The TP-Index: A Dynamic and Efficient Indexing Mechanism for Temporal Databases

Han Shen Beng Chin Ooi Hongjun Lu

Department of Information Systems and Computer Science
National University of Singapore, Singapore 0511.
email: {shenh, ooibc, luhj}@iscs.nus.sg

Abstract

To support temporal operators efficiently, indexing based on temporal attributes must be supported. In this paper, we propose a dynamic and efficient index scheme called the time polygon (TP-index) *for temporal databases. In the scheme, temporal data are mapped into a two-dimensional temporal space, where the data can be clustered based on time. The data space is then partitioned into time polygons where each polygon corresponds to a data page. The time polygon directory can be organized as a hierarchical index. The index handles long duration temporal data elegantly and efficiently. Our performance analysis indicates that the time polygon index is efficient both in storage utilization and query search.*

1. Introduction

More recently, it is realized that "time" constitutes an important dimension in the evolution of a database, and hence historical information which is useful should be retained in the underlying database. However, research in temporal databases has largely been focused on extensions of existing data models for the proper handling of temporal information. The full potential of temporal databases, however, can only be realized if there exist temporal operators which can enhance the retrieval capabilities of the underlying database management system. As with other databases, efficient indices and storage structures are necessary to support these operators. So far, a number of storage techniques have been introduced. Rotem and Segev[Rot87] proposed that the time dimension could be viewed as one of the dimensions in a multi-dimensional space and hence temporal data could be organized using a multi-dimensional partition file. But their approach cannot handle time intervals effectively. Both Lomet and Salzberg[LoB90] and Kolovson and Stonebraker[KoS89] studied the problem with the assumption that historical data are stored separately from current data in an optical disk, which is more appropriate in the context of a rollback database. Gunadhi and Segev[GuS93] investigated the methods of indexing time-dependent data within the concept of a first normal form relational representation of temporal data. Elmasri et al. [EWK90] proposed a *time index* scheme which provides access to temporal data valid in a given time interval. However, duplications may exist on some selected time intervals, and thus degrade the space utilization and query efficiency to some extent.

In this paper, we map temporal data into data points in a triangular space. The X dimension of the space is the time and the Y dimension of the space is the length of period. Data points in the space are clustered and partitioned into polygons. Data points falling within the same polygon are stored in one page. To efficiently organize these data pages, we propose an index scheme called *the Time Polygon index (TP-index)* to support retrievals for various types of queries. With such an organization, duplication is avoided. Further, the index is well suited for append-only database where the data are inserted to the right most of the time dimension and are not bounded. A performance analysis is conducted and the results indicate that the TP-index is an efficient indexing structure for temporal databases.

The rest of this paper is organized as follows. Section 2 defines an *interval-spatial transformation* which maps a given time interval to a point in a two-dimensional temporal space, and analyzes the spatial representations of different temporal queries in the

The work was in part supported by NUS Research Grant RP910694 and RP910654

temporal space. In Section 3, we describe structure of the TP-index. Its update algorithms are then presented in Section 4. Section 5 analyzes the performance of the TP-index scheme, and compares with the time index approach [EWK90]. We conclude in Section 6.

2. Temporal Selection as Spatial Search

We represent the time dimension using both discrete time points and time intervals. A *time interval*, denoted by $[a, b]$ is defined to be a set of consecutive equidistant time instants (points), where a is the first time instant and b is the last time instant of the interval. The *time dimension* is represented as a time interval [0,*now*], where 0 represents the starting time and *now* refers to the current time which is continuously increasing.

Temporal data can be viewed as spatial objects in a multi-dimensional space in which one or more dimensions are time dimensions. The following is an *interval-spatial* transformation which maps a given time interval to a point in a two-dimensional temporal space:

Definition [Interval-spatial transformation]

Let $\mathcal{I}$ be the set of all time intervals $[a, b]$ in the time dimension $[0, now]$, and $\mathcal{P}$ be the set of discrete time points in the time dimension. The *interval-spatial transformation*, denoted by $\mathcal{T}$, is a function from $\mathcal{I}$ to $\mathcal{P}^2$, such that $\mathcal{T}([a, b])= (a, b - a)$.

The *interval-spatial* transformation forms the basis for transforming a temporal relation to a *spatial rendition*. The spatial rendition obtained provides a highly visual representation of the answers to temporal queries.

