate different real applications scenarios we used different data distributions. The start position of the intervals was always uniformly distributed on the interval domain, while the duration was varied. Following data distributions have been considered:

- Uniformly distributed start and uniform distributed length within the range [1, 10000] with 20% of uniformly distributed *now-relative* data.
- Uniformly distributed start and exponentially distributed length according to the exponential distribution function $y = e^{-0.00041*x}$ with 20% of uniformly distributed *now-relative* data.

Uniform distribution of interval start, appearance of *now-relative* data and exponential distribution of the duration reflects most real world applications

where short intervals are more likely to occur than long intervals. We used maximum timestamp approach to represent current time. Furthermore, in real world applications there is usually a upper bound for the interval duration and in our case we have chosen 10,000 (days) for the upper bound, not considering *now-relative* data, which are represented with maximum timestamp approach.

All data set distributions had separate relations with different number of tuples, 250,000, 500,000 and 1,000,000.

5.2. Query sets

In our experiment we tested performance on intersection queries and particularly on point query as its specific case. Because of the nature of our query algorithm, by comparing the data region with the rectangular query region, as has been shown in subsection 3, results for performance evaluation apply to the other query types.

The point query that timeslices the timeline at the current time was used to determine how access method performs with *now-relative* data. The point query that timeslices the time line at the current time is considered to be the most important because most often we will ask queries about the current state of reality.

5.3. Update sets

Most often updates in Temporal databases happen when facts cease to be valid (in valid time databases) or tuple is logically deleted (in transaction time databases). In both cases ending time of interval that contain sematic for 'now' (*now-relative* data) is replaced with the current time. We tested performance of our TD-tree on updates of randomly selected *now-relative* interval data of 100 tuples. As explained in update algorithm to perform update it is required to perform delete from the previous region and insert interval into the new region.

5.4. Experiments

The same data set is used both for RI-tree and TD-tree testing experiment. The initial relations with structure Employment(ID, Name, Position, Start, End) were replicated and altered accordingly to suit each particular method.

Relations for testing the performance of the RI-Tree, were altered with column *node*, which is calculated for every row of data by algorithm as explained in paper [10]. Two B^+-tree composite indexes have been created *LowerIndex* (node, Start) and *UpperIndex* (node, End). A point query is performed by calling the dedicated procedure that collects leftnodes and rightnodes and then performs the transformed SQL statement as instructed in [11].

Relations for testing the performance of the TD-tree were altered with column *Region*, which is calculated according the algorithm as explained in Subsection 4.1. The root node, which contains information for λ, blocking-factor,

adjustment date and maximum depth of the tree we stored as one tuple relation. Because TD-tree has only leaf nodes it can be organised as a list and stored in directory tables. To ensure that the population of a region corresponds to one block, so it can be be efficiently retrieved, we introduced a blocking factor. We built relation Employment as index organised table using *Region* and *ID* as a primary key.

5.5. Results

To compare the space requirements for RI-tree and TD-tree, we considered tables with different number of rows. We generated tables with 250,000 rows, 500,000 rows and 1,000,000 tuples. All tables are altered to suit the particular approach and all required primary and secondary indexes are created. In Fig. 9 we show the space requirement for the TD-tree and RI-tree. Results represent the sum of used space for table, primary/secondary indexes and for the TD-tree we also added required space for the directory table.

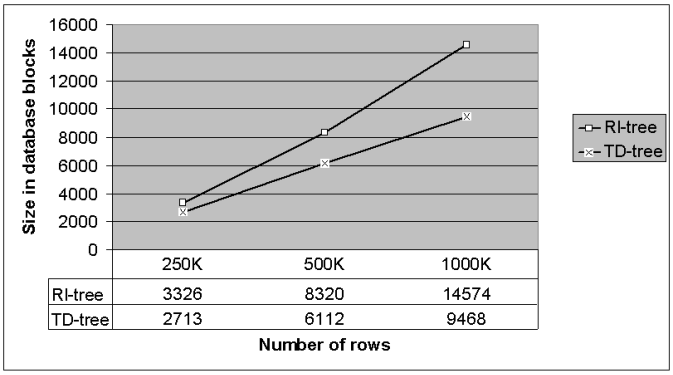

Fig. 9. Comparison of the space usage (Table plus indexes)

