Performance Evaluation of Temporal Indexing Structures

Mong Li Lee Beng Chin Ooi Kian-Lee Tan Peng Jiang
School of Computing
National University of Singapore
Lower Kent Ridge Rd, Singapore 119260
email: {leeml, ooibc, tankl}@comp.nus.edu.sg

Abstract

A number of indexing structures for temporal databases have been proposed in the past few years. Many have been proposed without much comparison with other indexing structures. In this paper, we present an extensive comparative performance evaluation of the Time Split B-tree (TSB-tree), Append-Only tree (AP-tree), R-tree, and Time-Polygon tree (TP-tree). For the R-tree structure, we examined two ways of representing temporal data. We also extended the TP-tree to cater to key-range time-slice queries. We implemented these indexes, and evaluated them on a wide range of indexing attributes, queries and large data sets. The indexes are constructed on time-invariant key and transaction time, time-invariant key and valid time, and time-varying key and valid time. The TI-Time software from the Time Center (University of Arizona) is extended to generate large data sets with different data distributions of one million versions. Temporal queries such as time-slice, key-range time-slice, and past versions queries are used to benchmark the efficiency of the indexes. Our results provide insights into the strengths and weaknesses of these indexes. With the proliferation of temporal indexes and the lack of extensive performance study, the experimental results are important to serve as guidelines for selection of a suitable index and the design of a new index.

1 Introduction

In a *temporal database*, the notion of *time* is used to model the temporal behavior of data so that past data can be maintained and accessed. Two widely used notions of time are the *valid time* and *transaction time*. Valid time represents the time interval when the database fact is true in the modeled world, whereas transaction time is the time point when a transaction is committed. A database that supports both time dimensions is known as a *bitemporal* database. Temporal records are typically modeled as collections of line segments, each has a *begin time*, an *end time*, a *time-invariant attribute*, and a *time-varying attribute*.

To support efficient query retrieval based on time and key, new access methods that index and manipulate data based on temporal relationships are required. Temporal data can be indexed based on its start time, end time, or the whole interval, together with the time-varying attribute or time-invariant attribute. Indexing structures that are based on start time or end time are straightforward and structurally similar to existing indexes such as B^{+}-tree [C79]. Such an index is not efficient for answering queries that involve *time-slice* since no information on the data space is captured in the index. While this problem can be alleviated by replicating records, replication increases storage cost as well as the height of the index which affects the query and update cost. If temporal data are to be indexed as an interval, they have to be transformed into points or data space information must be properly captured as in the R-tree. As such, most temporal indexes proposed so far are mainly based on the conventional B^+-tree and spatial indexes like the R-tree [G84].

The Write-Once B-tree (WOBT) [E86] by Easton is a modification of the B^+-tree, that indexes out-of-date data on write-once optical disks (WORMs). Many variations of this method have since been proposed including the Time-Split B-Tree [LS89], the Persistent B-tree [LM91], the Multiversion B-tree [BGO93] and the Multiversion Access Structure [VV97]. Lomet and Salzberg proposed a variation of the WOBT, called the Time-Split B-Tree (TSB-Tree) [LS89, LS90, LS93] for indexing temporal data based on both key attribute and transaction time. The Multiversion B-tree (MVBT) and the Multiversion Access Structure (MVAS) are similar to WOBT but provide efficient support for deletions which are functionally required for valid time databases. Elmasri, Wuu and Kim introduced a B^+-tree based index called the Time-Index [EWK90, EJK92, EWK93] to index valid time data. Since intervals are line segments that overlap a range of index values, duplication is used to ensure that data can be retrieved correctly. Gunadhi and Segev's observation that temporal database is mainly append-only led them to develop the Append-only Tree (AP-tree) [GS93], another variation of B^+-tree, where a new tuple is always inserted into the rightmost leaf node of the index. To handle both valid time and key attribute, a two-level structure called Nested ST-Tree (NST) was proposed. The upper level of the NST is a B^+-tree for key values, and the second level is the AP-tree of each key value.

The R-tree [G84] was originally proposed to index multi-dimensional spatial data. Being a multi-dimensional index, the R-tree can be used to index temporal data as line segments or as points. A two-dimensional R-tree can be used to index time intervals of the form $[T_s,\ T_e]$ (T_s and T_e are the start and end time of the interval) as line segments and key values of the time-invariant attribute. Alternatively, time intervals can be represented as two-dimensional data points, and hence the data can be indexed using R-trees in a three-dimensional space. The Time Polygon tree(TP-tree) [SOL94] was proposed to support valid time on the time interval. In the TP-tree, temporal data are mapped into data points in a triangular temporal space.

The proliferation of temporal indexing techniques is evident. Unfortunately, most of the techniques have been proposed with limited comparative study with other indexing structures. Like most indexing structures, the desirable properties of an efficient temporal index include efficient usage of disk space, speedy retrievals of queries and efficient update processing. In this paper, we present an extensive performance evaluation of the TSB-tree, AP-tree, R-tree and TP-tree. For the R-tree structure, we examined the two ways of representing temporal data. We also extended the TP-tree to cater to key-range time-slice queries. We believe that a comprehensive comparative study will provide database practitioners and application implementors some guidance in selecting an appropriate index. Furthermore, such a study will also provide insights into the strengths and weaknesses of each index mechanism, which may be helpful to designers in developing new and more robust indexes. We have omitted the Time-Index as a previous study between the TP-tree and the Time-Index showed that the Time-Index is rather inefficient [SOL94]. Our work can be seen as a complement to the recent survey by Salzberg and Tsotras [ST97]. There, they provided a theoretical comparison of the various temporal indexing techniques and presented the worst case analysis. Here, we implemented five of the techniques and study their performance by taking into consideration factors such as buffer replacement algorithms and data distribution.

We shall not consider bitemporal indexing [KTF95] since most existing indexes were proposed for either valid or transaction time data only. We implemented the indexes on the time-invariant key and transaction time, time-invariant key and valid time, and time-varying key and valid time, and our evaluation covers a wide range of large data sets and queries. We extended the TI-Time source code provided by the Time Centre at the University of Arizona to generate large data sets with different distributions of valid time and transaction time data. Each data set contains one million records. The efficiency of these indexes is extensively tested using a canonical set of temporal queries that retrieve data based on valid time or transaction time, and time-invariant key or time-varying key. Due to the size of the database, index nodes of indexes cannot be retained in memory throughout the processing, and hence frequent re-fetching may be necessary. To reduce node re-fetching, we have also implemented a replacement strategy that has been tailored for hierarchical indexing structures.

Our performance results show some very interesting findings. While the TSB-tree is superior in most cases, it performs poorly for past versions queries. However, better results for the TSB-tree can be obtained by incorporating the C-list in [VV97]. On the contrary, the AP-tree which performs the worst in most cases outperforms the other techniques in past versions queries and inclusion search. The extended TP-tree is generally better than the R-tree-based techniques for key-range time-slice queries. Our results also show that the relative performance of the techniques for the various queries are similar regardless of the indexing attributes. In terms of storage efficiency, the TSB-tree fairs poorly which may affect its scalability, while the AP-tree requires the least amount of storage. As for the cost of insertion, our results show that the 3-D R-tree is superior while the AP-tree performs the worst.

The remainder of this paper is organized as follows. In the next section, we propose and examine an extended set of canonical queries. In Section 3, we review the set of indexes under study. For each index, we discuss its structure and its search algorithm, and its extension for handling queries that it was not designed for. We discuss the implementation and the experiments that we conduct in Section 4. Performance results for experiments on indexing time-invariant key and transaction time are presented in Section 5. Results for experiments on indexing time-invariant key and valid time, and time-varying key and valid time are respectively presented in Sections 6 and 7, and finally we conclude in Section 8 by summarizing the strengths and weakness of the indexes which can serve as guidelines on the choice of an appropriate temporal index mechanism.

2 Temporal Queries

Various queries for temporal databases have been discussed in the literature [GS93, S94, SOL94]. Like many other applications, temporal indexes must be able to support a common set of simple and frequently used queries efficiently. In this section, we describe the queries that constitute the canonical set of temporal queries. These queries shall be used to benchmark the efficiency of a temporal index.

We shall use a *tourist* temporal relation shown in Table 1 as an example in our discussion that follows. We assume that the time granularity for this application is one day for both valid and transaction time. Consider the first tuple. The object with pid $p1$ is at New York from day 0 to day 2 inclusive. Its transaction time starts at day 1, ends when there is an update to the tuple. The attribute pid is a time-invariant key, and city is a time-varying key. Valid time ranges of the same invariant key could overlap, although none in the tourist example does.

tuple	pid	city	period	trans_time
t1	p1	New York	[0, 2]	1
t2	p2	Washington	[5, now]	1
t3	p3	New York	[0, 6]	1
t4	p4	Detroit	[0, 7]	2
t5	p5	Washington	[4, 6]	2
t6	p5	New York	[7, now]	3
t7	p6	Seattle	[3, now]	3
t8	p4	Washington	[10, now]	3
t9	p3	New York	[12, now]	3
t10	p1	Los Angeles	[3, 6]	3
t11	p7	New York	[14, now]	4
t12	p1	Detroit	[7, 9]	4
t13	p1	Detroit	[10, 12]	5
t14	p9	Los Angeles	[3, 8]	6
t15	p1	San Francisco	[13, now]	6

Table 1: A sample *tourist* relation

.

Our set of canonical temporal queries is built on the set defined in [S94], by further classifying queries in each category based on the search predicates – *intersection, inclusion, containment and point*. Such finer classification allows us to study the effectiveness of the indexes on different kinds of search predicates. The following constitutes the basic set of queries:

Q1) Time-slice queries: Find all versions that are valid during the given time interval [T_s, T_e]. Based on the search operation on the temporal index, time-slice queries can be further classified as:

T1) Intersection Queries: Given a time interval [T_s, T_e], retrieve all the versions whose time intervals intersect it. For example, a query to find all tourists who are in US during the interval [3,7] would return 9 tuples: t2,t3,t4,t5,t6,t7,t10,t12 and t14.

T2) Inclusion Queries: Given a time interval [T_s, T_e], retrieve all the versions whose time intervals are included in it. For example, the query "Find all tourists who stay in a city between day 3 and day 7" would return 2 tuples: t5 and t10.

T3) Containment Queries: Given a time interval [T_s, T_e], retrieve all the versions whose time intervals contain it. For example, the query "Find all tourists who stay in a city from day 3 to day 5" would result in 5 tuples: t3,t4,t7,t10 and t14.

T4) Point Queries: Given a specific time point t(instant), retrieve all the versions whose time intervals contain the time point. Point queries can be viewed as the special case of intersection queries or containment queries when the time interval [T_s, T_e] is reduced to a single time instant T. For example, the query "Find all tourists who are in US on day 1" would result in 3 tuples: t1, t3 and t4.

Q2) Key-range time-slice Queries: Find all records which are in a given key range [k_s, k_e] that are valid during the given time interval [T_s, T_e]. It is a conjunction of keys and time. Like the time-slice query, the time-slice part of the query can assume one of the predicates described in T1 - T4. Note that the key can either be a time-invariant attribute or a time-varying attribute, and the time can either be valid time or transaction time. The search remains the same although the semantics may be subtlely different. For example, the query to find all tourists who are in New York during the interval [3,7] is a key-range time-slice query with intersection predicate. The result of the query is now 2 tuples instead: t3 and t6. As another example, the query "Retrieve all tourists who are in cities with names beginning in the range [D,N] on day 1" would be a point key-range time-slice query that results in 3 tuples: t1, t3 and t4.

Q3) Past Versions Queries: Find all the historical versions of the records in the given key range [k_s, k_e]. Such a query is independent of the temporal range. For example, the query "Find all tourists who visited New York" is a past versions query. This query will return the tuples: t1,t3,t6,t9 and t11.

Q4) Exact-Match Queries: Find the versions of the record with key k at time t. An example of this category is "Find all tourists who visited New York on day 1", and results in tuples: t1 and t3.