Consider a database which keeps record of visitors to the United States. A **residence** relation for this database is shown in Figure 1. The tuples can be mapped to discrete data points in the two-dimensional space shown in Figure 2.

Observe that the spatial rendition in Figure 2 represents temporal data with the following constraints: (1) any tuple with a starting time $v_s = a$ must be mapped to a point on the line $x = a$; (2) any tuple with an ending time $v_e = b$ must be mapped to a point on the line $x + y = b$; (3) any tuple with a time duration c must be mapped to a point on the line $y = c$.

Obviously, any temporal query can be transformed into a spatial search operation in the temporal space. Consider the following queries: (a) list all persons who entered the country on or before day t_a; (b) list all persons who left the country on or after day t_b; (c) list all persons who remained in the country for a total duration of t_c or less days. The answers to each of these queries can be found by retrieving all data points which fall in the regions shown in Figure 3(a), (b), and (c) respectively. Furthermore, a temporal query with multiple selection criteria can be transformed into an intersection of the regions corresponding to the individual selection criterion. For instance, the query "List all persons who entered the country on or before t_a and left on or after t_b", shown in Figure 3(d), is simply the intersection of the two regions in Figure 3(a) and (b). Similarly, disjunctions of selection criteria can be modeled as the union of respective regions. For example, the query "List all persons who entered on or before t_a or remained for t_c days or less" can be answered by retrieving all points in the region shown in Figure 3(e). Finally, a degenerate instance of such queries corresponds to selection on time points, such as "Find all persons who were in the United States on t_f" corresponds to the search space shown in Figure 3(f).

tuple	pid	entry_pt	period
t1	p1	NY	[0,3]
t2	p1	LA	[4,*now*]
t4	p2	SFO	[0,5]
t7	p3	LA	[0,7]
t8	p3	SFO	[8,9]
t10	p4	NY	[2,3]
t11	p4	LA	[8,*now*]
t12	p5	LA	[10,*now*]
t13	p6	NY	[12,*now*]
t14	p7	NY	[11,*now*]

Figure 1: The **residence** relation

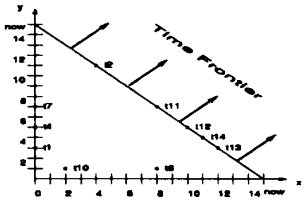

Figure 2: A spatial rendition of the **residence** relation

3. Organization of the TP Directory

Once temporal tuples are mapped into data points in the two-dimensional temporal space using the interval-spatial transformation, the data points should

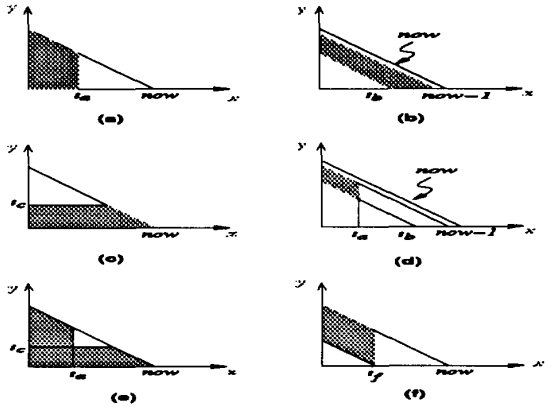

Figure 3: Search regions in the temporal space

be organized such that they can be efficiently retrieved based on temporal relationships. In this section, we present our strategies for organizing the data space.

Data points in the triangular space must be partitioned into groups such that each group can be stored in one page. A B^+-tree like index structure, called the *TP-tree* is used to index the data items. An internal node of the tree has entries of the following format:

[child-pointer, polygon],

where *child-pointer* points to a child node and *polygon* describes the entire data space of the child node. A polygon in an internal node is called an *internal bounding polygon*. It encloses other bounding polygons as its subspaces. A leaf node of the TP-tree consists of leaf entries and a tree pointer pointing to a succeeding leaf node. A leaf entry has the form:

[bucket-pointer, polygon],

where *bucket-pointer* points to a data bucket where the data points in the polygon are stored. A polygon containing data points is called *leaf bounding polygon*. Internal bounding polygons and leaf bounding polygons are all constrained to the five well-formed shapes illustrated in Figure 4. The formal definitions can be found in [SOL93].

