A Triangular Decomposition Access Method for Temporal Data - TD-tree

Bela Stantic **Rodney Topor** **Justin Terry** **Abdul Sattar**

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

Abstract

In this study, we investigate and present a new index structure, Triangular Decomposition Tree (TD-tree), which can efficiently store and query temporal data in modern database applications. TD-tree is based on spatial representation of interval data and a recursive triangular decomposition of this space. A bounded number of intervals are stored in each leaf of the tree, which hence may be unbalanced. We describe the algorithms used with this structure. A single query algorithm can be applied uniformly to different query types without the need of dedicated query transformation. 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 is demonstrated to be superior to its best known competitors. Also, presented concept can be extended to more dimensions and therefore applied to efficiently manage spatio-temporal data.

Keywords: Temporal Databases, Access Method

1 Introduction

A temporal database is one that supports some aspect of time distinct from user-defined time. Over the last two decades interest in the field of temporal databases has increased significantly, with contributions from many researchers (Date et al. 2002), (Snodgrass 2000). 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 (Date et al. 2002). 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 (Dyreson et al. 1995). Many multidimensional access structures have been proposed and some of them have been recommended for handling temporal data (Kumar et al. 1998). The effectiveness of the majority of these index structures has been theoretically evaluated (Salzberg & Tsotras 1999). Proposed access methods for temporal data can be classified on the techniques used as follows:

- Extensions of data partitioning spatial indexing structures (Guttman 1984) such as the Segment R-tree (Kolovson & Stonebraker 1991), 4R-tree (Bliujute & et al. 2000), or a number of partially persistent methods (Kumar et al. 1998),
- Modifications of regular B^+-tree access structures such as the Fully Persistent B^+-tree (Lanka & Mays 1991) and the Snapshot index (Tsotras et al. 1995),
- Techniques based on incremental structures such as the Time Index (Elmasri et al. 1990), Time Index$^+$ (V.Kouramajian & et.al 1994) and the Monotonic B^+-tree (R.Elmasri et al. 1993),
- Employing the existing B^+-tree access structure by mapping of one dimensional ranges to one dimensional points, as is the case in MAP21 (Nascimento & Dunham 1999), mapping strategy that linearize the data like Interval Space Transformation method (IST) (Goh & et al. 1996) or managing the intervals by two relational indexes the RI-tree (Kriegel et al. 2000), (Enderle et al. 2005).

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 region 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 (Stantic et al. 2004), (Stantic et al. 2003).

We intend to propose access method for temporal data that relies on the exploitation of the relational database systems built-in functionalities and to utilises the native Data Definition Language (DDL), Data Manipulation Language (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 (Nascimento & Dunham 1999), Time Index (Elmasri et al. 1990), Interval B-tree (Ang & Tan 1994), those based on interval space transformation (Goh & et al. 1996) and RI-tree (Kriegel et al. 2000).

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 (Kriegel et al. 2000) it has been shown that the RI-tree is superior to the Window-List (Ramaswamy 1997), Oracle Tile Index (T-Index) and IST-technique. In (Kriegel et al. 2001) work 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 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 (Stantic et al. 2010) 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 tree depth hence does not affect 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 algorithm 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.

2 Related work

A number of index structures for temporal data are described in the literature (Salzberg & Tsotras 1999). The existing temporal access structures, as highligthed in section 1 fit in one of the following groups: (1) Extensions of data partitioning methods spatial indexing structures; (2) Modifications of regular B^+-tree access structures; (3) Techniques based on employing the existing B^+-tree access structure on mapping of one dimensional ranges to one dimensional point.

We will focus on indexing structures from group (3), 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 (3) and highlight their advantages and disadvantages.

The Time Index (Elmasri et al. 1990) is an index structure for valid time intervals. It is a set of linearly ordered indexing points that is maintained by a B^+-tree. 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, the space requirement is $O(n^2)$ for storing n intervals. This is a problem, particularly for data with many long living tuples.