To measure the query performance we used a data set of one million tuples. Results shown in Fig. 10 are for the point query with uniformly distributed start and exponentially distributed length with 20% of *now-relative* data. Results represent disk I/O and average CPU usage for different points on timeline which contain different answer sizes. We performed tests with all data distributions mentioned in subsection 5.1, but testing resulted in similar qualitative results as those presented here.

The Theory of Indexability [8] identifies I/O complexity cost, measured by the *number of disk accesses*, as one of the most important factors for measuring query performance. Other measures of importance such as *CPU usage* and query response time are also used in conjunction with the number of disk accesses to assess the performance of the query processes.

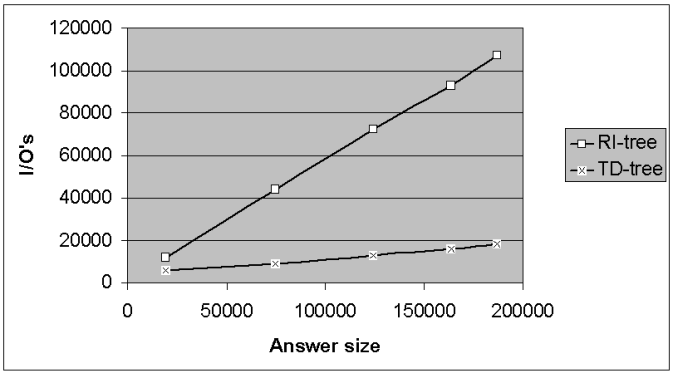

Fig. 10. Physical disk I/O as a factor of answer size

Table 3. Average number of answers per one Physical disk I/O

$Answer\ Size$	$TD-tree\ DiskI/O$	$Answers/\ DiskI/O$	$RI-tree\ DiskI/O$	$Answers/\ DiskI/O$
19365	6102	3.17	11902	1.63
74727	8952	8.35	43891	1.70
124280	12958	9.59	72426	1.72
163530	16012	10.21	92776	1.76
186795	18054	10.35	107068	1.74

For the TD-tree the number of leaf regions accessed to answer the query is simply the number of regions returned in the *Primary* filter. *Secondary* filtering only does pruning so it does not require any additional disk access, it only adds CPU usage. When the answer is smaller, interval objects pruned with the secondary filter effect the performance of the TD-tree and number of answers per one physical disk I/O is relatively smaller. In Table 3 we can see that TD-tree even for a small answer size has better factor of answers per physical disk reads. For the RI-tree the number of answers per physical disk read is not dependent on query load, however for the TD-tree, due to the secondary filter features, the number of answers per physical read is dependent on query load and reaches the best performance on larger query loads.

Beside the queries mentioned in subsection 5.2, we tested applicability and performance of the TD-tree on several other query types, such as during, contain, and even before and after. These results will be mentioned and analysed in the next subsection.

The TD-tree performs well on *now-relative* data using *MAX* approach to represent current time, because the area where these intervals are stored can be divided as often as required as shown in Figure 12. If we represent interval objects that belong to the particular RI-tree nodes in two dimensional space, it can