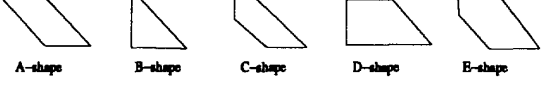

Figure 4: Five well-formed shapes of bounding polygons

A polygon has to be partitioned when the entries in it overflows. Partitioning can be performed by introducing a line parallel to X-axis (called an X-line), or a line parallel to the *time-front* (called a time-line). We refer to the above two partitions respectively as *X-partition* and *time-partition*. Two resultant polygons formed after a partition are called *buddies*. During merging phase, only buddies can be merged so that the resultant polygon can still have a well-formed shape. Figure 5 shows that recursive partitions using X-partition and time-partition result in smaller polygons with the well-formed shapes. The three rationales for allowing only X-partition and time-partition are: One, the shapes are similar to those of query regions; Two, irregular shapes make maintenance and testing expensive; Three, the resultant subspaces of an X-partition or time-partition still have the well-formed shapes.

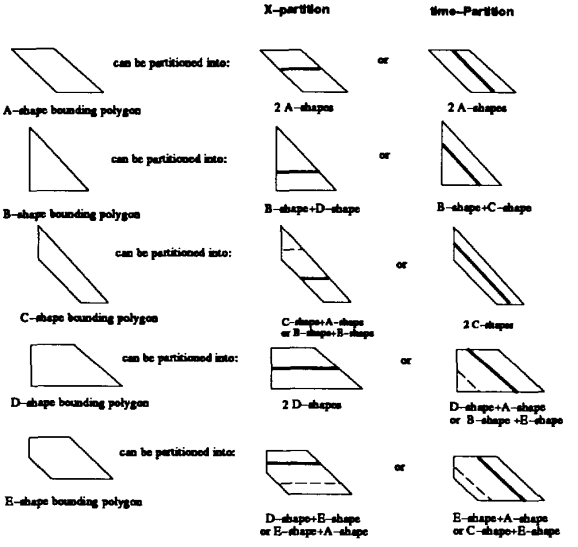

Figure 5: Partitions for different leaf bounding polygons

Throughout this paper, we use m_T and M_T to respectively denote the minimum and maximum number of entries allowed in a node of a TP-tree.

A TP-tree has the following properties:

$Prop_1$: The union of all the bounding polygons described by the entries of a node spans the whole data space of that node.

$Prop_2$: The root of a TP-tree has at least two children unless the TP-tree is a one node tree.

$Prop_3$: Each node has between m_T to M_T entries unless it is the root of a TP-tree.

$Prop_4$: Any bounding polygons described by the entries within one node do not overlap.

$Prop_5$: Each polygon described by an entry of a node adopts one well-formed shape.

$Prop_1$ ensures that there is no *dead space* in an internal bounding polygon which is not covered by its enclosed polygons. $Prop_4$ guarantees that only one path needs to be traversed in order to search for a data point in the temporal space. $Prop_5$ guarantees that all the internal and leaf bounding polygons have well-formed shapes. The structure of a TP-tree is illustrated in Figure 6.

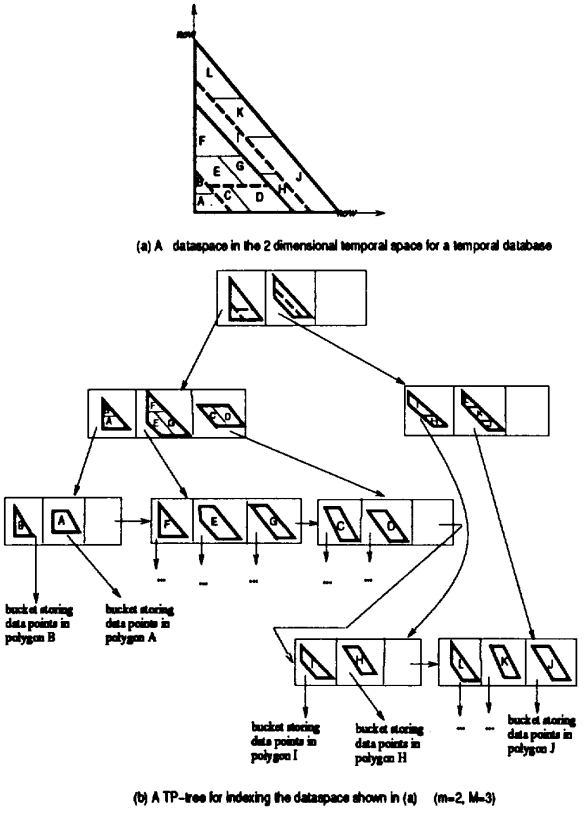

Figure 6: A TP-tree

4. Operations

In this section, we present the algorithms for accessing and updating a TP-tree.