To answer time-slice queries, the index must be able to support retrieval based on time. For key-range time-slice queries, the search should be based on both key and line segments. To support valid time, an index must support dynamic addition, deletion and update of data on the time-dimension, and support time that is beyond the current time. In other words, retroactive and proactive updates are required. An index that has been designed for valid time can be easily extended for transaction time even though a transaction database can be thought of as an evolving collection of objects The major differences are that (1) delete operation is not required for transaction time databases, and (2) time increases on one end dynamically as it progresses. However, it is much more difficult to extend a transaction time index for indexing valid time data since transaction time indexes are designed based on the fact that transaction times do not overlap, and such property is quite often built into the index. Further, some transaction time indexes assume that intervals are always appended from the current time.

3 Temporal Indexes

Now, let us review the four indexes we study, the Time-Split B-tree, the AP-tree, the R-tree and the TP-tree. The first two are selected as representatives of indexes that are based on B^+-tree, while the last two are multi-dimensional indexes. Further, the TSB-tree allows us to test on the effectiveness and efficiency of duplicating records.

3.1 The Time-Split B-Tree

The Time-Split B-Tree (TSB-tree) [LS89, LS90, LS93] is a variant of the Write-Once B-Tree (WOBT) [E86]. The TSB-tree is one of the first temporal indexes that support search based on key attribute and transaction time. An internal node contains entries of the form <*att-value*, *trans-time*, *Ptr*>, where *att-value* is the time-invariant attribute value of a record, *trans-time* is the timestamp of the record and *Ptr* is a pointer pointing to a child node [LS89].

Searching algorithms are affected by how a node is split and the information it captures about its data space. Therefore, we shall begin by looking at the splitting strategy. In the TSB-tree, two types of node splits are supported, key value and time splits. A key split is similar to a node split in a conventional B^+-tree where a partition is made based on a key value. For the time split, an appropriate time is selected to partition a node into two. Unlike key split, all record entries that persist through the split time are replicated in the new node which stores entries with time greater than the split time. Figure 1 shows the TSB-tree time splitting in which the record <p1, Detroit, 4> is duplicated in the historical and new nodes. If the number of different attribute values in a node is more than $\lfloor M/2 \rfloor$ (M is the number of entries in a node), a key split is performed; otherwise the node is split based on time. If no split time can be used except the lowest time value among the index item, a key split is executed instead of time split. To search based on key and time, index keys and times of internal nodes are used respectively to guide the search. With data replication, data whose time intersects the data space defined in the index entries are properly contained in its subtree, and this enables very fast search space pruning.

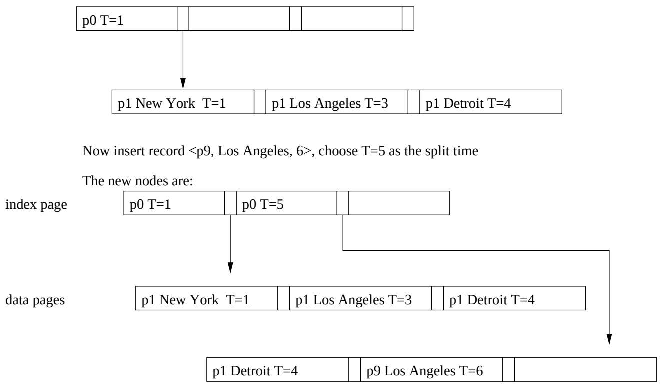


Figure 1: Time splitting in TSB-tree.

The TSB-tree only supports transaction times in the sense that times of the same invariant key must strictly be in increasing order. In other words, there is no time overlapping among versions of a record. When a record is updated, the existing record becomes a historical record, and a new version of the record is inserted. The TSB-tree can answer all the canonical queries.

For each key K_i, the TSB-tree keeps all timestamps $\{T_{t1}, T_{t2} \ldots T_{tn}\}$ ($T_{t1} < T_{t2} < \ldots < T_{tn}$) of the key in its index nodes. For the TSB-tree to support the time-interval query predicates, we assume that a record with $<K_i, T_{tj}>$ ($T_{tj} \in \{T_{t1}, T_{t2} \ldots T_{tn-1}\}$) is valid in the time interval $[T_{tj}, T_{tj+1}]$, and a record with $<K_i, T_{tn}>$ is valid in $[T_{tn}, NOW]$.

3.2 Append-only Tree

The Append-only tree (AP-tree) [GS93] introduced by Gunadhi and Segev is a straightforward extension of the B^+-tree for indexing append-only valid time data. In an AP-tree, leaf nodes of the tree contain all the start times of a temporal relation. In a non-leaf node, pointer of each time value points to a child node in which this time value is the smallest value (this rule does not apply to the first child node of each index node.).

Since both the update of an existing record and insertion of a new record will only cause incremental append to the database, every insertion to the AP-tree will always be performed directly at the rightmost leaf node. All the subtrees but the rightmost one of the AP-tree are 100% full. When the rightmost leaf node is full, the node is not split, instead a new rightmost leaf node is created and is attached to the most appropriate ancestor node. To search for a record whose interval falls within a given time interval, we need to use the end time of the search interval to get the leaf node that contains the record whose start time is just before the search end time. ¿From that node, we scan the leaf nodes on its left.

To answer queries involving both key and time-slice, a two-level index tree called the Nested ST-Tree (NST) was proposed. The first level of an NST is a B^+-tree that indexes key values, and the second level is an AP-tree that indexes temporal data that correspond to records with the same key value. In the B^+-tree, each leaf node entry has two pointers, with one pointing to the current version of the record with this key, and the other pointing to the root node of the AP subtree. A query involving only key value can directly access the most recent version of the record through the B^+-tree. Figure 2 shows the structure of the NST. An index structure similar to the NST was also proposed to index time-varying attribute and time. Since the temporal attribute is not unique, the qualified tuples will overlap over their associated time intervals. Several approaches for partitioning tuples along the time dimension were proposed to minimize the overlap.

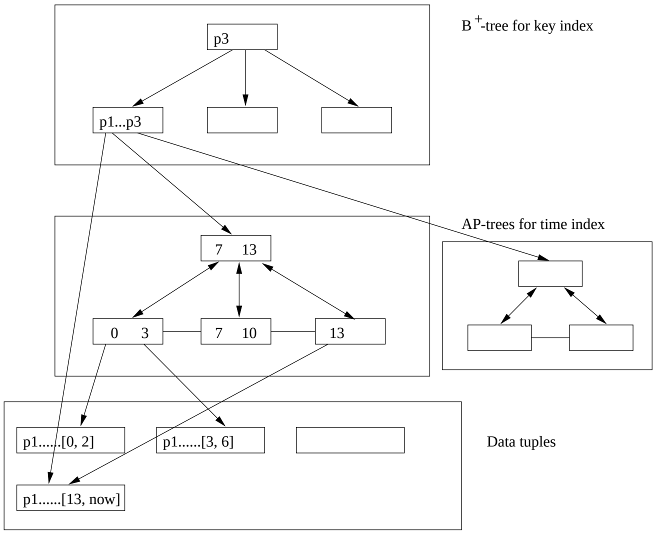


Figure 2: A Nested ST-tree structure

The AP-tree only supports monotonic appending with incremental time value. Therefore, the multiplicity of the update operations will be limited. The basic AP-tree itself can support queries involving only time-slice. A more expensive structure such as the NST has to be used to answer key-time queries. In our experiment, the AP-tree is used for the time-slice query since it is clearly more efficient than the NST-tree. For the key-range time-slice and past versions queries, we use the NST-tree. However, we use the name AP-tree to refer to either of them.

3.3 The R-tree

The R-tree [G84] is a multi-dimensional generalization of the B-tree, that preserves the height-balance property. The R-tree has received a great deal of attention due to its well defined and efficient structure for non-zero sized spatial object indexing.

An entry in a leaf node consists of an record-identifier of the data object and a k-dimensional bounding rectangle which bounds its data objects. In a non-leaf node, an entry contains a child-pointer pointing to a lower level node in the R-tree and a bounding rectangle covering all the rectangles in the lower nodes in the subtree.

In order to locate all objects which intersect a query rectangle, the search algorithm descends the tree from the root. The algorithm recursively traverses down the subtrees of bounding rectangles that intersect the query rectangle. When a leaf node is reached, bounding rectangles are tested against the query rectangle and their objects are fetched for testing if they intersect the query rectangle. It is quite common that multiple paths are searched. Therefore, the constraint of least coverage is employed during object insertion to reduce overlap between bounding rectangles.

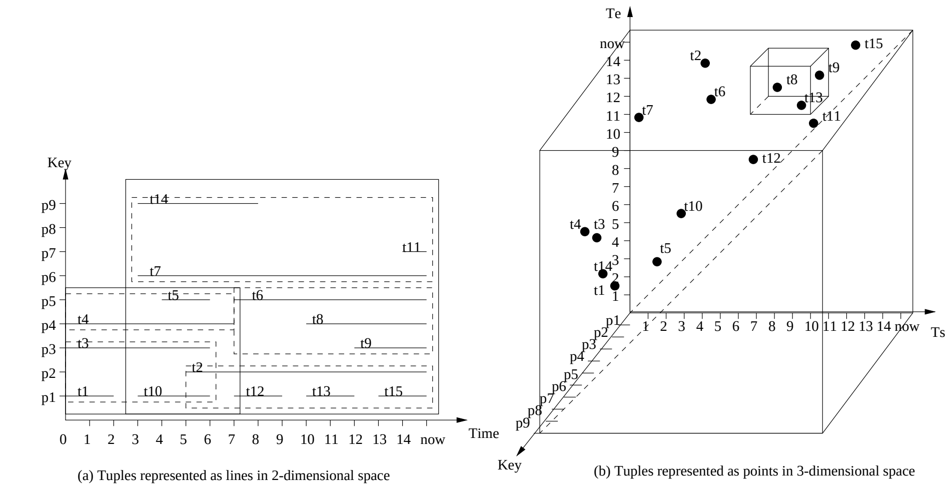
Kolovson and Stonebraker adapted the R-tree to index historical data. The structure stores some of the nodes on the optical disk [KS91, K93]. In the case that the interval data collections have non-uniform length distributions, overlap between bounding rectangles can be quite severe due to some long intervals. To overcome this shortcoming, the Segment R-tree (SR-tree) was proposed to store interval records in both non-leaf nodes and leaf nodes. Data replication and reduction in the fan-out of internal nodes may increase the height of the tree and hence affect its search performance.

As noted in the introduction, we can use a 2-dimensional R-tree (2-D R-tree) or 3-dimensional R-tree (3-D R-tree) to index temporal data and its key. To use a 2-D R-tree, time intervals [T_s, T_e] are treated as line segments in a two-dimensional space, with keys on the other dimension. To index temporal data using a 3-D R-tree, we have to map the time intervals and keys into points (key, T_s, T_e) in a 3-dimensional space. Figure 3 shows the two structures for the *tourist* relation.

Both indexes can handle the pure time query, key-time query and pure key query of the set of canonical queries. For the 2-D R-tree, all searches are implemented as intersection search. For the 3-D R-tree, search intervals must be mapped into the search regions in the triangular space. Figure 4 shows the query regions on the time dimension for the four search operations. As an example, consider the intersection search. Let the query time interval be [QT_s, QT_e]. For an interval in the database to intersect the query interval, either its end time must be in the interval or its start time must be in the interval. Thus, no record with end time less than QT_s needs to be considered, and no record with start time after QT_e needs to be examined. We then have the query region as indicated by the shaded portion.

3.4 The Time-Polygon Index

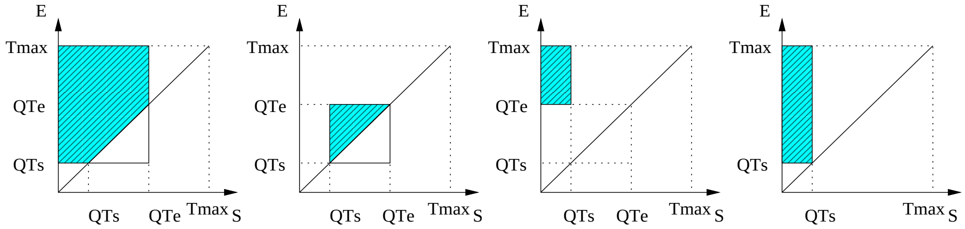
The Time-Polygon Index (TP-Index) [SOL94] was proposed to index temporal database based on valid time. The essential idea of the TP-Index is to map the time interval [T_s, T_e] into a point [T_s, $T_e - T_s$] in a triangular 2-dimensional space. Figure 6 illustrates the transformation of the time interval to the spatial representation for the *tourist* relation. The X-axis denotes the discrete time



(a) Tuples represented as lines in 2-dimensional space

(b) Tuples represented as points in 3-dimensional space

Figure 3: Space partitioning in the R-tree



(a) Intersection search

(b) Inclusion search

(c) Containment search

(d) Point search

Figure 4: Query regions for R-tree on the time dimension

points in the interval [0, *now*], and the Y-axis represents the time duration of a tuple. The points on the line named *time frontier* represent tuples with ending time of *now*. The time frontier will move dynamically along with the progress of time.