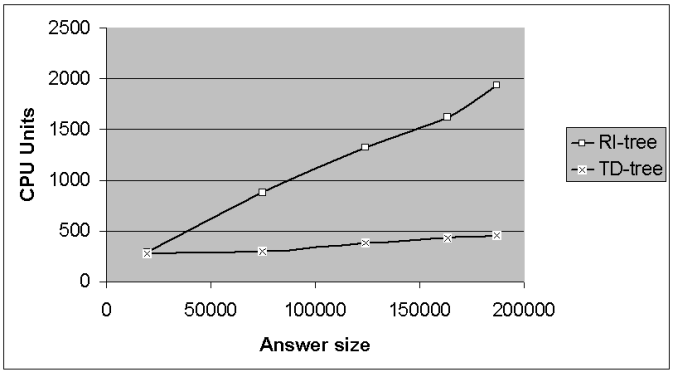

Fig. 11. CPU usage as a factor of answer size

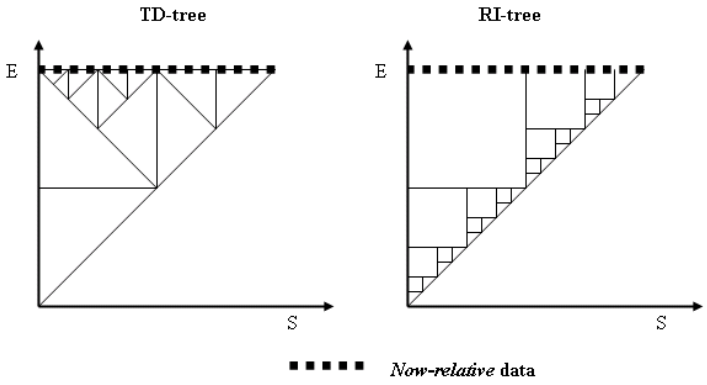

Fig. 12. Now-relative data in TD-tree and RI-tree

be seen that all *now-relative* data will be located in the root node (biggest rectangle in Figure 12) or on the right edge of the RI-tree. Knowing the importance of *now-relative* data in temporal databases this is a significant constraint.

5.6. Comparative analysis

When making performance measurements of index structures, it is important to not only consider response time but also other parameters such as space requirements, clustering, CPU usage, updates, and locking. In our analysis, we have concentrated on space requirements, physical disk reads, CPU usage and clustering of data. Because both the RI-tree and TD-tree rely on the relational paradigm, updates and locking are handled well by the RDBMS itself.

The TD-tree requires only one virtual index structure, which means only leaf nodes have to be stored. The list of leaf nodes are stored in the directory table and its size is very small comparing to the table itself. In our experiment the TD-tree directory for one million interval objects required only 26 data blocks. In addition to directory table there is a need for extra space considering that table is index organised by region, which is comparable with the primary index of RI-tree method.

The RI-tree requires two composite index structures *lowerIndex* and the *upperIndex*. One composite index is on *node* and $Start$ - start of the interval, and another composite index is on *node* and End - end of the interval. The size of the indexes depend on the number of interval objects and in our experiment one million interval objects required 6708 data blocks (3354 each index), which is significantly bigger than the 26 blocks required for the TD-tree directory. For this reason, the total number of blocks required for table and index structures for TD-tree is much smaller than the number of total blocks required for RI-tree. This difference increases with increasing number of interval objects, as shown in Fig. 9.

The TD-tree enables efficient usage of clustering of the data by one dimension, i.e region, as every region associate with block size. Clustering data improves the query performance and reduces the number of physical I/O, as shown in Table 3, clustering ensures higher number of answers per physical disk I/O. In contrast, the RI-tree can not efficiently use clustering of data as it has to decide which dimension to use start or end. If it is clustered by *node* it will not result in similar improvements, as in RI-tree *node* are fixed size and are too large to provide effective clustering.

In Figure 10 it can be seen that the virtual structure of the TD-tree, clustering of data and the query algorithm significantly reduces the physical disk I/O reads. This is particularly the case when the answer size is bigger due to the good clustering, which is achieved by dividing the regions as often as needed.

The TD-tree performs as good on *now-relative* data as on any other data. We could not notice any difference in the number of physical disk I/O's and CPU usage for a point query, which timeslices time line at the current time and at any other time point on the time line. This is because the area where these intervals are stored can be divided as often as required.