4.1. Searching

Temporal search in the transformed space is similar to spatial search in a spatial data structure [Ooi90]. Descending the tree from the root node, for each data entry whose data space (a polygon) intersects with the query region, its subtree is traversed.

Algorithm: Search

```
Search (p, q)
Input: q – query region;
    p – pointer to a node in a TP-tree, initially to the root node.

for each entry e in the node pointed by p do
    if e is a leaf entry whose data space is a polygon B then
        for each data point t in B do
            if t satisfies the search conditions then
                add t to the answer;
    else
        if INTERSECT(e.polygon, q) ≠ null then
        Search(e.child-pointer, q)
end Search
```

4.2. Insertion

To insert a data point, its temporal attribute is used to search for the leaf entry pointing to a data bucket where the data point should be put into. Then, the data point is inserted into the data bucket pointed by the entry. If it overflows, the corresponding bounding polygon is partitioned into two buddy subspaces and the data points are distributed accordingly.

Algorithm: Insert

```
Insert (p, d)
Input: d – data point to be inserted;
    p – pointer to a node in a TP-tree, initially to the root node.

for the entry e in the node pointed by p whose
e.polygon contains the position of d do
    if e is a leaf entry whose data space is polygon B then
        insert data point d into B;
        if B overflows then Split(B);
    else Insert(e.child-pointer, d);
end Insert
```

4.3. Splitting

After insertion, a data bucket may overflow and split is needed. This can be performed by partitioning the corresponding leaf bounding polygon into two buddy polygons, and distributing the data points into the two resultant polygons. The selection of an X-partition or a time-partition is based on whether a partition can evenly distribute the data points into two subspaces. Furthermore, to ensure good clustering, a bounding polygon with almost equal intervals along the X-line and the time-line is preferred.

Algorithm: Split

Split(B)
Input: B – a leaf bounding polygon corresponding to an overflowed data bucket.

X-partition(B, l_x, xn_1, xn_2) to obtain an X-line l_x which divides the polygon B into two subspaces with xn_1 and xn_2 numbers of data points;
Time-partition(B, l_t, tn_1, tn_2) to obtain a time-line l_t which divides the polygon B into two subspaces with tn_1 and tn_2 numbers of data points;
if $|xn_1 - xn_2|$ approximately equals to $|tn_1 - tn_2|$ **then**
    choose a partition which generates evener intervals in its subspaces S_1 and S_2
else
    choose a partition which more evenly distributes the data points in its subspaces S_1 and S_2;
record the buddy relationship between S_1 and S_2;
creat a new leaf entry e_{new} and set $e_{new}.polygon \leftarrow S_2$;
for each parent node P having an entry e whose $e.polygon$ is B **do**
    modify e by $e.polygon \leftarrow S_1$;
    if one more entry of P is needed to accommodate e_{new} **then**
        add the entry e_{new} into P;
        if P has more than M_T entries **then**
            Internal-partition(P) to partition the internal bounding polygon of P
end Split

The algorithms `X-partition` and `Time-partition` respectively introduce an X-line and a time line to divide a leaf bounding polygon into two subspaces with almost same number of data points in each. Due to space constraint, we will not outline the algorithms here.

Partitioning a leaf bounding polygon may cause an internal bounding polygon to be partitioned. As shown in Figure 7, when a leaf bounding polygon W is partitioned into subspaces H and G, the entry corresponding to W in the node N is also split into two entries. If N has more than M_T entries, it has to be split into two nodes N_1 and N_2, and the internal bounding polygon S is partitioned into S_1 and S_2. The entry pointing to N in the parent node P is also replaced by two corresponding entries. Because an internal bounding polygon encloses subspaces of polygons, we may not be able to find an X-line or a time-line that can evenly partition the enclosed polygons without any overlap. This problem can be solved if we allow a node to have more than one parent[1]. The entry whose bounding polygon has been cut by a partition line appears as entries in both new resultant nodes. For instance, in Figure 7(e)(f), a time-line dividing S into S_1 and S_2 also cuts the subspace F into F' and F''. To avoid poor storage efficiency, we allow N_1 and N_2 to have entries pointing to the same bucket for polygon F. For efficiency reason, a partition line cutting through subspaces is used only if the algorithm cannot find an X-line or time-line that roughly partitions the entries.