The triangular temporal space is partitioned into some groups such that each group is the cluster of data points suited for a certain search pattern. Partitioning along X- and Y-dimensions, and parallel to the time frontier produce five polygonal shapes as shown in Figure 5. Polygons used in the TP-index are not minimum bounding polygons. The polygons are derived through recursive partitioning, and can be easily merged when the tree is collapsed. The structure of the TP-index is like that of an R-tree. Figure 6 shows the partition of the temporal space and the TP-tree structure of the *tourist* relation. To support proactive additions of records (eg. T_n in Figure 6(a)), a virtual time frontier that assumes the largest T_e (T_{max}) has to be introduced, and partitions that are adjacent to the time frontier have to be extended outward.

The TP-index was designed solely to index valid time and handle time-interval access in temporal database. To enable the TP-index to support the time-invariant key, we need to add a third

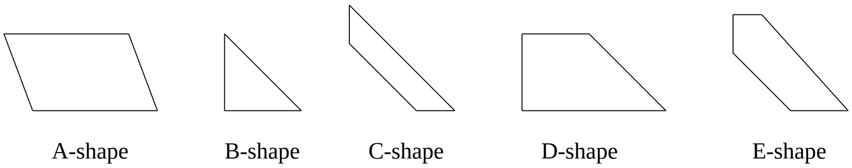
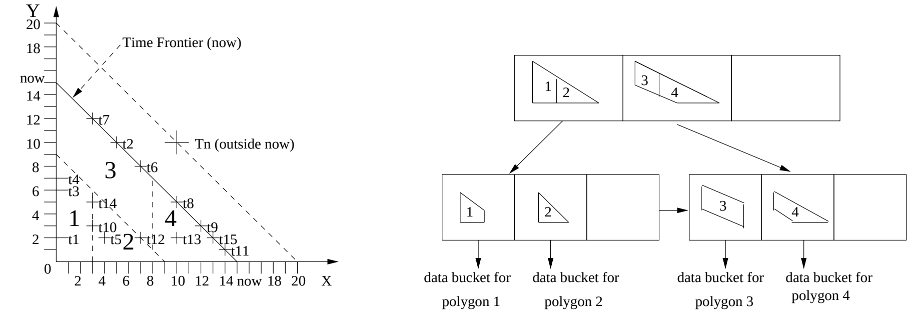


Figure 5: The five polygon shapes in TP-tree



(a) Partitioning of the temporal space for tourist relation

(b) An TP-tree structure

Figure 6: An TP-tree for *tourist* relation

dimension into it and thus extend it to index data in a 3-dimensional space. In the data space, the X-axis and Y-axis hold the same definitions as before; the Z-axis denotes the key value of the data points in the space (see Figure 7). ¿From now on, we shall simply use the term TP-tree to refer to the extended TP-index.

Initially, data points are bounded in the three-dimensional temporal space. When overflow occurs, these data points are partitioned into groups such that each group can be stored in one data page. Partitions must cluster the data points to be suited for temporal search patterns. There are three partitions for the TP-tree: 1) y-partition: introducing a plane parallel to the x-z plane (called the y-plane), 2) time-partition: introducing a plane parallel to the time frontier (called the time-plane), 3) key-partition: introducing a plane parallel to the x-y plane (called the key-plane). The y-partition and time-partition for different bounding polygons are similar to those described in [SOL94]. Note that after the key-partition, the shapes of the resultant bounding polygons are the same as that before the partitioning. Searching based on time is similar to that proposed in [SOL94], where search time intervals must be mapped into appropriate query regions. The query regions for the various search operations on the time-dimension are shown in Figure 8. For example, consider the query interval [QT_s, QT_e] for an inclusion search. Since all matching intervals must start from QT_s, those intervals that start before QT_s should be excluded. Similarly, since the query interval ends at QT_e, all intervals that end after QT_e should be excluded. The resultant query region is thus the shaded region as shown in the figure.

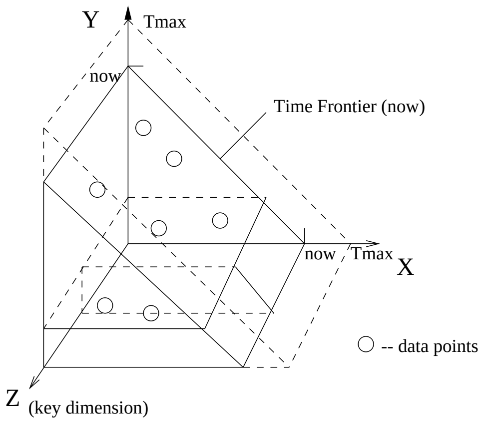


Figure 7: A three-dimensional spatial rendition of the TP-Tree

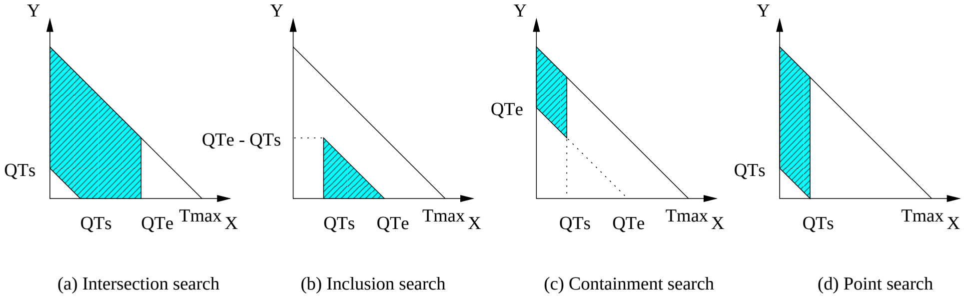


Figure 8: Query regions for the TP-tree

4 Implementation and Experimental Setup

In this section, we describe our implementation details and the experimental setup. In particular, we shall discuss a buffer management scheme, and generation of data and queries. Table 2 summarizes the parameters and their values used in our experiments.

4.1 Implementation of Index and Buffer Management

We implemented the five indexes, the TSB-tree, AP-tree, 2-D R-tree, 3-D R-tree and TP-tree, in C on a SUN SPARC workstation[1]. Depending on the experimental study, the indexes are built on different indexing attributes. In our study, we have built the indexes on time-invariant key and transaction time, time-invariant key and valid time, and time-varying key and valid time. We have also used two node sizes, 2K bytes and 4K bytes.

For a million versions, a temporal index can be fairly large, and hence it is unlikely that the

[1]Following suggestions from Professor R. Snodgrass, the codes for these indexes are available at http://dragon.iscs.nus.edu.sg:8080/~research

Parameter	Default Values	Variations
System Parameters		
index node size	4K page	2K page
buffer size	32 pages	
Database Parameters		
number of records	1000000	
size of time-invariant key	4 bytes	
size of time unit	4 bytes	
interarrival arrival time follows Poisson distribution		
with mean λ	5	2
version duration follows Poisson distribution	200	20, 100,
with mean μ		500, 1000
time-invariant attribute follows uniform distribution	[1,10000]	
time-varying attribute follows uniform distribution	[1,100000]	
Query Parameters		
number of queries per canonical query type	100	
time-invariant attribute (uniformly distributed)	[1,10000]	
time-varying attribute (uniformly distributed)	[1,100000]	
attribute range	1000 (10% of domain)	100 - 5000
time range	15000	1000 - 30000

Table 2: Parameters and their values.

entirety of an index fits in memory. Instead, some index pages are paged out as we traverse the tree, and re-fetched at a later time when they are re-referenced. To reduce page re-fetching, we implemented a priority-based buffer replacement strategy [COL92]. The priority-based page replacement strategy employs the *least useful policy* (LUF policy), which has been designed based on the way an index is traversed. We also extended the replacement algorithm for two-level index structures such as the NST.

Under the strategy, priority is assigned to index pages. An index page is *useful* if it will be referenced again in a traversal of an index structure; otherwise, the index page is *useless* in the current traversal. The strategy works as follows:

1) Useless index pages are replaced before useful pages.

2) Among useless index pages, those that correspond to index nodes in a lower level (with smaller level number) are replaced before those in an upper level (with larger level number). Ties among useless index pages at the same level are resolved arbitrarily.

3) Among useful index pages, those that have larger level numbers are replaced before those that have smaller level number. This is because in a depth-first traversal, a useful page further from the root level is more recently accessed, and therefore will be re-accessed sooner than one that is nearer to the root level.

Due to the large amount of experiments we have conducted on different queries, we only present performance results for one buffer size. In other words, we do not study the effect of buffer on the efficiency of indexes. The main concern here is to minimize the page re-fetching effect on our performance comparison. The buffer size for our buffer management system is fixed at 32 pages, which is sufficient for traversing the trees with height of up to 5 levels.

4.2 Generation of Data

To ensure that the data generated are realistic, we use an employee movement application as our model. Consider a database maintained by an international company to keep track of its employees' movement. An EMPLOYEE_MOVEMENT relation for this database has the form of <EmpNo, City, Allowances, T_s, T_e>, in which the employee number, EmpNo, is an unique key which is time-invariant, and the valid time interval [T_s, T_e] represents the period in which the employee is on an assignment to a city. City, in this example, is a time-varying attribute. The transaction time interval which is not reflected is defined by the time when the assignment form is filled and the time just before the next assignment. Depending on when an assignment is filled, the start time of a valid time and transaction time may be far apart. Such an application is common in large international organizations, where their employees may be expected to visit other cities (locally or internationally) for businesses or training. To produce data that exhibits such characteristics, we use Poisson distribution function with various mean inter-arrival time and mean duration time to simulate the rate at which employees are assigned to visit different cities and the corresponding duration of the visit. To test the effectiveness of the indexes meaningfully, we must generate data that simulate such a database. We used and extended the Time-Integrated Testbed of the Department of Computer Science, University of Arizona, to generate temporal relations with different distributions in arrival time (start time of an interval) and version duration.

Each database contains 1,000,000 versions. The time-invariant attribute is uniformly distributed over [1,10000], and each attribute value has at most 1000 versions. For each version, we also generate its time-varying attribute value, with a value in [1,100000]. For the first version, the start time and duration are randomly generated. Subsequent versions of the same key value have their start times following the end times of the previous versions while the durations are randomly generated. For the experiments, we index on either the time-invariant or time-varying attribute and the temporal attribute.

For each different set of mean arrival and duration time, we generate two sets of data, one with the constraint that the time intervals of a time-invariant key do not overlap (i.e., two versions of a time-invariant key do not have overlap in their time), and the start time of a version continues from the end time of previous version, and another with intervals may be disjoint or overlap. The first data set is used to compare the indexes that are built on the time-invariant key and transaction time. The second data set is used to study the performance of the indexes that are built on time-invariant key and valid time, and time-varying key and valid time. For valid time and time-varying key indexing, we assign a time-varying value for each version.

4.3 Generation of Queries

In our study, we generated and used the set of canonical queries to study the query performance of the indexing structures. Four sets of time-slice queries (i.e., intersection search, inclusion search, containment search and point search), four sets of key-range time-slice queries (i.e., key-range intersection search, key-range inclusion search, key-range containment search and key-range point search), and a set of past versions queries were generated. We did not study an exact match query, since it is similar to conventional search. Each set of queries contains 100 queries with different keys and time ranges. The keys are randomly picked from its domain (i.e., [1,10000] for time-invariant key, and [1,100000] for time-varying key). Where there is a key-range search, a predetermined fixed range is used to determine the end of the range. For time ranges, the starting time is generated using the Poisson distribution, and the duration is predetermined and fixed. Should the ending time exceeds the current time, then the ending time is set to the current time.