The Interval B-tree (IB-tree) (Ang & Tan 1994) overcomes the problems related to the extensive space usage of the Time index. It represents an implementation of the Edelsbrunners 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) (Goh & et al. 1996) 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 created 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 (Szalay et al. 2007) 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 (Kriegel et al. 2000). 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 (Kriegel et al. 2001) 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

Symbol	*Explanation*
λ	Arbitrary maximum value
i_s	Interval Start
i_e	Interval End
S	Vertical axis of of Cartesian space
E	Horizontal axis of Cartesian space
l	maximum depth of the tree
P	Parent triangle
C	Child subtriangle
d	Direction of the triangle apex
b	Blocking factor

Table 1: Parameters and symbols definitions

the interval. Fig. 1 shows a set of intervals A, B, C and D and their representation in two-dimensional space .

In order to help the reader to follow our discussion in Table 1 we provide a legend of symbols used with an explanation.

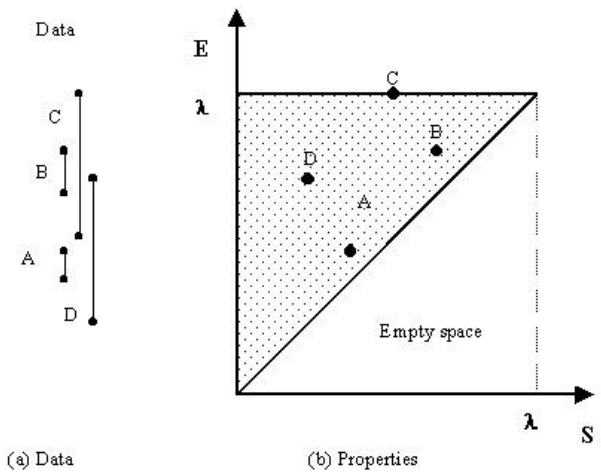

Figure 1: Interval representation in two-dimensional space

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 (Allen 1983), which we now consider. Let $I_{vt} = [i_s, i_e)$ be a stored interval and $I_{qt} = [i_s, i_e]$ be a query interval with exactly the same starting and ending points (note query interval is closed on both sides in contrast to data interval which is semi-open according to the definition).

Each of the 13 IA relationships may now be represented as a region, line or point in our two-dimensional space. 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 that represent simple comparison of the same starting or ending points. Such relationships include those where both intervals start or end with the same epoch, *e.g.*, the relationships 'start': s, 'start-by': s_i, 'finish': f and 'finish-by': f_i, and those where one interval starts with the same epoch that the other ends, *e.g.*, 'before': b, 'after': b_i, 'meet': m and 'meet-by': m_i. Each of these eight relationships 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.
- Relationships between two intervals that require the comparison of both start and end points of both intervals. These five relationships are 'overlap': o, 'overlap-by': o_i, 'during': d, 'contain': d_i and 'equal': eq. To efficiently answer these queries special access method is required.

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.

The universe of intervals is: $U = \{\, [u_s, u_e] \mid 0 \leq u_s < u_e \leq \lambda\}$. Please note that due to the definition of intervals semi-open u_s must be only less than u_e and can not be equal. Then Fig. 2 (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;
- Intersection Query (IQ), Fig. 2 (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. 2 (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 is T_0;
- Contained-in Query (CQ), Fig. 2 (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. 2 (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 $[(0, \lambda), (i_s, i_e)]$, ,

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 2 (b) and containment query Figure 2 (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.