[1]at the lowest level, a data bucket can have more than one leaf nodes which have entries pointing to the bucket

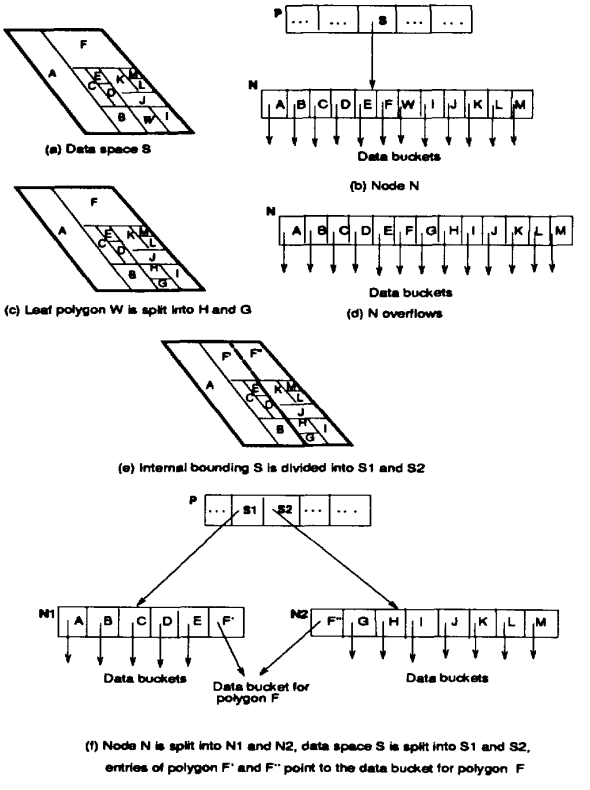

Figure 7: Partitioning an internal bounding polygon

The internal partition algorithm is described as follows.

Algorithm: Internal-partition

Internal-partition(P)
Input: P – a node which has U number of entries ($U > M_T$).

Par_1:
    for each existing X-line which goes through the data space S of P **do**
        calculate the numbers of the enclosed polygons in the two subspaces divided by the X-line, and record the numbers by xn_1 and xn_2 respectively;
    among these X-lines, choose one which generates the smallest $\sigma_1 = |xn_1 - xn_2|$;
Par_2:
    for each existing time-line which goes through the data space S of P **do**
        calculate the numbers of the enclosed polygons

```
        in the two subspaces divided by the time-line, and
        record the numbers by tn1 and tn2 respectively;
    among these time-lines, choose one which generates
    the smallest σ2 = |tn1 − tn2|;
if σ1 ≤ σ2 then
    choose a partition line from Par1 and σ ← σ1
    else choose a partition line from Par2 and σ ← σ2;
if σ > threshold then
    Par3 :
        for each existing X-line or time-line that
        partially goes through the dataspace S do
            virtually extend it to go through the whole space
            of S, it divides S into two subspaces with n1 and n2
            numbers of enclosed polygons respectively;
        choose the line that generates the smallest
        |n2 − n1| and (n1 + n2 − U);
S is divided into S1 and S2 by a chosen partition
line, record S1 by a node N1 and S2 by a node N2;
for each parent node PP of P do
    for the entry e in PP pointing to P, e.polygon ← S1;
    add one entry e_new into PP, e_new.polygon ← S2;
    if PP has more than M_T entries then
        if PP is the root node then
            introduce a new root node N0' with N1
            and N2 as its children
        else Internal-partition(PP) to propagate
        the partition upwards
end Internal-partition
```

4.4. Deletion Algorithm

To delete a data point, a TP-tree is traversed to search for the leaf entry that points to a data bucket containing the data point. Deletion may cause a bounding polygon to underflow. To improve the fill rate, entries contained in an underflowed polygon are merged into its buddy polygon, and merging may propagate upward till the root.