4.4 Experimental Setup

We conducted three sets of experiments. The first studied the performance of the indexes built on time-invariant key and transaction time. The second set's indexes are based on the time-invariant key and valid time. The final set is performed on indexes constructed on the time-varying key and valid time. For each set of experiments, we used two different mean inter-arrival times (λ), and three different mean duration (μ) to generate 6 databases with (λ, μ) of (2, 20), (2, 100), (2, 200), (5, 20), (5, 100) and (5, 200).

For each database, we queried the database with the generated queries, and recorded the number of page accesses for each query. The average number of page accesses for each set of (100) queries is used to evaluate the indexes for that category. In this way, we can figure out the kind of queries that a particular index may be effective or ineffective in handling. We also looked at the storage efficiency of the various techniques.

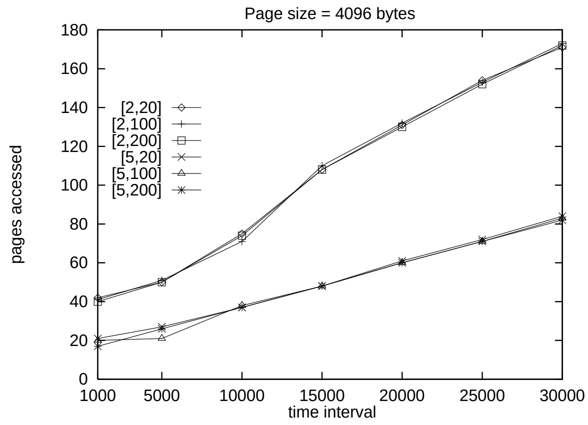
Due to the large number of experiments conducted on different databases and different queries, we can only present some of the more interesting representative results. For a detailed and complete study, readers are referred to [JOT96].

5 Experiment 1: Indexing Invariant Keys and Transaction Time

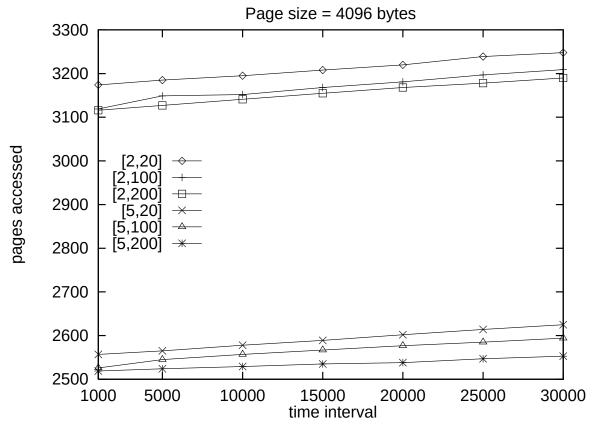
In this experiment, we study the indexing schemes based on time-invariant key and transaction time. We shall present the results based on the types of queries.

5.1 Time-Slice and Key-Range Time-Slice Intersection Search

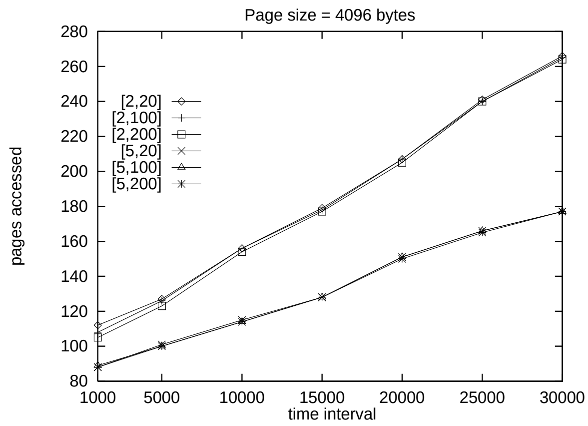
Figures 9(a)-(e) show the performance of the TSB-tree, AP-tree, TP-tree and R-trees for time-slice intersection search queries under different mean inter-arrival times and mean duration times when page size of 4K bytes is used. The performance of all five indexes is affected by the search time range we use in the query, the longer the search range the worse the performance. One important observation is that while the mean duration time has little effect on a few indexes, the inter-arrival time has significant effect on the performance of most indexes. Longer mean inter-arrival time means less overlap in time intervals. For indexes such as the TSB-tree, 3-D R-tree and TP-tree, shorter inter-arrival times mean time intervals of different keys are clustered closely, and the same search range intersects more intervals and hence more pages are accessed. Among the five indexes, the performance of the 2-D R-tree and the AP-tree are affected by the duration of time intervals. For the 2-D R-tree which indexes time intervals as line segments, the degraded performance is due to the fact that the Minimum Bounding Rectangles (MBR) in the internal nodes have more overlap



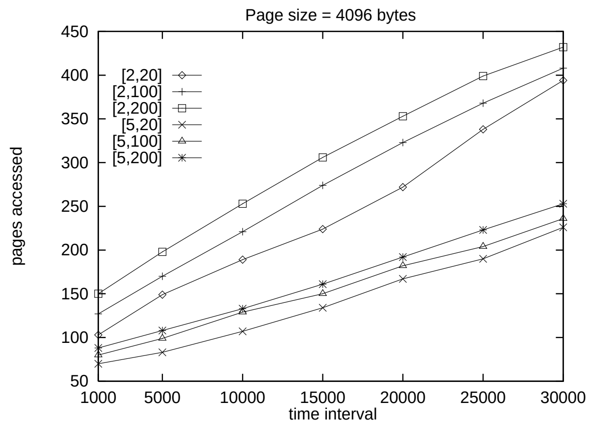
(a) TSB-tree



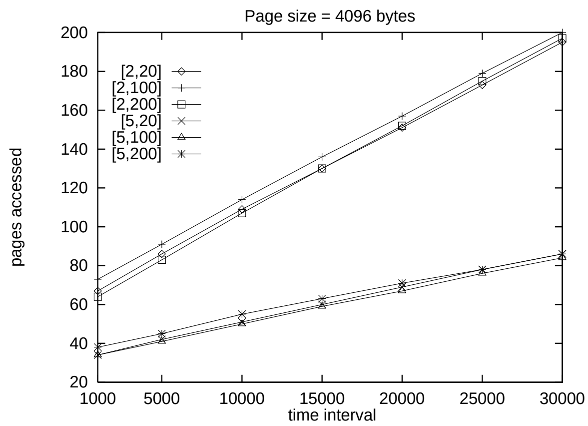
(b) AP-tree



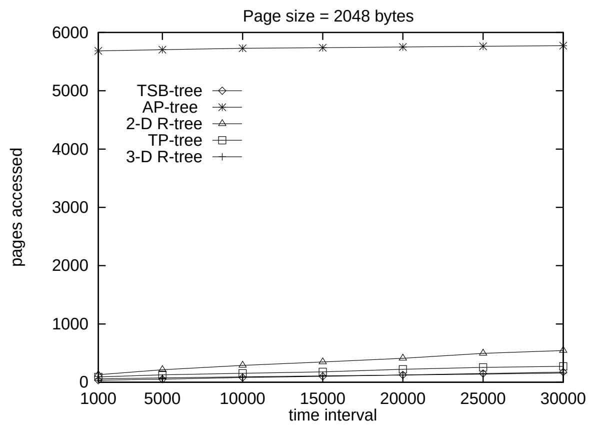
(c) TP-tree



(d) 2-D R-tree



(e) 3-D R-tree

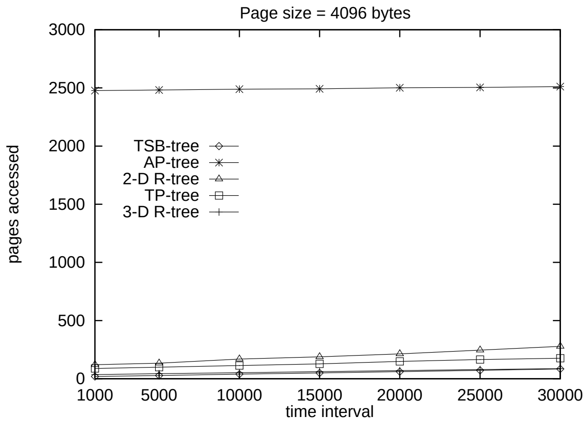


(f) $(\lambda, \mu) = (5, 200)$

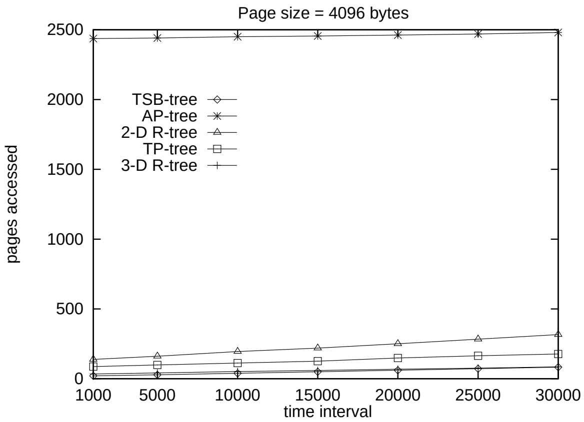
Figure 9: Performance of indexes for time-slice intersection search.

for longer line segments. For the AP-tree, the opposite effect is observed. Two factors contributed to this. First, (recall that) the data set is non-overlapping for each key value. Second, a longer duration essentially "stretches" the lifespan of the relation. As a result, the number of nodes to be scanned by AP-tree is smaller for longer duration.

¿From the figures, we can also observe that the TSB-tree performs the best. For a mean inter-arrival time of 2 time units, it requires between 40 to 175 page accesses, and for a mean inter-arrival time of 5 time units, the page accesses are from 20 to 80. This can be interpreted by the fact that the TSB-tree has a high degree of data clustering in both key and time dimensions. On the contrary, the AP-tree is inferior to all the other techniques. Its page accesses exceeds 2500 pages! This is because to search for the intervals intersecting with the query interval [T_s, T_e] in the AP-tree, a leaf node is first determined using T_e. All leaf nodes on its right, which contain intervals whose start time is larger than T_e, are ignored. Leaf nodes on its left must be searched. The 3-D R-tree is slightly better than the TP-tree, which is in turn better than the 2-D R-tree. We also make similar observations when the page size is only 2K bytes (see Figure 9(f)).



(a) $(\lambda, \mu) = (5, 500)$



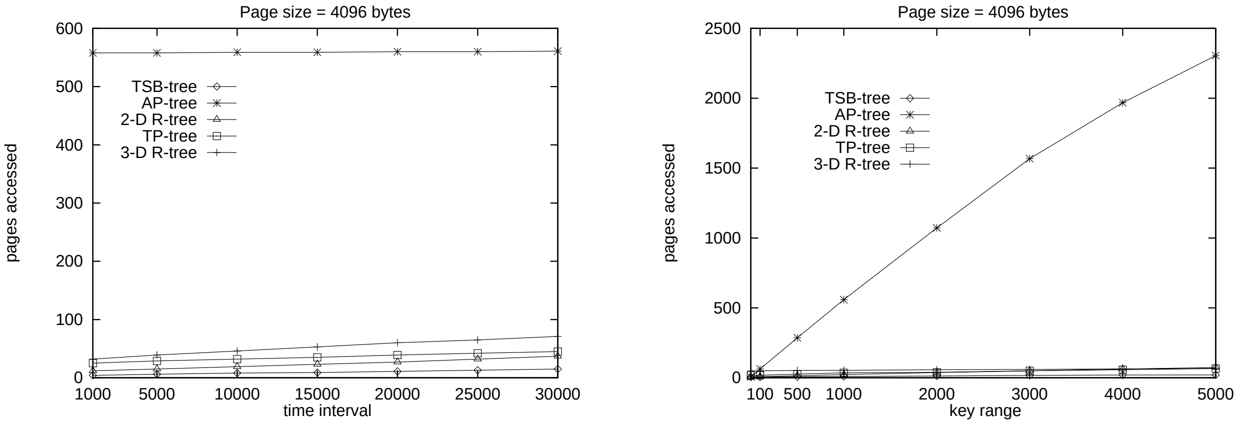
(b) $(\lambda, \mu) = (5, 1000)$

Figure 10: Effect of duration in time-slice intersection query.