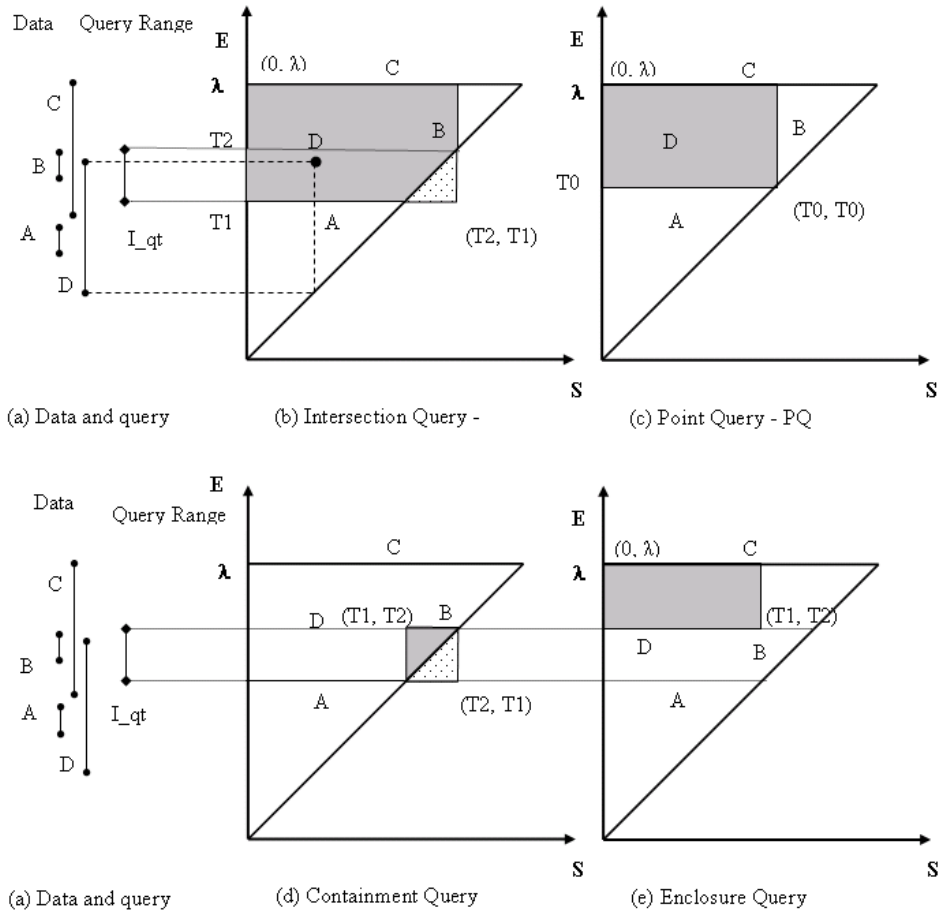

Figure 2: Query regions

4 The triangular decomposition tree

4.1 Basic data structure

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 our two-dimensional space) are stored in the database in blocks associated with the leaves of the TD-tree. Figure 3 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. 4.

Given a triangle in this decomposition, its apex and direction uniquely determine the apex and direction of each of its two subtriangles. We call these subtriangles *low* and *high* children. Figure 4 shows,

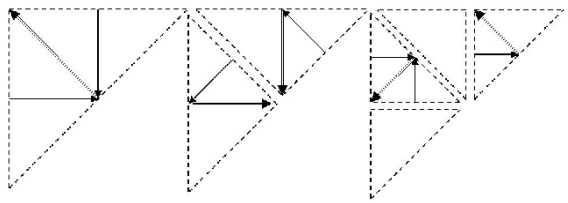

Figure 3: Unbalanced triangular decompositions of the basic triangle.

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".

By observation of Fig. 4, 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 subtriangles of a parent triangle P with direction d, Algorithm 1.

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

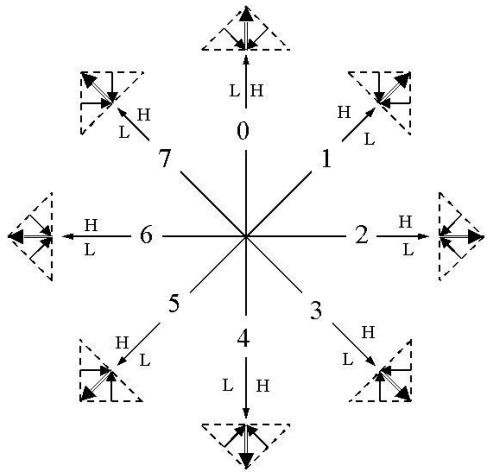

Figure 4: Positions of low and high subtriangles

Algorithm 1 Children apex directions

```
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 if
end
```

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:

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

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 2, *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.