Algorithm: delete

```
delete(d)
Input: d – data point to be deleted.

search for the entry e of a leaf node N that contains
the location of d, S ← e.polygon, delete d from S;
Merge:
if S has less than m_T entries then
    insert the remaining entries into S's buddy SB;
    enlarge the buddy SB to cover the dataspace S;
    if SB overflows then split(SB) ;
    remove the entry e from the node N;
    if N has less than m_T entries then
        repeat from Merge with S being replaced
        by the polygon of N;
end delete
```

4.5. Reorganization of the TP-tree As Time Increases

Because of the nature of temporal databases, most updates occur in an append mode. Insertions of new tuples occur mostly in increasing time value.

Our partitioning strategies and the algorithms designed have the advantage of easy incremental reorganization as time increases. When the current time increases from the old time point now_1 to the new time point now_2, the temporal space expands as shown in Figure 8. A leaf entry e_{new} corresponding to the expanded bounding polygon S_{new} of C-shape is introduced. If the root node N_0 of the TP-tree has less than M_T entries, then e_{new} is added into N_0. If N_0 has M_T entries, one new root N_0' is introduced to the TP-tree with two children N_0 and N_{new} (N_{new} contains the new leaf entry e_{new}). For those temporal data points which have the duration until now_2, shift them into S_{new} in the new TP-tree following the same procedure of inserting data points in a TP-tree.

Figure 8: Increase now from now_1 to now_2

5. Performance Analysis

In this section, we study the performance of the proposed TP-index scheme. We also compare the performance of the TP-index with the time index [EWK90].

5.1. Storage Efficiency

First, we analyze the storage cost for the TP-index.

Let S_T be the total number of data pages (buckets) used for storing N number of data points, S_T^i be the number of pages used for storing the index structure of TP-tree, and S_T^b be the number of data pages used for storing data points in data buckets. We use 4K bytes for the page size, 32 bytes for a pointer p, and 8 bytes for a coordinate. According to Figure 9, (A, x_1, y_1, x_4), (B, x_1, y_1, x_4), (C, x_1, y_1, x_4), (D, x_1, y_1, x_4, d), and (E, x_1, y_1, x_4, d) are respectively sufficient to represent polygons of shape A, B, C, D, and E. So we need at most $pol = 1 + 4 \times 8 = 33$ bytes to represent a bounding polygon. The maximum number of entries in one node is: $M_T(p + pol) + p = 4K$, i.e., $M_T = 62$.

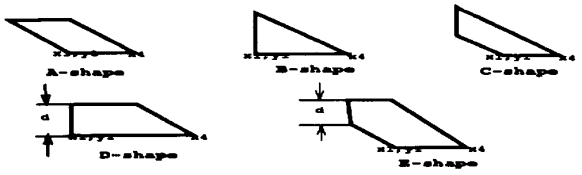

Figure 9: Internal representations for five type polygons

For N number of data points, if we assume that each data bucket stores $M_T ln2$ number of data points on the average, then $S_T^b = \left\lceil \frac{N}{M_T ln2} \right\rceil$. If we suppose a TP-tree with height h_T is $ln2$ full on the average, then

$$S_T^i \approx \left\lceil \frac{N}{M_T{}^2 ln^2 2} \right\rceil \times \left\lceil 1 + \frac{1}{M_T ln2} + \cdots + \frac{1}{(M_T ln2)^{h_T - 1}} \right\rceil$$
$$\approx \left\lceil \frac{N}{M_T^2 ln2} \right\rceil \times \left\lceil \frac{M_T ln2}{M_T ln2 - 1} \right\rceil$$

Thus: $S_T \approx \left\lceil \frac{N}{M_T^2 ln^2 2} \right\rceil \times \left\lceil \frac{M_T ln2}{M_T ln2 - 1} \right\rceil + \left\lceil \frac{N}{M_T ln2} \right\rceil$

Now, we analyze the storage cost for the time index [EWK90]. For explanation purpose, we call a B^+-tree in the time index *BI-tree*. Assume the same system parameters as those of the TP-index, the maximum number of entries in a node of a BI-tree is: $M_E(p+t)+p = 4K$, i.e., $M_E \approx 100$. We also assume that all bucket entries corresponding to the same leaf node is clustered together whenever possible.

Let S_E be the total number of pages used for storing N temporal tuples by the time index approach. $S_E = S_E^i + S_E^b$, where S_E^i is the number of pages required by the BI-tree, and S_E^b refers to the number of pages needed to store all the bucket entries.