Our experimental testing shows that the TD-tree query algorithm performs well on other query types such as: during, contain, before and after. This is because it compares the data region with the query region and uses the same algorithm. It is important to mention that the RI-tree needs dedicated query transformation for specific query type. Despite the TD-tree performing well on before and after query types it has worse performance in comparison with the straight forward usage of one dimensional indexes, as was anticipated and highlighted in section 3. The TD-tree does not perform well on query types such as start and finish because the query region is a line. However, these query types can be efficiently answered with one dimensional index, which is also highlighted in section 3.

6. Conclusions

We described a new approach that is demonstratively better than existing approaches for handling temporal, and more generally, interval data. More specifically, in this study, we:

- Presented a two-dimensional interval space representation of intervals and interval relationships to reduce all interval relationship problems to simpler spatial intersection problems;
- Showed that a wide range of interval query types can be reduced to an intersection of data with a rectangular region, so one algorithm can be applied uniformly;
- Proposed the "Triangular Decomposition Tree" (TD-tree) and associated algorithms that can efficiently answer a wide range of query types including point or timeslice queries;
- Experimentally evaluated the TD-tree by comparing its performance with RI-tree, and demonstrated its overall superior performance.

The TD-tree is a unique access method as it uses tree structures, and at the same time has some characteristics of hashing approaches due to it only stores data in leaf nodes. In contrast to hashing methods that do not perform well on range queries, the TD-trees can efficiently answer a wide range of different query types. It is important to mention that the management of the virtual structures is done automatically by using database triggers, which fire on insert, calculates, updates the region of the record and also increments the region in the directory. If required, it also initiates and performs splits. Similarly, database triggers fire on updates/deletes and performs actions in line with Deletion and Insert Algorithms.

As a wide range of query regions of interest can be reduced to rectangles, it is possible to answer such queries using a single algorithm without requiring any query transformation. This itself, and the fact that the TD-tree can be incorporated within commercial RDBMS, makes the TD-trees superior to other methods proposed for temporal data.

References

1. J. Allen. Maintaining knowledge about temporal intervals. *Communications of the ACM*, 26(11):832–843, 1983.
2. C. Ang and K. Tan. The Interval B-tree. *Information Processing Letters*, 53(2):85–89, 1994.
3. R. Bliujute and et al. Light-Weight Indexing of General Bitemporal Data. In *Statistical and Scientific Database Management*, pages 125–138, 2000.
4. C. Date, H. Darwen, and N. Lorentzos. *Temporal Data and the Relational Model*. Morgan Kaufmann, 2002.
5. R. Elmasri, G. Wuu, and Y. Kim. The time index: An access structure for temporal data. *Proc. 16th Conf. Very Large Databases*, pages 1–12, 1990.