Algorithm 2 Children apex position calculation

```
begin
switch (P.d)
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/root(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/root(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/root(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/root(2));
end if
end
```

4.2 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. 5 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 2.

Algorithm 3 Insertion

```
Input: object_for_insertion OBJ , Directory D,
       blocking_factor, max_population and max_depth
       of the tree
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_region population > max_population then
   Increment the population in D of target_region;
   else;
   perform Split;
end if
end
```

Given the sample decomposition from Fig. 5, the directory would be as shown in Table 2. 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'.

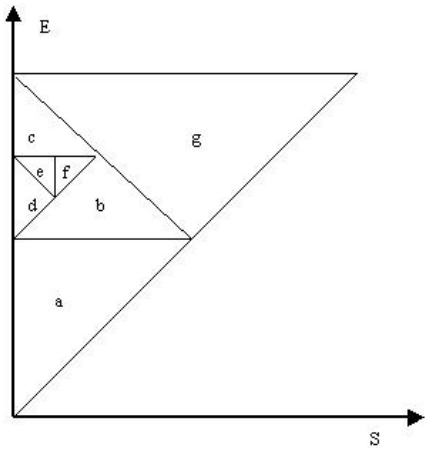

Figure 5: Running example regular decomposition

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

Table 2: Directory for sample tree

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 5 and Table 2, 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.

Algorithm 4 Split

```
Input: Split Region SR, Directory D, blocking_factor max_pop, max_depth of the tree
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
  if child region is POINT then
    Exit
  end if
end while
end
```

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.

4.3 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 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`. 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

Algorithm 5 Query algorithm

```
begin
Input: Directory D , Query region Q
Output: Containing leaves A1, Overlapping leaves A2
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 if
end while
end
```

triangle intersects a rectangle are straightforward geometric operations based on the vertices of the two operations.

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.

4.4 Deletion algorithm

Algorithm 6 `Deletion` 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 booth children falls under the one third of the blocking factor.

Algorithm 6 Deletion

```
Input: object_for_deletion, Directory D , blocking_factor
begin
delete_region = region where object_for_deletionbelongs
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 if
end
```

4.5 Update algorithm

Update of the interval, which causes change in region where interval will belong. Update can be seen as delete and insert and therefore handled by `Delete` and `Insert` algorithms. Update is particularly common for *now-relative* data when interval ending point stops following the current time and has to be closed. When an update is required initially the deletion algorithm is applied, which may cause the merging of regions, then an insert is performed as explained in subsection 4.2.

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 (Kriegel et al. 2000).

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, updateability, concurrency control and space efficiency. Furthermore it has been proven (Kriegel et al. 2000) that the RI-tree is superior to the Window-List (Ramaswamy 1997), Oracle Tile Index (T-Index) and IST-technique (Goh & et al. 1996) 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 (Enderle et al. 2005) 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 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 subsection 4.5, 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 (Kriegel et al. 2001). 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 (Kriegel et al. 2000).

Relations for testing the performance of the TD-tree were altered with column *Region*, which is calculated according the algorithm as explained in subsection 4.2. 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 table. 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. 6 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 add required space for the directory table.

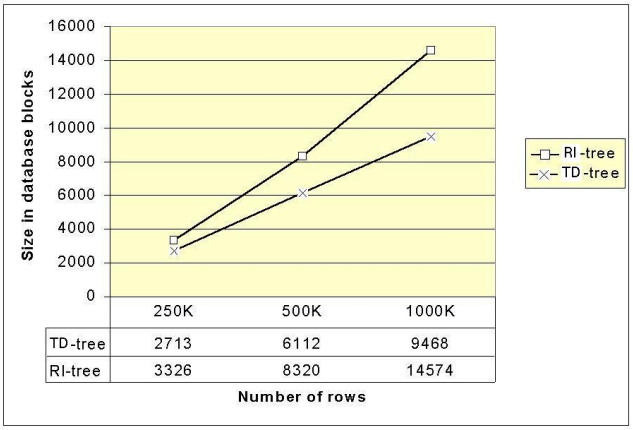

Figure 6: Comparison of the space usage (Table plus indexes)

To measure the query performance we used a data set of one million tuples. Results shown in Fig. 7 and Table 4 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 (Hellerstein et al. 1997) 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.