We assume that the arrival of temporal tuples is a Poisson process, and hence inter-arrival time is exponentially distributed with mean $1/\lambda$. Let X be the Poisson random variable which represents the number of temporal tuples arriving in a unit time with mean value λ. The duration of each tuple is assumed to be uniformly distributed over the interval $[0, 2\mu]$. Obviously, the size of the BI-tree depends on the number of distinct indexing points in BP (refer to [EWK90]). $|BP|$ can be approximated by $2Ne^{-\lambda}$. Assuming that the nodes in the BI-tree is $ln2$ full on the average, the number of leaf nodes is given by $\lceil \frac{2Ne^{-\lambda}}{M_E ln2} \rceil$. Thus, $S_E^i \approx \left\lceil \frac{2Ne^{-\lambda}}{M_E ln2} \right\rceil \times \left\lceil \frac{M_E ln2}{M_E ln2 - 1} \right\rceil$.

We estimate the number of entries in the leading bucket of each leaf node by the expected number of arrivals in a period μ, which yields the value $\mu\lambda$. The number of incremental entries for each leaf node is: $(M_E ln2)\lambda$. Hence, $S_E^b = \left\lceil \frac{2Ne^{-\lambda}}{M_E ln2} \right\rceil \times \left\lceil \frac{\mu\lambda + M_E \lambda ln2}{M_E} \right\rceil$

Consequently,

$$S_E = \left\lceil \frac{2Ne^{-\lambda}}{M_E ln2} \right\rceil \times \left(\left\lceil \frac{M_E ln2}{M_E ln2 - 1} \right\rceil + \left\lceil \frac{\mu\lambda + M_E \lambda ln2}{M_E} \right\rceil \right)$$

The above expression suggests that even for a fixed N, the storage requirements of the time index varies with the characteristics of the temporal tuples. Unlike the time index, the TP-index is not dependent on the characteristics of the temporal data. This is due to the fact that each temporal tuple is represented only once as a data point in the two-dimensional temporal space. In addition, the TP-tree index approach incurs less space than the BI-tree index approach. Figure 10 shows the results obtained from a simulation study in which N is kept constant at 100000. The results confirm our analysis: space cost for the TP-index remains very low, and also remains constant with no regard for μ or λ. The space cost for the time index, on the other hand, fluctuates as a function of both of these parameters, and the cost is higher than that of the TP-index.

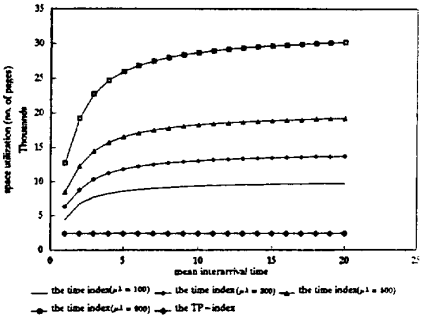

Figure 10: Space utilization

5.2. Query Efficiency

In this subsection, we examine the search performance of the proposed TP-index scheme against the Elmasri's time index.

We analyze a query type which is best supported by the time index, i.e., queries retrieving all temporal tuples whose starting time is between time a and time b. It can be accomplished by (i) traversing the BI-tree to locate the leaf node containing time point a, and (ii) following the sequential links between leaf nodes and retrieving all incremental entries in the buckets, right up to time point b. Let Q_E be the number of page accesses needed to answer the query. We assume that a is a randomly selected time point from interval $[0, now]$, and the length of the interval $b - a$ is an exponential random variable with mean γ. We can obtain: $Q_E \approx \left\lceil \frac{2\lambda\gamma e^{-\lambda}}{M_E ln2} \right\rceil \times \left\lceil 1 + \frac{\mu\lambda + M_E \lambda ln2}{M_E} \right\rceil$ (refer to [SOL93] for the derivation process). As can be seen, the cost is effected by the temporal features μ, γ, and λ.