To simulate the effect of long-lived versions on the performance of indexes, data with μ=500 and 1000 are generated. Figure 10 summarizes the performance of indexes under varying search time intervals. The results confirm our earlier observation that the performance of the 2-D R-tree is more sensitive to the duration. This is because long-lived intervals cause more overlap among bounding rectangles in the 2-D R-tree. For indexes that represent time intervals as points, their performance is less affected by the duration.

The results for the key-range time-slice intersection queries are very similar to that for the time-slice queries. Here, we shall present the results for the default settings where (λ, μ) has value of (5, 200). In order to see the effect of key range, we keep the query time range constant, and similarly, to see the effect of the query time range, we keep the other constant. Figure 11(a) shows the result when the key range is fixed at 1000, while Figure 11(b) looks at the effect of varying the key range when the time range is fixed at 15000 time units. Like the time-slice query results, we observe that the AP-tree is also not effective compared to the others due to its two level structure. With such a structure, each AP-tree in the second level of the nested structure is small, and many of such small trees must be searched. We also note that the AP-tree is more sensitive to the key ranges than time ranges. (see Figure 11(b)). This is logical since the first level of the nested structure is the B$^+$-tree for keys and the key range determines the number of AP-trees in the second level that need to be searched. As the key range increases, the performance deteriorates. Whereas for a fixed time range, the average number of leaf nodes that need to be searched do not differ greatly.

As for the two R-tree-based indexes, unlike previous results, the 3-D R-tree is now not as efficient as the 2-D R-tree as data clustering at a higher dimension is not as effective as that at the 2-D R-tree. The result is consistent with the recent finding in [BKK96]. The TSB-tree retains its good performance in key-range time-slice query because of its high degree of data clustering in both key and time dimensions.



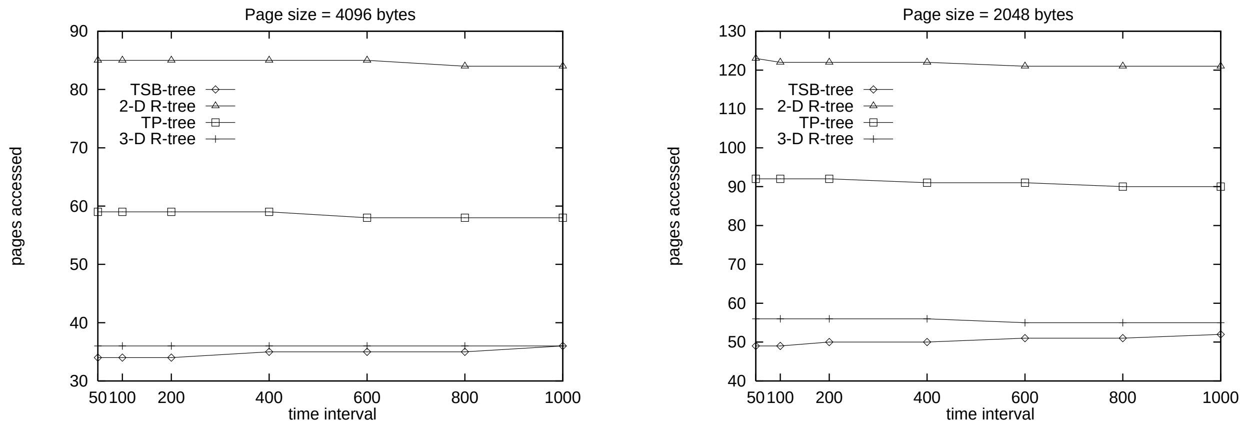
(a) Key range = 1000 (b) Time range = 15000

Figure 11: Performance of intersection search in key-range time-slice query $(\lambda, \mu) = (5, 200)$

5.2 Time-Slice and Key-Range Time-Slice Containment Search

Containment search for time-slice query is a search for all versions whose time intervals contain the given time interval. Most indexes make use of intersection search to evaluate containment search queries. For the AP-tree, the search is similar to that used in the intersection search, where a leaf node must be found using the end time of the query time interval, followed by scanning the leaf nodes from right to left. Hence we exclude the AP-tree in this experiment because of its poor performance. For containment search to be sensible, we cannot use large time intervals as before. Here, we use smaller time intervals. Experimental results for node size of 2K and 4K bytes given a fixed (λ, μ) of (5,200) are summarized in Figure 12. The results exhibit similar behavior as that of time-slice intersection search, where the TSB-tree is very efficient, followed by the 3-D R-tree, the TP-tree, and the 2-D R-tree.

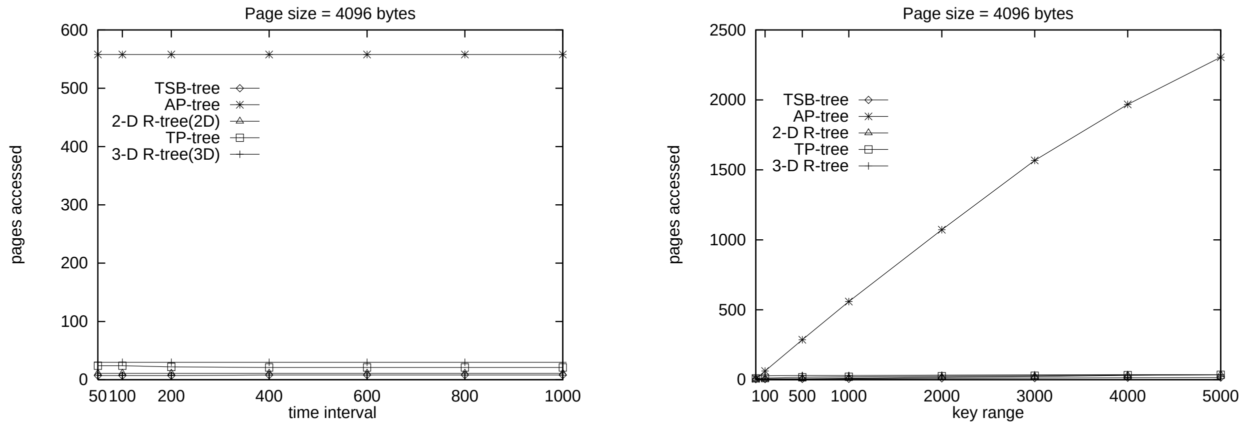
Figure 13 shows the performance of the indexes for containment key-time search. We also fix the key range to 1000 to study the effect of query time range (see Figure 13(a)), and set the query time range to 200 when we study the effect of key range. The results also support our earlier observation that the key range affects the AP-tree more than the time range.



(a) Page size = 4K bytes

(b) Page size = 2K bytes

Figure 12: Performance of time-slice containment search $(\lambda, \mu) = (5, 200)$



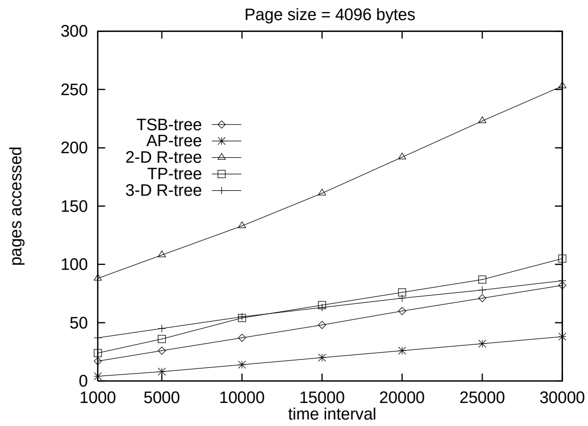
(a) Key range = 1000

(b) Time range = 200

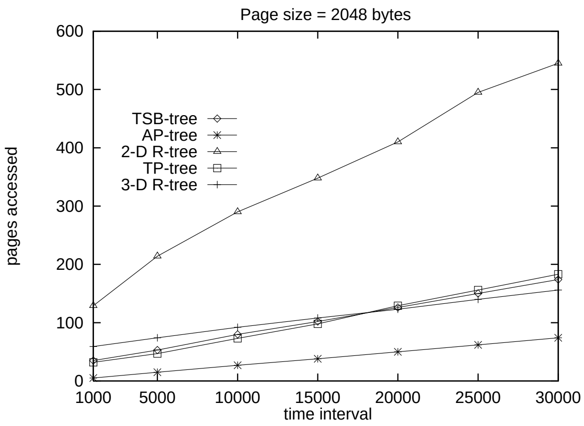
Figure 13: Performance of key-range time-slice containment search $(\lambda, \mu) = (5, 200)$

5.3 Time-Slice and Key-Range Time-Slice Inclusion Search

Quite often, we would like to search for versions that are valid within a given time interval. Unlike the containment search, it is meaningless for us to specify a search interval that is too small. In this experiment, we use the search intervals as those in the intersection experiment. The results for the two node sizes for the default setting is shown in Figure 14. The results indicate that the AP-tree performs the best for such search. For the containment and intersection searches, all the leaf nodes on the left of the leaf node that contain the T_e have to be scanned. Now, the scan is stopped when an entry with start time less than that of the query time interval. The 2-D R-tree does not perform well and is fairly sensitive to the query time range.

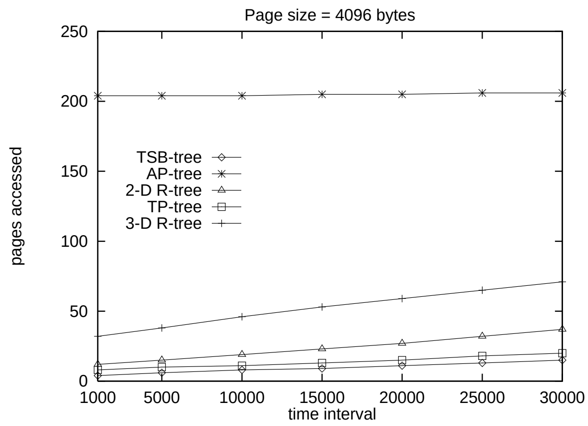


(a) Page size = 4K bytes

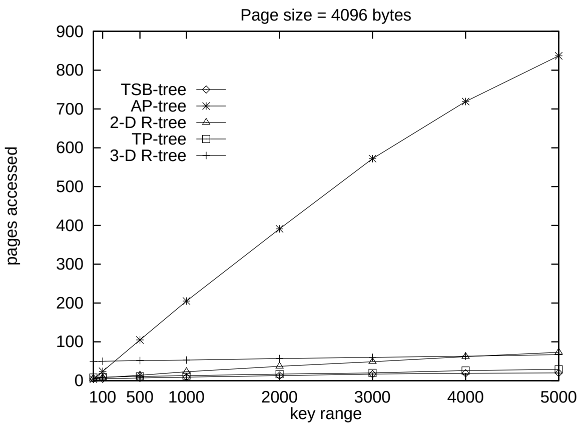


(b) Page size = 2K bytes

Figure 14: Performance of time-slice inclusion search $(\lambda, \mu) = (5, 200)$



(a) Key range = 1000



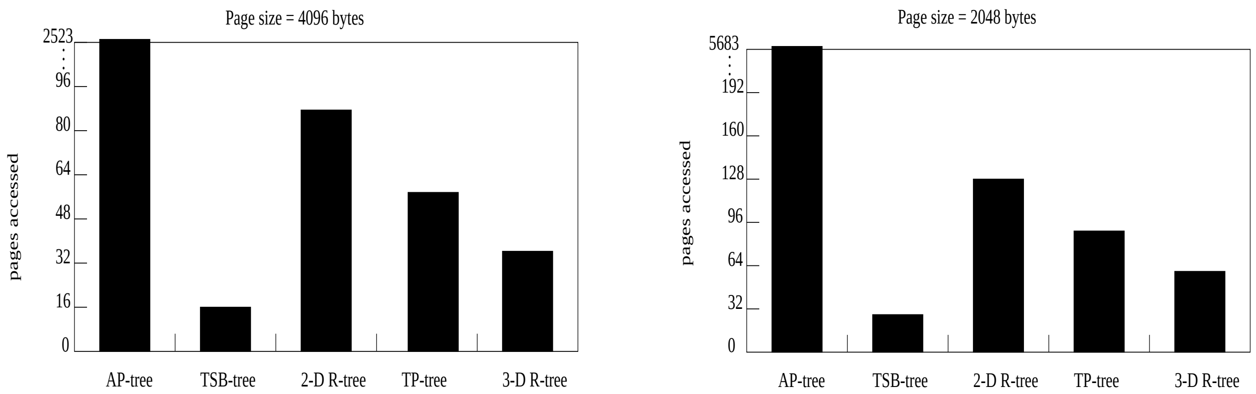
(b) Time range = 15000

Figure 15: Performance of key-range time-slice inclusion search $(\lambda, \mu) = (5, 200)$

The results of the key-range time-slice inclusion queries under the default setting is shown in Figure 15. When the time search is conjuncted with key, the effectiveness of the AP-tree is greatly reduced, since for each key value, one AP-tree has to be searched. However, the performance is still much better than that of key-range time-slice intersection search (see Figure 11). Again, the results show that the AP-tree is sensitive to key range, but not the time range. The TP-tree and TSB-tree perform better than the two R-trees. The TP-tree transforms a search interval into a search region in its 3-dimensional triangular space, and a search region for inclusion query is one of the smallest, and hence its results are much better than before.