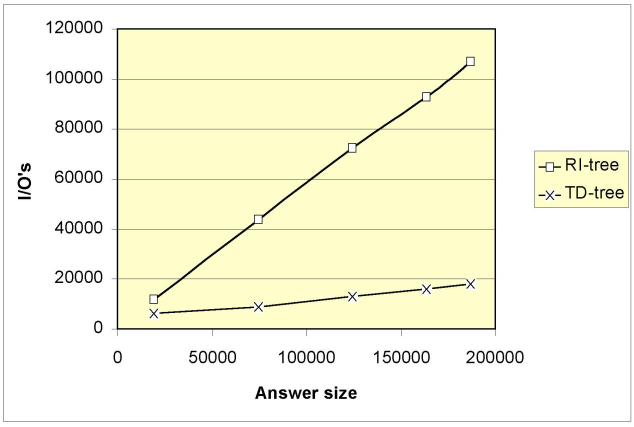

Figure 7: Physical disk I/O as a factor of answer size

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-

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

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

Answer Size	$TD-tree$ *CPU*	$RI-tree$ *CPU*	$TD-tree$ *Resp. time*	$RI-tree$ *Resp. time*
19365	276	293	6	8
74727	301	878	18	26
124280	383	1,321	27	42
163530	427	1,622	37	55
186795	460	1,942	42	63

Table 4: CPU Usage and query response time I/O

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.

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 and uniformly distributed start and exponentially distributed length 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 for 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 `Employment` table and index structures for TD-tree is much smaller than the number of total blocks required for RI-tree table and index structures. This difference increases with increasing number of interval objects, as shown in Fig. 6.

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 7 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.

Due to the query algorithm, which is able to bulk include or reject relevant sub-trees, CPU usage is kept low, which together with low disk I/O results in better query response time, Table 4.

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 demonstrably better than existing approaches for handling temporal, and more generally, interval data. More specifically, in this study, we:

- Classified the traditional interval relationships between pairs of intervals and identified which relationships represent challenging tasks for temporal data access methods;
- 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 the 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 structure and at the same time has some characteristics of hashing because it only stores data in

leaf nodes. In contrast to hashing methods that do not perform well on range queries, TD-trees can efficiently answer a wide range of different query types. This efficiency is ensured as it is possible to quickly include or prune all relevant and irrelevant subtrees without requiring disk accesses to the internal nodes, using only the query algorithm proposed in this paper. It is important to mention that the management od the virtual structure is done automatically by using database triggers, which fire on insert and calculates and updates the region of the record and also increments the region in the directory. If required, it also initiate and performs split. Similarly, database trigger fires on update/delete and performs actions in line with `Delete` and `Insert` Algorithm.

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 fact that the TD-tree can be incorporated within commercial RDBMS, makes the TD-trees superior to other methods proposed for temporal data.

Presented concept can be extended to more dimensions and therefore applied to spatio-temporal data.

References

Allen, J. (1983), 'Maintaining knowledge about temporal intervals', *Communications of the ACM* **26**(11), 832–843.

Ang, C. & Tan, K. (1994), 'The Interval B-tree', *Information Processing Letters* **53**(2), 85–89.

Bliujute, R. & et al. (2000), Light-Weight Indexing of General Bitemporal Data, *in* 'Statistical and Scientific Database Management', pp. 125–138.

Date, C., Darwen, H. & Lorentzos, N. (2002), *Temporal Data and the Relational Model*, Morgan Kaufmann.