6. J. Enderle, N. Schneider, and T. Seidl. Efficiently processing queries on interval-and-value tuples in relational databases. In *VLDB '05: Proceedings of the 31st international conference on Very large data bases*, pages 385–396. VLDB Endowment, 2005.
7. C. H. Goh and et al. Indexing temporal data using existing b+-trees. *Data and Knowledge Engineering*, (18):147–165, 1996.
8. J. Hellerstein, E. Koutsupias, and C. Papadimitriou. On the Analysis of Indexing Schemes. *16th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, 1997.
9. C. Kolovson and M. Stonebraker. Segment indexes: Dynamic indexing techniques for multi-dimensional interval data. *Proc. ACM SIGMOD*, pages 138–147, 1991.
10. H.-P. Kriegel, M. Potke, and T. Seidl. Object-relational indexing for general interval relationships. In *Proc. 7th Intl Symposium on Spatial and Temporal Databases (SSTD01)*, 2001.
11. H.-P. Kriegel, M. Ptke, and T. Seidl. Managing intervals efficiently in object-relational databases. *Proceedings of the 26th International Conference on Very Large Databases*, pages 407–418, 2000.
12. A. Kumar, V. Tsotras, and C. Faloutsos. Designing Access Methods for Bitemporal Databases. *IEEE Transactions on Knowledge and Data Engineering (TKDE'98)*, 10(1):1–20, 1998.
13. S. Lanka and E. Mays. Fully persistent b + trees. *Proc. ACM SIGMOD Conf. on the Management of Data*, pages 426–435, 1991.
14. M. A. Nascimento and M. H. Dunham. Indexing Valid Time Databases via B^+-Tree. *IEEE Transactions on Knowledge and Data Engineering*, 11(6):929–947, 1999.
15. S. Ramaswamy. Efficient Indexing for Constraint and Temporal Databases. In *Proceedings of the 6th International Conference on Database Theory*, pages 419–431, 1997.
16. R.Elmasri, G. Wuu, and V.Kouramajian. The Time Index and the Monotonic B^+-Tree. *In A. Tansel et.al., editors Temporal Databases: Theory Design and Implementation*, pages 433–456, 1993.
17. B. Salzberg and V. J. Tsotras. Comparison of Access Methods for Time Evolving Data. *ACM Computiong Surveys*, 31(1), 1999.
18. R. T. Snodgrass. *Developing Time-Oriented Database Applications in SQL*. Morgan Kaufmann, 2000.
19. B. Stantic, S. Khanna, and J. Thornton. An Efficient Method for Indexing Now-relative Bitemporal data. *In Proceeding of the 15th Australasian Database conference (ADC2004), Denidin, New Zealand*, 26(2):113–122, 2004.
20. B. Stantic, J. Terry, R. Topor, and A. Sattar. Indexing Temporal Data with Virtual Structure. In *Advances in Databases and Information Systems - ADBIS2010*, pages 591–594, 2010.
21. B. Stantic, J. Thornton, and A. Sattar. A Novel Approach to Model NOW in Temporal Databases. *In Proceeding of the 10th International Symposium on Temporal Representation and Reasoning (TIME-ICTL 2003), Cairns*, pages 174–181, 2003.
22. A. S. Szalay, J. Gray, G. Fekete, P. Z. Kunszt, P. Kukol, and A. Thakar. Indexing the Sphere with the Hierarchical Triangular Mesh. *CoRR*, abs/cs/0701164, 2007.
23. V. J. Tsotras, B. Gopinath, and G. Hart. Efficient management of time-evolving databases. *IEEE Trans. Knowledge and Data Eng*, 7(4):591–608, 1995.
24. V.Kouramajian and et.al. The time index+: An incremental access structure for temporal databases. *In Proceedings of the Third Interbnational Conference on Knowledge and Data Engineering (CIKM'94)*, pages 232–242, 1994.

Dr Bela Stantic is Senior Lecturer at the School of Information and Communication Technology, Griffith University, and member of the Institute for Integrated and Intelligent Systems (IIIS). He is also a Senior Researcher at National ICT Australia (NICTA) Queensland Research Lab (QR). He has been an academic staff member at Griffith University since 2001. His research interests include efficient management of complex data structures, temporal and spatio-temporal databases, bioinformatics, and database systems.

Rodney Topor has been a Professor of Computing Science in the School of Information and Communication Technology at Griffith University since July 1991. Prior to that he worked in the Computer Science departments at The University of Melbourne and Monash University. He has served as Head of School and in other management roles at Griffith. His research interests include programming methodology, programming languages, database systems, knowledge representation and Web application development.

Dr Justin Terry is member of the Institute for Integrated and Intelligent Systems (IIIS). He completed his PhD in 2010 and his topic was on indexing multi-dimensional data. His research interest includes efficient management of high-dimensional data.

Professor Abdul Sattar is the founding Director of the Institute for Integrated and Intelligent Systems and a Professor of Computer Science and Artificial Intelligence at Griffith University. He is also a Research Leader at National ICT Australia (NICTA) Queensland Research Lab (QRL). He has been an academic staff member at Griffith University since February 1992 within the School of Information and Communication Technology. His research interests include knowledge representation and reasoning, constraint satisfaction, intelligent scheduling, rational agents, propositional satisfiability, temporal reasoning, temporal databases, and bioinformatics.

Accepted: October 20, 2010.