5.4 Time-Slice and Key-Range Time-Slice Point Search

Point search is a special case of the intersection search, where the time interval is reduced to a point. In fact the results exhibit similar pattern to that of the intersection search. The result for

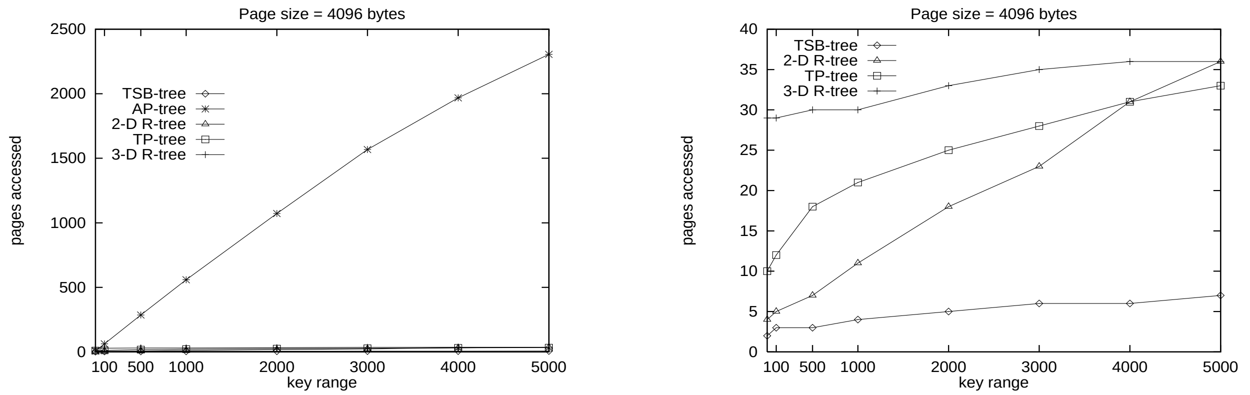


(a) Page size = 4K bytes (b) Page size = 2K bytes

Figure 16: Performance of time point search $(\lambda, \mu) = (5, 200)$

the default setting for time-slice point search, as shown in Figure 16, shows that the AP-tree needs much more page accesses for the same reason as in the intersection search. The TSB-tree performs better than the others.

In the key-range time-slice point search, as shown in Figure 17 for the default setting, for small key range, the query efficiency of the 2-D R-tree is relatively better than that of the 3-D R-tree due its better clustering effect. However, as the key range increases, more paths need to be searched, and that causes its page accesses to go up.

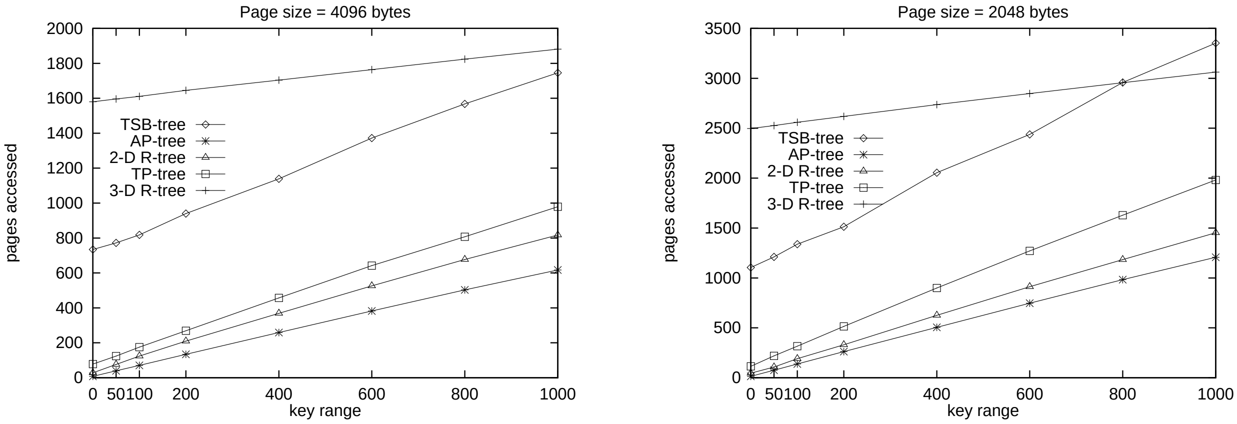


(a) Results with the AP-tree (b) Results without the AP-tree

Figure 17: Performance of key-range time-slice point search $(\lambda, \mu) = (5, 200)$

5.5 Past Versions Search

To answer past versions query efficiently, it is important to cluster data by the time-invariant key in an indexing structure. By linking all the past versions of a given key together, the best performance of this query can be expected. However, although the TSB-tree, AP-tree, R-trees and TP-tree do have some feature of data clustering by key, none of them provide an explicit method to link the historical versions of a given key. Hence, a search based on the key is required. ¿From Figure 18, it can be observed that the TP-tree is performing better than the 3-D R-tree due to its better defined data space. Once again, the performance of the 2-D R-tree is better off than the 3-D R-tree since the query space with the time dimension undefined cuts more bounding rectangles in a 3-D R-tree than in a 2-D R-tree. Similarly, the TSB-tree is not effective when one of its search dimension is not defined. The AP-tree turns out to be the most efficient for the past versions query. For each key that satisfies the search condition, the whole second level AP-tree is retrieved for all the versions. The result is consistent with key search in the previous experiments, however, the performance of the other indexes degraded very significantly and hence the contrast.



(a) Page size = 4K bytes

(b) Page size = 2K bytes

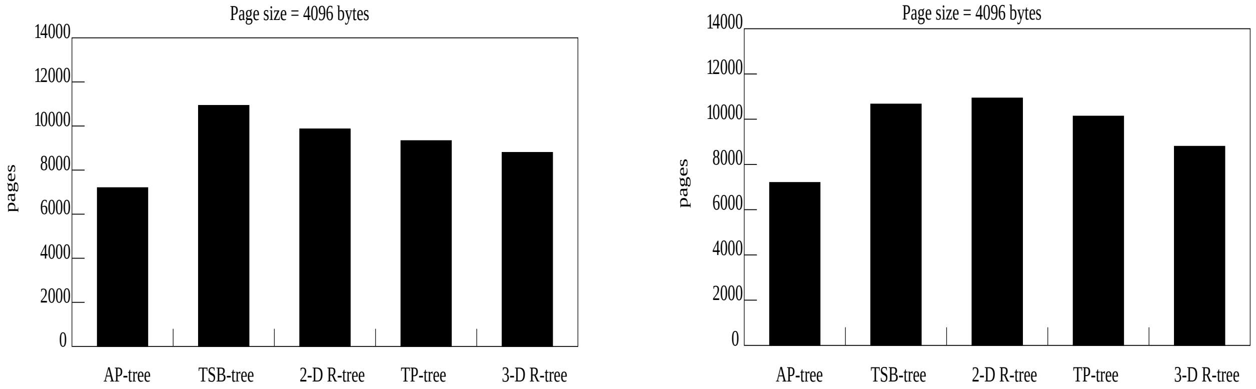
Figure 18: Performance of past version search $(\lambda, \mu) = (5, 200)$

5.6 Storage Performance

Storage efficiency is an important factor to consider in designing an index. This is because an index with poor storage efficiency is unlikely to scale well. Figure 19 shows the storage efficiency of the five indexes for two data distributions when the node size is 4K bytes. The relative storage requirements for the various indexes for the 2K page size are similar, and so the result has been left out. The AP-tree (or NST) incurs the least storage cost even though it is a two-level structure. This is because the index nodes in the AP-tree are 100% full except for the rightmost nodes. For other indexes in the experiment, the index nodes are around 70% full. The storage cost for the TSB-tree is relatively high. The TSB-tree incurs the highest storage cost due to the data replication technique it employs. It can also be observed that the 3-D R-tree requires less storage cost than the 2-D R-tree, and the storage cost for the TP-tree is between those of the 2-D R-tree and 3-D R-tree. The storage requirement for the TSB-tree, the two R-trees and the TP-tree vary little with the mean inter-arrival time(λ) and the mean duration time(μ), while the AP-tree's remains constant.

5.7 Insertion Cost

In order to obtain the insertion cost for each of the temporal indexes, we make 1 million tuples insertions to the database. We compute the cost of inserting 1000 tuples after every 100000 tuples have been inserted. The keys inserted are in the range of 1 to 100000. The timestamps is in arbitrary monotonically increasing numbers in order to accomodate insertions to the AP-tree. Deletions are

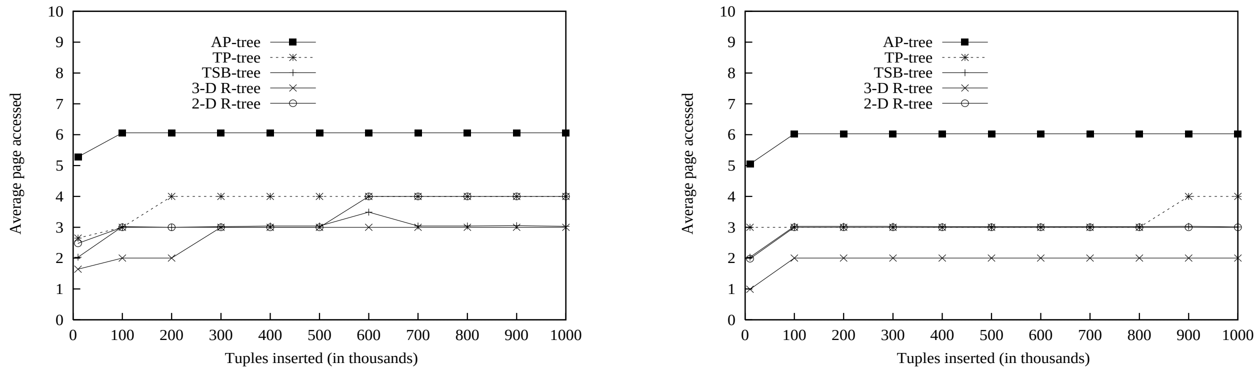


(a) $(\lambda, \mu) = (5, 200)$.

(b) $(\lambda, \mu) = (2, 20)$

Figure 19: Storage costs for the indexing structures (4 KB)

not considered in our experiment study since some of the indexes such as the TSB-tree and the AP-tree support applications where data is mainly updated and infrequently deleted. We use two page sizes, 2K and 4K, in order to show the insertion costs of the various indexes as the trees grow in height. The results shown in Figure 20 shows that the insertion costs for the five indexes are relatively constant when the fanout is high. The AP-tree incurs the highest insertion cost because our experiment uses the more expensive two-level NST structure. Inserting into the first level which is a B+ tree takes an average of 4 page access per insertion while inserting into the second level which is the AP-tree takes an average of 2 page access per insertion. Both 2-D R-tree and 3-D R-tree have about the same insertion cost of 2-3 page access per insertion, which is basically the height of these trees. However, we observe that the 3-D R-tree is slightly more efficient than the 2-D R-tree. The TP-tree and the TSB-tree have insertion costs of about 3-4 page access per insertion. The spikes for the TSB-tree when the page size is 2K indicates the occurrence of page splits.