Dyreson, C. E., Snodgrass, R. T. & Freiman, M. (1995), Efficiently Supporting Temporal Granularities in a DBMS, Technical Report TR 95/07.
URL: *citeseer.nj.nec.com/dyreson95efficiently.html*

Elmasri, R., Wuu, G. & Kim, Y. (1990), 'The time index: An access structure for temporal data', *Proc. 16th Conf. Very Large Databases* pp. 1–12.

Enderle, J., Schneider, N. & Seidl, T. (2005), 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', VLDB Endowment, pp. 385–396.

Goh, C. H. & et al. (1996), 'Indexing temporal data using existing b+-trees', *Data and Knowledge Engineering* (18), 147–165.

Guttman, A. (1984), R-Trees: a Dynamic Index Structure for Spatial Searching, *in* 'Proceedings of the 1984 ACM SIGMOD international conference on Management of data', pp. 47–57.

Hellerstein, J., Koutsupias, E. & Papadimitriou, C. (1997), 'On the Analysis of Indexing Schemes', *16th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems* .

Kolovson, C. & Stonebraker, M. (1991), 'Segment indexes: Dynamic indexing techniques for multidimensional interval data', *Proc. ACM SIGMOD* pp. 138–147.

Kriegel, H.-P., Potke, M. & Seidl, T. (2001), Object-relational indexing for general interval relationships, *in* 'Proc. 7th Intl Symposium on Spatial and Temporal Databases (SSTD01)'.

Kriegel, H.-P., Ptke, M. & Seidl, T. (2000), 'Managing intervals efficiently in object-relational databases', *Proceedings of the 26th International Conference on Very Large Databases* pp. 407–418.

Kumar, A., Tsotras, V. & Faloutsos, C. (1998), 'Designing Access Methods for Bitemporal Databases', *IEEE Transactions on Knowledge and Data Engineering (TKDE'98)* **10**(1), 1–20.

Lanka, S. & Mays, E. (1991), 'Fully persistent b + trees', *Proc. ACM SIGMOD Conf. on the Management of Data* pp. 426–435.

Nascimento, M. A. & Dunham, M. H. (1999), 'Indexing Valid Time Databases via B^+-Tree', *IEEE Transactions on Knowledge and Data Engineering* **11**(6), 929–947.

Ramaswamy, S. (1997), Efficient Indexing for Constraint and Temporal Databases, *in* 'Proceedings of the 6th International Conference on Database Theory', pp. 419–431.

R.Elmasri, Wuu, G. & V.Kouramajian (1993), 'The Time Index and the Monotonic B^+-Tree', *In A. Tansel et.al., editors Temporal Databases: Theory Design and Implementation* pp. 433–456.

Salzberg, B. & Tsotras, V. J. (1999), 'Comparison of Access Methods for Time Evolving Data', *ACM Computiong Surveys* **31**(1).

Snodgrass, R. T. (2000), *Developing Time-Oriented Database Applications in SQL*, Morgan Kaufmann.

Stantic, B., Khanna, S. & Thornton, J. (2004), '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.

Stantic, B., Terry, J., Topor, R. & Sattar, A. (2010), 'Indexing Temporal Data with Virtual Structure', *Advances in Databases and Information Systems - ADBIS2010* pp. 591–594.

Stantic, B., Thornton, J. & Sattar, A. (2003), '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* pp. 174–181.

Szalay, A. S., Gray, J., Fekete, G., Kunszt, P. Z., Kukol, P. & Thakar, A. (2007), 'Indexing the Sphere with the Hierarchical Triangular Mesh', *CoRR* **abs/cs/0701164**.

Tsotras, V. J., Gopinath, B. & Hart, G. (1995), 'Efficient management of time-evolving databases', *IEEE Trans. Knowledge and Data Eng* **7**(4), 591–608.

V.Kouramajian & et.al (1994), 'The time index+: An incremental access structure for temporal databases', *In Proceedings of the Third Interbnational Conference on Knowledge and Data Engineering (CIKM'94)* pp. 232–242.