Let us analyze the query cost of the TP-index now. We denote by Q_T, the number of pages needed to retrieve all temporal data points corresponding to the same query. This temporal query is mapped into a spatial search. The expected number of data points is bounded by $\lambda\gamma$. Approximately, $\lceil \frac{\lambda\gamma}{M_T{}^2 ln^2 2} \rceil$ number of leaf nodes are searched. To search for one leaf node in the TP-tree, h_T number of pages is traversed, where h_T is the height of the TP-tree, $h_T \approx \lceil log_{(M_T ln 2)} N \rceil$. Consider the time for the associated buckets as well, then Q_T is given by $\left\lceil \frac{\lambda\gamma}{M_T{}^2 ln^2 2} \right\rceil \times (1 + h_T)$

Figure 11 depicts the search performances of the time index and the TP-index with $\mu\lambda = 600, \gamma = 1000$ and $N = 5000000$. When the mean arrival rate increases, the performance of the TP-index degrades slightly. However, the result is based on a conservative assumption that h_T number of pages are traversed for each leaf node. For range queries, the number of pages searched for each leaf node on the average is much less than h_T because a query region is usually covered by neighbouring leaf bounding polygons linked by succeeding tree pointers in leaf nodes. Hence, once a leaf node is found, the succeeding leaf nodes can be found by the pointers, there is no need to search each leaf node from the root.

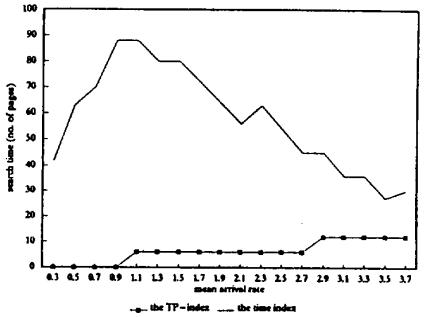

Figure 11: Search performances of the TP-index and the time index

TP-index approach supports other query types with a single type of region search, while the time index approach has to scan the entire database. Some detailed analysis can be found in [SOL93].

As a summary, the TP-index is an efficient index scheme in terms of space efficiency and search efficiency. We are now conducting some experiments on the time index [EWK90], the R-tree [Gut84] and our TP-index to further compare the performance.

6. Conclusion

Most existing indices are not appropriate for indexing temporal data. In this paper, we addressed the mapping of temporal data into data points in a two-dimensional temporal space and, proposed a dynamic and efficient time index called the time polygon index (TP-index). The data are clustered based on temporal characteristics and are organized in a B$^+$-tree like index called the TP-tree. The performance analysis indicates that the TP-index is efficient in both storage requirement and query retrieval.

Acknowledgements

The authors would like to thank to Mr. Cheng Hian Goh for his joining to the discussion on the draft version of this paper.

References

[ElW90] R. Elmasri and G. Wuu, "A Temporal Model and Query Language For Temporal Databases," *Proc. 6th Int'l. Conf. On Data Engineering*, pp. 76-83, 1990.

[EWK90] R. Elmasri, G. T. J. Wuu, and Y. J. Kim, "The Time Index: An Access Structure For Temporal Data," *16th Int'l. Conf. On Very Large Data Bases*, pp. 1-12, 1990.

[Gut84] A. Guttman, "R-tree: A Dynamic Index Structure For Spatial Searching," *Proc. ACM SIGMOD Int'l. Conf. On Management Of Data*, pp. 47-57, 1984.

[GuS93] H. Gunadhi and A. Segev, " Efficient Indexing Methods For Temporal Relations," IEEE Trans. On Knowledge and Data Eng., 5, 3, pp. 496-509, 1993.

[KoS89] C. Kolovson and M. Stonebraker, "Indexing Structures Techniques For Historical Databases," *IEEE 5th Int'l. Conf. On Data Engineering*, pp. 127-137, 1989.

[LoB90] D. B. Lomet and B. Salzberg, "The HB-Tree: A Multiattribute Indexing Method With Good Guaranteed Performance," *ACM Trans. On Database Systems, 15, 4*, pp. 625-658, 1990.

[Ooi90] B. C. Ooi, *Efficient Query Processing In Geographic Information Systems*, Lecture Notes in Computer Science 471, Springer-Verlag, 1990.

[Rot87] D. Rotem and A. Segev, " Physical Organization Of Temporal Data," *Proc. Int'l. Conf. On Data Eng.*, pp. 454-466, 1987.

[SOL93] H. Shen, B. C. Ooi, and H. J. Lu, " The TP-Index: A Dynamic and Efficient Indexing Mechanism for Temporal Databases," Technical Report No. TRC6/93, Department of Information Systems and Computer Science, National University of Singapore, 1993.