(a) Page size = 2K bytes

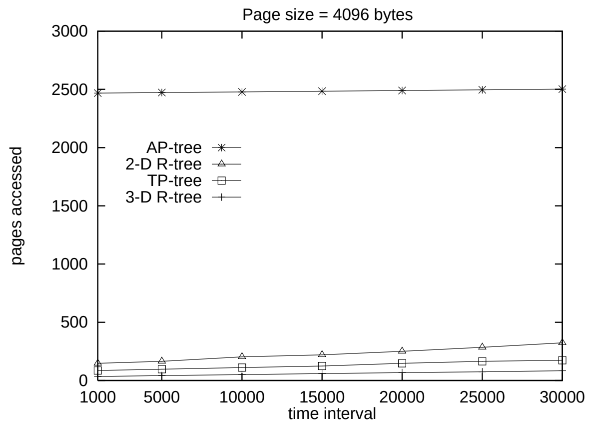
(b) Page size = 4K bytes

Figure 20: Insertion costs for the indexing structures

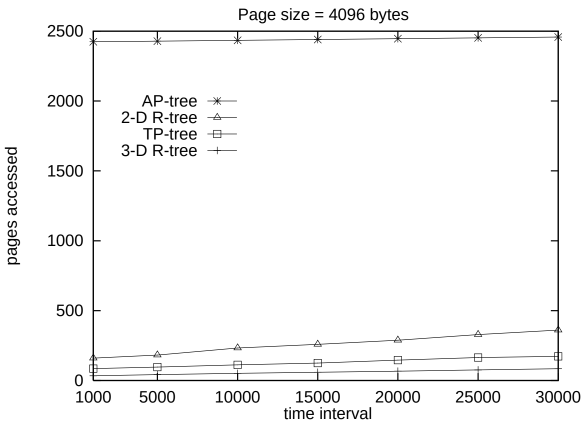
6 Experiment 2: Indexing Invariant Key and Valid Time

For the data we used in the previous experiments, we imposed the constraint that the versions do not overlap in time. For valid time databases, time can overlap, and can be pro-actively inserted or retroactively updated. However, the overlap should not be very high, and for the data we generated, we have about 5% of versions that overlap with others. For the experiments that follow, we use the same set of parameters to generate the data, but we allow some data to overlap. The same set of default settings also applies. Since the TSB-tree cannot support valid time data, we do not compare the TSB-tree in this section. Instead of presenting the whole set of experiments, we selectively present some representative results. This is because the results are more or less similar to those of transaction time. In particular, we shall leave out most of the results for key-range time-slice queries and storage efficiency because the performance results exhibit the same behavior as those for transaction time. Interested readers are referred to [JOT96].

Figure 21 shows the first performance results when an invariant key can have overlapping time intervals. The results presented are for (λ, μ) values of (5, 500) and (5, 1000) respectively and when 4K node size is used. Compared with Figure 10, the number of page accesses for the 2-D R-tree increases, while those of the TP-tree, AP-tree and 3-D R-tree have no significant changes.



(a) (λ, μ) = (5, 500)



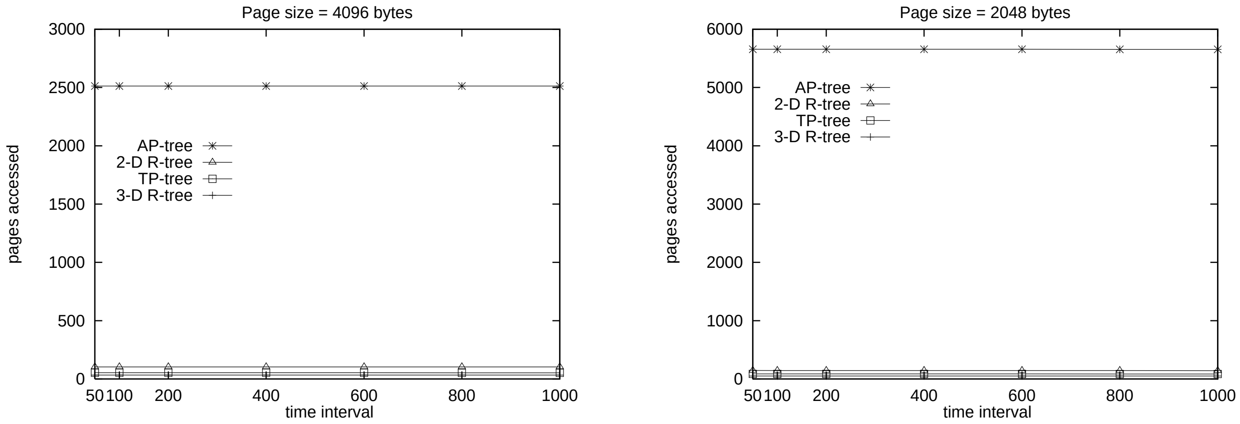
(b) (λ, μ) = (5, 1000)

Figure 21: Performance of time-slice intersection search in valid time databases

We now move on to look at the performance results for experiments on containment search in valid time databases, which is shown in Figure 22. As before, the 2-D R-tree which is affected by the density of line segments over a time point is most sensitive to valid time data. The 3-D R-tree and the TP-tree are hardly affected by overlapping temporal data.

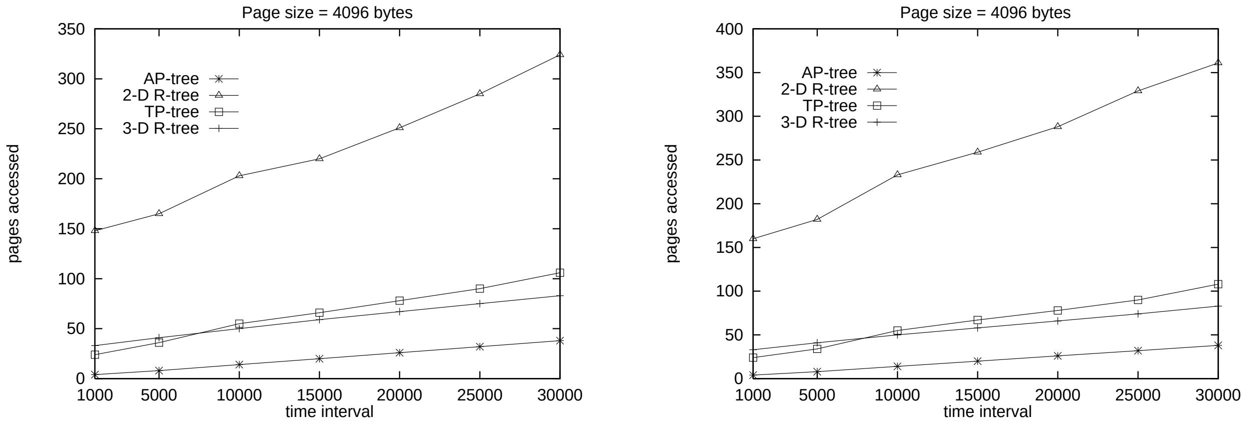
Figure 23 displays the performance of the AP-tree, R-trees and TP-tree on time-slice inclusion search when overlapping time intervals are allowed. In Figure 23(a), λ = 5 and μ = 500, and Figure 23(b), λ = 5 and μ = 1000. In both figures, a page size of 4K bytes is used. The TP-tree outperforms the 2-D R-tree and 3-D R-tree. When the mean duration time increases to a very long time (i.e., 1000), the number of intervals that overlap increases. Hence, indexes such as the 2-D R-tree which is sensitive to data density degrade in their efficiency.

Figure 24 shows the performance of inclusion search in key-range time-slice query for the default settings. The TP-tree incurs the least number of page accesses compared to both the 2-D R-tree



(a) Page size = 4K bytes (b) Page size = 2K bytes

Figure 22: Performance of time-slice containment search $(\lambda, \mu) = (5, 200)$



(a) $(\lambda, \mu) = (5, 500)$ (b) $(\lambda, \mu) = (5, 1000)$

Figure 23: Performance of time-slice inclusion search

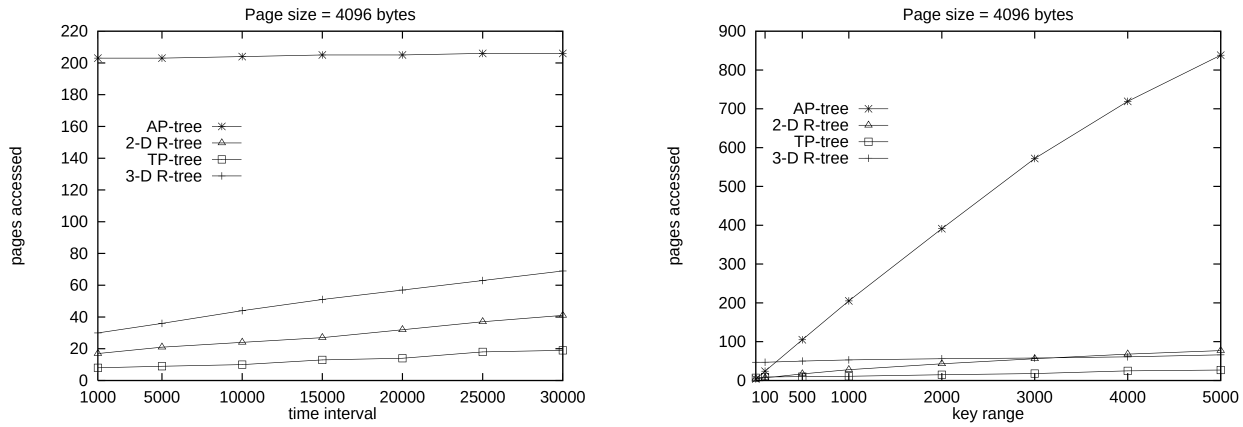
and 3-D R-tree and is comparable to the TSB-tree. The AP-tree incurs a large number of I/Os because the time sub-tree for each key has to be searched. We can observe that the number of page accesses for the 2-D R-tree increases drastically when the query interval range increases.

As shown in Figure 25, the results of the four indexes studied for time-slice point search are similar to that for the transaction time case. We have the AP-tree performing the worst, the 3-D R-tree the best, and the TP-tree is better than the 2-D R-tree.

The results for past versions search are also similar to those for transaction time, and so is not presented here.

7 Experiment 3: Indexing Time-Varying Key and Valid Time

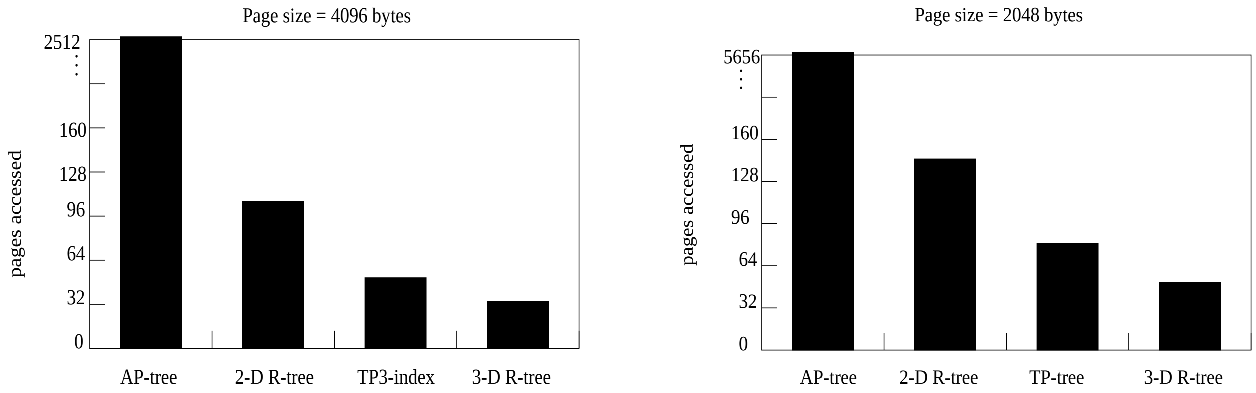
In the previous sections on our experimental study, we have used the time-invariant key for indexing. In this experiment, we indexed on the time-varying key instead [J94]. As the experiments on time-slice queries are independent of the non-temporal attribute, the results are similar to those as



(a) Key range = 1000

(b) Time range = 15000

Figure 24: Performance of key-range time-slice inclusion search $(\lambda, \mu) = (5, 200)$



(a) Page size = 4K bytes

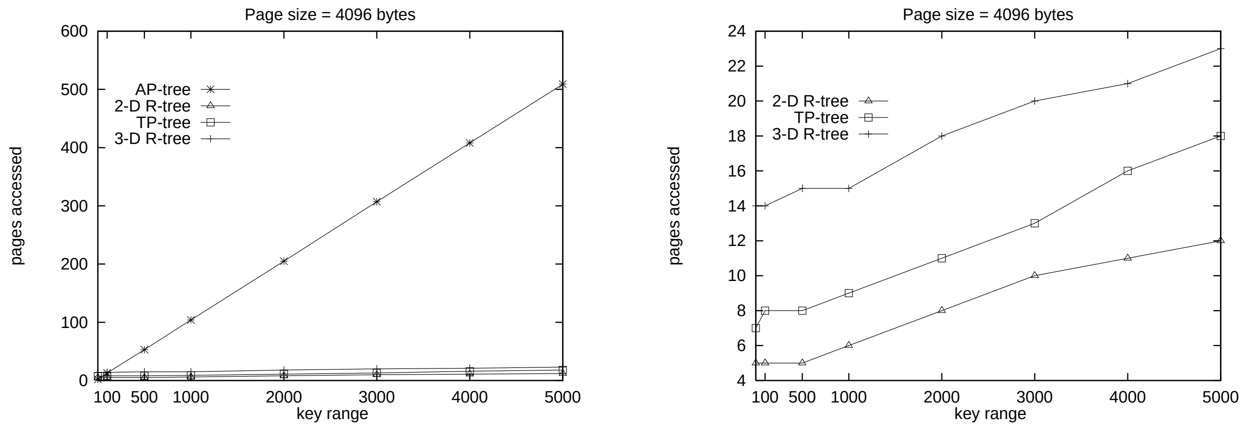
(b) Page size = 2K bytes

Figure 25: Performance of time point search $(\lambda, \mu) = (5, 200)$

the previous experiment on time-invariant key. Thus, we restrict our discussion in this section to key-range time-slice queries.

Referring to the EMPLOYEE_MOVEMENT example relation, the city attribute is the time-varying attribute. In this experiment, 10000 time-varying attribute values are randomly picked from [1, 100000], and the versions are uniformly distributed over these values. For each time-varying attribute value, we have a large number of versions that contain such value. Although each time-invariant key also has many versions associated with it, the versions are spread out over time. Here, a large number of versions with the same time-varying key may overlap in time. The TSB-tree is based on the multi-versioning concept and only indexes on the time-invariant key. For the experiments that follow, we exclude the TSB-tree, since each time-varying key has many versions with overlapping time intervals. λ and μ used are set to the default values of 5 and 200 respectively. Although we used the same query key ranges in our searches, the percentage of the range with respect to the domain range is much smaller now, and hence we expect proportionally better results.

The results of the experiment are summarized in Figures 26 and 27. We can observe that the



(a) Including AP-tree.

(b) Excluding AP-tree

Figure 26: Performance of key-range time-slice intersection search in time-varying key indexing (Time Range = 15000)

data pages accessed for AP-tree increase sharply when the query range increases, which is why the AP-tree result is removed from Figure 27. The results show that the efficiency of the TP-tree is between that of the 2-D R-tree and 3-D R-tree in key range and time intersection, containment and point searches. For key range and time inclusion search, The results show that the TP-tree is comparable to the 2-D R-tree when the query range increases.

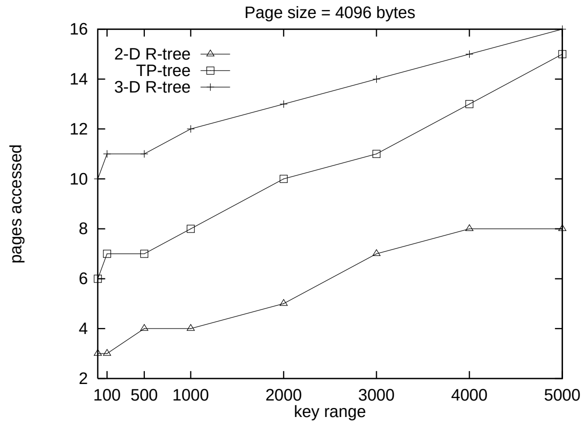
8 Conclusion

With the proliferation of temporal indexes, extensive experiments are necessary to study their efficiency. In this paper we have presented an extensive experimental study to investigate the query and storage efficiency of four representative temporal indexes, the TSB-tree, the AP-tree, the R-tree, and the TP-tree. We extended the TP-tree in order to index temporal data based on both time-invariant key and time. For the R-tree, we implemented the 2-D R-tree to index temporal data as line segments, and the 3-D R-tree to index them as points. In the experiments, we used an LUF buffer replacement strategy to retain the useful index nodes in memory as long as possible. We then made use of an extension of the TI-Time program of University of Arizona to generate different valid time and transaction time data sets that follow the Poisson distribution with various mean inter-arrival and duration times. For each set, extended canonical queries were run.

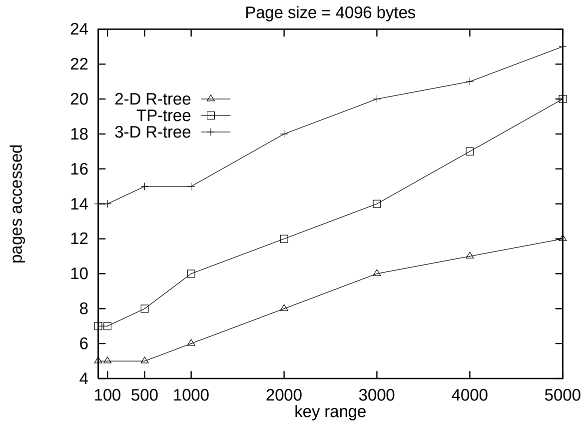
Our experimental results can be summarized as follows:

- The AP-tree is not efficient in intersection, containment and point search and the search cost can be as large as $O(n/B)$ where n is the number of intervals, and B is the page size.

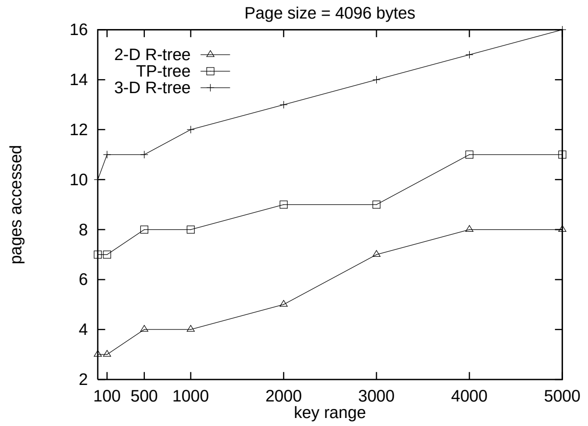
- In general, the TSB-tree which can only support transaction time, outperforms the R-tree variants and TP-tree in time-slice query and key-range time-slice query. However, its performance in past versions query is poor when the time is not defined. We note that the TSB-tree performance in past versions query can be improved if it points to previous versions as is done in the MVBT.



(a) Containment search.



(b) Inclusion search.



(c) Point search.

Figure 27: Performance of key-range time-slice containment, inclusion and point search (time range = 15000)

- The 2-D R-tree and AP-tree require less page accesses in past versions query.
- The overall performance of the TP-tree is better than the 2-D R-tree except for the past versions query, the TP-tree also outperforms the 3-D R-tree in inclusion search, past versions query and most key range time-slice query.
- In terms of storage efficiency, the TSB-tree is the least efficient due to data replication, while the AP-tree is superior.
- The insertion cost for the AP-tree is the worst due to its expensive two-level structure while the 3-D R-tree requires the least number of page access per insertion.

Surprisingly, for those indexes that are able to support both valid time and transaction time, the performance of these indexes is hardly affected by the overlap of valid time. The storage cost for the indexes is not highly affected by mean inter-arrival time and mean duration time, whereas the mean inter-arrival time and mean duration time are two important factors that affect the query performance.

Acknowledgment

We would like to thank Prof. Richard Snodgrass, Department of Computer Science, University of Arizona, for kindly providing us the Time-IT source codes. The project was in part funded by NUS Grant 950694.

References

[BGO93] B. Becker, S. Gschwind, T. Ohler, B. Seeger and P. Widmayer. On Optimal Multiversion Access Structures. *Proceedings of SSD'93*, 1993.

[BKK96] S. Berchtold, D.A. Keim and H-P Kriegel. The X-tree: An index structure for high-dimensional data. *Proceedings of the 22nd VLDB Conference*, 1996.

[COL92] Chee Yong Chan, Beng Chin Ooi and Hongjun Lu. Extensible buffer management of indexes. *Proceedings of the 18th VLDB conference*, 1992.

[C79] D. Comer. The Ubiquitous B-tree. *ACM Computing Surveys*, 11(2), 1979.

[E86] M.C. Easton. Key-sequence data sets in indelible storage. *IBM Journal of Research and Development*, 30(12), 1986.

[EJK92] Ramez Elmasri, Muhammad Jaseemuddin, Vram Kouramajian. Partitioning of Time Index for optical Department. *Proceeding of 8th International Conference on Data Engineering*, Feb. 1992.

[EWK90] Ramez Elmasri, Gene T.J. Wuu and Vram Kouramajian. An access structure for temporal data. *Proceeding of the Sixteenth Very Large Database Conference*, 1990.

[EWK93] Ramez Elmasri, Gene T.J. Wuu and Vram Kouramajian. The Time-Index and the Monotonic B^+-tree, in *Temporal Databases: Theory, Design and Implementation*, Chapter 18, A. Benjamin/Cummings, 1993.

[G84] Antonin Guttman. R-tree: A dynamic index structure for spatial searching. *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*, 1984.

[GS93] Himawan Gunadhi and Arie Segev. Efficient indexing methods for temporal relations. *IEEE transactions on knowledge and data engineering*, Vol. 5, No. 3, June, 1993.

[J94] Christian S. Jensen et al. A consensus glossary of tempes for temporal databases. *SIGMOD Record*, Vol. 23, No. 1, March 1994.

[JOT96] P. Jiang, B.C. Ooi and K.L. Tan. An experimental study of temporal indexing structures. *unpublished manuscript, http://www.iscs.nus.sg/~ooibc/tp.ps*, 1996.

[K93] Curtis Kolovson. Indexing techniques for historical databases. *Temporal databases: Theory, Design and Implementation*, Chapter 17, Benjamin/Cummings, 1993.

[KS91] Curtis Kolovson and Michael Stonebraker. Segment indexes: dynamic indexing techniques for multi-dimensional interval data. *Proceedings of the 1991 ACM SIGMOD International Conference on Management of Data*, 1991.

[KTF95] A. Kumar, V.J. Tsotras and C. Faloutsos. Access methods for bi-temporal databases. *Proceedings of the International Workshop on Temporal Databases*, September 1995.

[LM91] S. Lanka, E. Mays. Fully Persisteny B+ Trees. *Proceedings of the 1991 ACM SIGMOD International Conference on Management of Data*, 1991.

[LS89] D. Lomet and B. Salzberg. Access methods for multiversion data. *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data*, June 1989.

[LS90] D. Lomet and B. Salzberg. The performance of a multiversion access methods. *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data*, May 1990.

[LS93] D.Lomet and B. Salzberg. Transaction time databases. in *Temporal Databases: Theory, Design and Implementation*, A. Benjamin/Cummings, 1993.

[S94] B. Salzberg. On indexing spatial and temporal data. *Information Systems*, 19(6), 1994.

[ST97] B. Salzberg and V.J. Tsotras. A comparison of access methods for time evolving data. *ACM Computing Surveys*, to appear. Also available as Technical Report No. TR-18, TimeCenter, Aalborg University, 1997:
http://www.cs.auc.dk/research/DBS/tdb/TimeCenter/publications2.html.

[SOL94] H. Shen, B.C.Ooi and H.J. Lu. The TP-index: A dynamic and efficient indexing mechanism for temporal databases. *Proceeding of the 10th International Conference on Data Engineering*, February 1994.

[TK95] V.J. Tsotras and N. Kangelaris. The snapshot index: An I/O optimal access method for timeslice queries. *Information Systems*, 20(3), 1995.

[VV97] P.J. Varman and R.M. Verma. An Efficient Multiversion Access Structure. *IEEE transactions on knowledge and data engineering*, Vol. 9, No. 3, June, 1997.